Multiparty Asynchronous Session Types

KOHEI HONDA, Queen Mary University of London
NOBUKO YOSHIDA, Imperial College London
MARCO CARBONE, IT University of Copenhagen

Communication is a central elements in software development. As a potential typed foundation for structured communication-centered programming, session types have been studied over the past decade for a wide range of process calculi and programming languages, focusing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centered applications. Presented as a typed calculus for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain the friendly type syntax of binary session types while specifying dependencies and capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers and is used as a basis of efficient type-checking through its projection onto individual peers. The fundamental properties of the session type discipline, such as communication safety, progress, and session fidelity, are established for general n-party asynchronous interactions.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; **Process calculi**; *Type theory*; *Type structures*; *Program analysis*; Operational semantics; • **Software and its engineering** → *Distributed programming languages*; *Concurrent programming structures;*

Additional Key Words and Phrases: Session types, the pi-calculus, projection, global types, global protocols, progress

ACM Reference Format:
Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty asynchronous session types. J. ACM 63, 1, Article 9 (March 2016), 67 pages.
DOI: http://dx.doi.org/10.1145/2827695

1. INTRODUCTION

Background. Communication is one of the central elements in software development, ranging from web services to parallel scientific computing to multicore programming. One of the main application areas of communication-based systems is business protocols. A *business protocol* is a series of structured and automated interactions among two or more business entities. During the 1990s, many attempts were made

A preliminary version of this article appeared in Proceedings of 35th annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2008).
Yoshida was supported by EPSRC EPSRC EP/K011715/1, EP/K034413/1, and EP/L00058X/1, EU project FP7-612985 UpScale and COST Action IC1201 BETTY. Carbone was supported by the Chords (granted by the Danish Agency for Science, Technology and Innovation) and COST Action IC1201 BETTY.
Authors' addresses: K. Honda, School of Electronic Engineering and Computer Science, Queen Mary, University of London, Mile End Road, London E1 4NS, United Kingdom; email: kohei.honda@eecs.qmul.ac.uk; N. Yoshida, Department of Computing, Imperial College London, South Kensington Campus, London SW7 2AZ, United Kingdom; email: n.yoshida@imperial.ac.uk; M. Carbone, IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark; email: carbonem@itu.dk.

DOI: http://dx.doi.org/10.1145/2827695

to describe and model business protocols in order to achieve, for example, automation, scalability, and correctness of protocols. As a result, several institutions started investing heavily in distributed computing technologies for the purpose of reducing the risk of centralized controls.

Against this background, the Web Services Choreography Description Language Working Group (WS-CDL WG) [WS-CDL 2003] was formed by W3C with the goal of defining a language standard for specifying web service business protocols by means of distributed interactions among peers (business entities). Recognizing the need for a foundational theory on which the design and infrastructure of the language were to be built, the working group took a strong interest in the π-calculus, leading to the involvement of Robin Milner and the authors as official invited experts in the standardization process. Although WS-CDL's design is informed by the π-calculus in both communication primitives and structuring constructs, WS-CDL differs from the π-calculus in that it describes message flows among multiple participants *globally*.

Engineers have always found it essential to use various notations for describing interaction patterns globally, such as the notations for cryptographic protocols, Message Sequence Charts [International Telecommunication Union 1996], and UML sequence diagrams. This is because a global description presents information on the behavior of systems that is not immediately available from the corresponding endpoint-based descriptions: How conversations among multiple participants evolve and interleave, what are the synchronization/communication points among participants, and how they together induce a desired global scenario. More formally, under a certain well-formedness condition, a global protocol automatically ensures that interactions satisfy the safety and deadlock-freedom properties.

WS-CDL follows these preceding global notations: An underlying intuition of its term *choreography* may be summarized as:

> *"Dancers dance following a global scenario (choreography) without a single point of control."*

Once specified, this scenario is to be executed by individual distributed processes without orchestrating nodes. A global description is meant to be executed by distributed interactions among end-point processes. Thus, each global description should be *projected* onto processes at each end-point whose mutual communications precisely realize the original global scenario. This translation from a global description to end-point processes is called *End-Point Projection* (EPP), in the terminology of the WS-CDL WG. The theory of multiparty session types introduced in this article was born from attempts to formalize the EPP in WS-CDL, applying the idea to *types*, in order to overcome a significant technical limitation of binary session types. This is one step beyond WS-CDL and gives closer links to recent tools for web services such as BPMN 2.0 Choreography [BPMNC 2012], as explained in the next paragraph.

Session Types. Over the past decade, *session types*, introduced in the 1990s [Honda 1993; Takeuchi et al. 1994; Honda et al. 1998], have provided a potential typed foundation for the design of communication-based systems. The main intuition behind session types is that a communication-centered application often exhibits a highly structured sequence of interactions involving, for example, sequencing and branching, which as a whole form a natural unit of conversation called a *session*. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis for validating programs through associated language primitives and a typing discipline.

As an example, consider a simple business protocol between a buyer (Buyer) and a seller (Seller) from Buyer's viewpoint: Buyer sends the title of a book (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, he then sends his address (a string) and Seller sends back the delivery date (a date); otherwise, he quits the

conversation. This can be precisely described by the following session type:

$$!\texttt{string};\ ?\texttt{int};\ \oplus\{\ \mathsf{ok} : !\texttt{string};\ ?\texttt{date};\ \mathsf{end},\quad \mathsf{quit} :\ \mathsf{end}\ \}. \tag{1}$$

This session type denotes patterns of communication operations (where ; denotes sequencing and $\oplus$ choice) describing Buyer's communication behavior in the business protocol. In particular, the term !`string` denotes an output of a value of type `string`, whereas ?`int` denotes an input of a value of type `int`. The choice $\oplus$ features the two options ok and quit. The term end represents the termination of the session. From Seller's viewpoint, the same session is described by the dual type

$$?\texttt{string};\ !\texttt{int};\ \&\{\ \mathsf{ok} : ?\texttt{string};\ !\texttt{date};\ \mathsf{end},\quad \mathsf{quit} :\ \mathsf{end}\ \}, \tag{2}$$

in which & means that a choice is offered.

Such an explicit representation of conversation structures allows us to deal with one of the most common bugs in communication-based programming, namely, the synchronization bug. A programmer expects that communicating programs should together realize a consistent conversation, but, unfortunately, they can easily fail to handle a specific incoming message or to send a message at the correct timing, with no way to detect such errors before runtime. An explicit specification as in Equation (1) guides us to principled programming of communication behavior and enables automatic protocol validation [WS-CDL 2003]. In addition, a clean separation between abstraction and implementation given by type-based abstraction and associated primitives leads to intelligible programs and flexible implementations [Hu et al. 2008, 2010]. Underlying these merits are the following central properties guaranteed by session types:

(1) Interactions within a session never incur a communication error (*communication safety*).
(2) Channels for a session are used linearly (linearity) and are deadlock-free in a single session (*progress*).
(3) The communication sequence in a session follows the scenario declared in the session type (*session fidelity, predictability*).

As a consequence of these properties, at each step in a session, a single input and a single output or a single selection and a single branching can take place via a session channel, moving to the next step.

Our previous research shows that the session-based programming framework is applicable to a wide range of calculi, programming languages, and computing environments, including calculi of mobile processes [Takeuchi et al. 1994; Gay and Hole 2005; Honda et al. 1998; Bonelli and Compagnoni 2007; Mezzina 2008; Yoshida and Vasconcelos 2007; Gay 2008; Dezani-Ciancaglini et al. 2007; Carbone et al. 2008], higher-order processes [Mostrous and Yoshida 2007, 2009], ambients [Garralda et al. 2006], multithreaded ML [Vasconcelos et al. 2006; Gay and Vasconcelos 2009], multicore programming [Yoshida et al. 2008], Haskell [Neubauer and Thiemann 2004a; Pucella and Tov 2008], F# [Bhargavan et al. 2009; Swamy et al. 2011], operating systems [Fähndrich et al. 2006], Java [Dezani-Ciancaglini et al. 2006; Coppo et al. 2007; Dezani-Ciancaglini et al. 2009; Hu et al. 2008; Gay et al. 2010; Hu et al. 2010; Ng et al. 2011], and Web Services [Carbone et al. 2006; 2007; WS-CDL 2003; Carbone et al. 2012; Sparkes 2006; Honda et al. 2007].

Multiparty Asynchronous Sessions. The foregoing studies on session types have focused on binary (two-party) sessions. Although many conversation patterns can be captured through a composition of binary sessions, there are cases where binary session types are not powerful enough for describing and validating interactions that involve more than two parties.

As an example, consider a simple refinement of the earlier Buyer-Seller protocol: Consider two buyers, Buyer1 and Buyer2, who wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller. It is extremely awkward (if logically possible) to decompose this scenario into three binary sessions, between Buyer1 and Seller, between Buyer2 and Seller, and between Buyer1 and Buyer2. Abstracting this protocol as three separate session types also means that our type abstraction loses essential sequencing information in this interaction scenario. For validating this conversation scenario as a whole, therefore, the conversation structure should be represented as a *single session*.

Many existing business protocols, including financial protocols, are written as a collaboration of several peers. Typical message-passing parallel algorithms also frequently demand distribution of a request to and collection of the results from many peers. All these use cases are most naturally abstracted as single multiparty sessions.

Furthermore, many of these applications are implemented with an *asynchronous transport,* where the senders send the messages without being blocked (but often preserving their order) to avoid the heavy overhead of synchronization. The widely used network transport, such as TCP, provides this mechanism through familiar APIs to alleviate the latency problem. Asynchronous message passing is also a standard assumption in financial messaging [AMQ 2015], parallel algorithms, and distributed objects and functions [Coppo et al. 2007; Hu et al. 2010, 2008; Ng et al. 2011; Neubauer and Thiemann 2004b; Fähndrich et al. 2006]. Thus, we ask:

Can we generalize the foregoing binary session types to multiparty asynchronous sessions, preserving clarity and their key formal properties?

Challenges of Multiparty Asynchronous Sessions. To answer this open question, we face two major technical difficulties. First, the simplicity and tractability of the theory of binary sessions come from a notion of *duality* in interactions found in Linear Logic [Girard 1987]. Consider the binary session type specified in Equation (1) for Buyer. Not only can Buyer's behavior be checked against the session type, but also the whole conversation structure is already represented in this single type since the interaction pattern of Seller is fully given as this type's dual (exchanging input and output and branching and selection in the original type). When composing two parties, we only have to check that they have mutually dual types. This framework based on duality is no longer effective in multiparty communication, where the whole conversation cannot be constructed from only single behavior. We need an effective means to abstract as a type a global scenario that a programmer wishes to realize through interacting programs (hence against which she would wish to check their correctness) and establish an effective method to ensure composability.

Second, linearity analysis of channels, which is the key for ensuring safety and progress, becomes highly involved under a combination of asynchrony and multiparty communication since a conflict of actions can arise more easily. A linearity property holds if a communication via the same channel of a global type does not break the order of messages as it is specified in the global description. This demands a precise causal analysis for correct sequencing of interactions distributed among multipeers.

This Work. This article presents a generalization of binary session types to multiparty sessions for the π-calculus. We propose *three* major technical contributions to overcome the aforementioned challenges:

(1) A new notion of types that can directly abstract intended conversation structure among n-parties as *global scenarios*, retaining an intuitive type syntax.

(2) Consistency criteria for a conversation structure with respect to a protocol specification given as a causality analysis of actions in global types, modularly articulating different kinds of dependency.
(3) A type discipline for individual processes (programs) that uses global types through their *projections* onto individual endpoint participants: The resulting endpoint types are directly associated with individual processes for type-checking.

The idea of type abstraction based on a global view (Point 1) comes from an *abstract version of "choreography"* developed in a W3C web services working group [Carbone et al. 2006; WS-CDL 2003]. Causality structures in asynchronous interactions are precisely and modularly captured in the abstract setting of global types, offering a foundation for the type discipline (Point 2). Through the use of global types, we propose a new effective method for designing, type-checking, and developing programs based on multiparty sessions (Point 3).

Let us illustrate Point 3 in detail. First, we design and agree upon a global type G as an intended conversation scenario. A team of programmers then develops code, one for each participant, incrementally validating its conformance to (the projections of) G. When programs are executed, their interactions automatically follow the stipulated scenario. The projection can also be used as a hint for modeling, designing, and debugging local behaviors of participants. After the development, a global type will serve as a basis of monitoring, maintenance, and upgrade. For materializing this design framework, the proposed framework presents a type discipline that can validate whether a program is typable or not, given G (as a shared agreement) and an individual program (as its endpoint realizer). The resulting type discipline guarantees all the original key properties of binary session types, such as communication error freedom, progress, and session fidelity in a general n-party session, underpinning its practical use. For further discussions on this development framework and its applications developed in industry and academia, see Sections 4.1, 6.1, and 7.

This article is a full version of Honda et al. [2008a], with detailed definitions and full proofs. It is also expanded with more examples and comparisons with recent related work. Section 2 gives the syntax and semantics of the calculus and motivates the key ideas through business and streaming protocol examples and a use case from OOI [2015]. Section 3 explains the global types. Section 4 describes the typing system. Section 5 establishes the main results. Section 6 discusses extensions and related works. Section 7 concludes with future issues and a summary of applications, software, and languages developed with industry collaborators based on the multiparty session type theory. The appendix contains the proofs of the propositions, lemmas, and theorems stated in the main sections.

2. MULTIPARTY ASYNCHRONOUS SESSIONS

2.1. Syntax for Multiparty Sessions

Several versions of π-calculi with session types have been proposed in the literature. A detailed survey can be found in Dezani-Ciancaglini and de' Liguoro [2010]. In this work, we use a simple extension of the original language for binary sessions [Honda et al. 1998; Takeuchi et al. 1994] to multiparty sessions.

Informally, a *session* is a series of interactions that serve as a unit of conversation. A session is established among multiple parties via a *shared name*, which represents a public interaction point. Then, fresh *session channels* are generated and shared among all participants who can use them for communicating with each other.

In the remainder, we use the following base sets:

—*shared names* or *names*, ranged over by $a, b, x, y, z, \ldots$;
—*session channels* or *channels*, ranged over by $s, t, \ldots$;

$$
\begin{array}{rll}
P & ::= \overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P & \text{multicast session request} \\
& \mid a_{[\mathtt{p}]}(\tilde{s}).P & \text{session acceptance} \\
& \mid s!\langle \tilde{e} \rangle; P & \text{value sending} \\
& \mid s?(\tilde{x}); P & \text{value reception} \\
& \mid s!\langle\!\langle \tilde{s} \rangle\!\rangle; P & \text{session delegation} \\
& \mid s?(\!(\tilde{s})\!); P & \text{session reception} \\
& \mid s \lhd l; P & \text{label selection} \\
& \mid s \rhd \{l_i \colon P_i\}_{i \in I} & \text{label branching} \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{conditional branch} \\
& \mid P \mid Q & \text{parallel composition} \\
& \mid \mathbf{0} & \text{inaction} \\
& \mid (\boldsymbol{\nu}\, n)P & \text{hiding} \\
& \mid \mathsf{def}\ D\ \mathsf{in}\ P & \text{recursion} \\
& \mid X\langle \tilde{e}\tilde{s} \rangle & \text{process call} \\
& \mid s :: \tilde{h} & \text{message queue} \\
e & ::= v \mid e\ \mathsf{and}\ e' \mid \mathsf{not}\ e \quad \ldots & \text{expressions} \\
v & ::= a \mid \mathsf{true} \mid \mathsf{false} & \text{values} \\
h & ::= l \mid \tilde{v} \mid \tilde{s} & \text{messages-in-transit} \\
D & ::= \{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i \in I} & \text{declaration for recursion}
\end{array}
$$

Fig. 1. Syntax.

—*labels*, ranged over by $l, l', \ldots$; and
—*process variables*, ranged over by $X, Y, \ldots$.

We use n for either a single shared name or a vector of session channels. The symbol $\tilde{s}$ denotes the vector of session channels $s_1, \ldots, s_k$ for some k. Similarly, for other names, channels and variables. Then, *processes*, ranged over by $P, Q \ldots$, and *expressions*, ranged over by $e, e', \ldots$, are given by the grammar in Figure 1.

Except for the first two primitives for session initiation and the final message queue, all constructs are from the binary session calculi [Honda et al. 1998]. Session initiation is introduced to establish a session between multiple processes, whereas message queues are added to model asynchronous session communication, as explained later.

Among the primitives for session initiation, the prefix process $\overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P$ initiates a new session through a shared interaction point a by distributing a vector of freshly generated session channels $\tilde{s}$ to the remaining $\mathtt{n} - 1$ participants, each of shape $a_{[\mathtt{p}]}(\tilde{s}).Q_{\mathtt{p}}$ for $2 \leq \mathtt{p} \leq \mathtt{n}$. All receive $\tilde{s}$, over which the actual session communications can now take place among the $\mathtt{n}$ parties. $\mathtt{p}, \mathtt{q}, \ldots$ range over natural numbers called *participants* of a session. As we formalize later through operational semantics, these primitives offer a distilled syntactic presentation of "sharing of a fresh context for a new session" among multiple parties.

Session communications are performed using the next three pairs of primitives: sending and receiving, session delegation, and reception (the former delegates to the latter the capability to participate in a session by passing channels associated with the session), and selection and branching (the former chooses one of the branches offered by the latter). Branching and selection constructs correspond to external and internal choices.

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

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

$$(\boldsymbol{\nu}\, nn')P \equiv (\boldsymbol{\nu}\, n'n)P \qquad (\boldsymbol{\nu}\, n)\mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}\, s_1 ... s_n)(s_1 :: \emptyset \mid \cdots s_n :: \emptyset) \equiv \mathbf{0} \qquad \mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0}$$

$$\mathsf{def}\ D\ \mathsf{in}\ (\boldsymbol{\nu}\, n)P \equiv (\boldsymbol{\nu}\, n)\mathsf{def}\ D\ \mathsf{in}\ P \quad \text{if } n \notin \mathsf{fn}(D)$$

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

$$\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv \mathsf{def}\ D \cup D'\ \mathsf{in}\ P \quad \text{if } \mathsf{dpv}(D) \cap \mathsf{dpv}(D') = \emptyset$$

Fig. 2. Structural congruence.

The next three (conditional, parallel, and inaction) are standard. $(\boldsymbol{\nu} a)P$ makes a local to P whereas $(\boldsymbol{\nu} \tilde{s})P$ makes $\tilde{s}$ local to P. The recursion and process call primitives realize recursive behavior. $s :: \tilde{h}$ is a *message queue* representing ordered messages in transit $\tilde{h}$ with destination s (which may be considered a network pipe in a TCP-like transport). $(\boldsymbol{\nu} \tilde{s})P$ and $s :: \tilde{h}$ only appear at runtime. We often omit trailing $\mathbf{0}$ and write $s!$ and $s?.P$, omitting the arguments if unnecessary. Informally speaking, if we map our syntax to TCP, each queue corresponds to the TCP FIFO channel. Then, a shared name correspond to a pair of an IP and a port name to initiate the session, while each session name is mapped to a pair of freshly generated IP and a port name that connects to the pair of the other side.

Binders are $\tilde{s}$ in $\overline{a}[2..\mathtt{n}]\,(\tilde{s}).P$, $a[\mathtt{p}]\,(\tilde{s}).P$ and $s?(\!(\tilde{s}\,)\!); P$, $\tilde{x}$ in $s?(\tilde{x}); P$, $\tilde{x}\tilde{s}$ in $X(\tilde{x}\tilde{s}) = P$, n in $(\boldsymbol{\nu}\, n)P$ and process variables in $\mathsf{def}\ D\ \mathsf{in}\ P$. The notions of bound and free identifiers, channels, alpha equivalence $\equiv_\alpha$, and substitution are standard. The functions $\mathsf{fpv}(P)$ and $\mathsf{fn}(P)$ denote the sets of *free process variables* and *free identifiers*, respectively. Function $\mathsf{dpv}(\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I})$ denotes the set of *process variables* $\{X_i\}_{i\in I}$ introduced in $\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$. The notation $\Pi_i P_i$ denotes the parallel composition of zero or more processes P_i.

Structural congruence $\equiv$ over processes is the smallest congruence relation on processes that includes the equations given in Figure 2. These are standard except that we allow a vector of session channels in hiding, which is convenient for some proofs in the typing system (no substantial difference arises regarding the nature of the calculus by hiding channels one by one).

Definition 2.1 (*Program Phrase and Program*). A process P is a *program phrase* if P has no queues and no ν-bound session channels (up to $\equiv$). P is a *program* (up to $\equiv$) if P is a program phrase in which no free session channels and process variables occur.

In the examples in Section 2.3, processes such as Buyer1, Buyer2, Seller, Kernel, DataProducer, and Consumer are programs; hence, they are also program phrases.

2.2. Operational Semantics

The operational semantics is given by the *reduction relation*, denoted $P \to Q$, which is the smallest relation on processes generated by the rules in Figure 3. In the figure, $e \downarrow v$ says that expression e evaluates to values v, but we leave its formal definition unspecified. We now explain each rule.

Rule [Link] describes a session initiation among $\mathtt{n}$-parties through n-party synchronization, generating m fresh session channels and the associated m empty queues ($\emptyset$ denotes the empty string). Each fresh channel is given a new empty queue. As a result, $\mathtt{n}$ participants now share the newly generated m channels, hence their queues. Note that the number of participants ($\mathtt{n}$) can be different from that of session channels (m),

$$\overline{a}_{[2..n]}(\tilde{s}).P_1 \mid a_{[2]}(\tilde{s}).P_2 \mid \cdots \mid a_{[n]}(\tilde{s}).P_n \;\rightarrow\; (\boldsymbol{\nu}\,\tilde{s})(P_1 \mid P_2 \mid ... \mid P_n \mid s_1 :: \emptyset \mid \cdots \mid s_m :: \emptyset) \qquad \text{[Link]}$$

$$s!\langle \tilde{e} \rangle; P \mid s :: \tilde{h} \;\rightarrow\; P \mid s :: \tilde{h} \cdot \tilde{v} \quad (\tilde{e} \downarrow \tilde{v}) \qquad \text{[Send]}$$

$$s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \mid s :: \tilde{h} \;\rightarrow\; P \mid s :: \tilde{h} \cdot \tilde{t} \qquad \text{[Deleg]}$$

$$s \lhd l; P \mid s :: \tilde{h} \;\rightarrow\; P \mid s :: \tilde{h} \cdot l \qquad \text{[Label]}$$

$$s?(\tilde{x}); P \mid s :: \tilde{v} \cdot \tilde{h} \;\rightarrow\; P[\tilde{v}/\tilde{x}] \mid s :: \tilde{h} \qquad \text{[Recv]}$$

$$s?((\tilde{t})); P \mid s :: \tilde{t} \cdot \tilde{h} \;\rightarrow\; P \mid s :: \tilde{h} \qquad \text{[SRec]}$$

$$s \rhd \{l_i \colon P_i\}_{i \in I} \mid s :: l_j \cdot \tilde{h} \;\rightarrow\; P_j \mid s :: \tilde{h} \qquad (j \in I) \qquad \text{[Branch]}$$

$$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \;\rightarrow\; P \qquad (e \downarrow \mathsf{true}) \qquad \text{[IfT]}$$

$$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \;\rightarrow\; Q \qquad (e \downarrow \mathsf{false}) \qquad \text{[IfF]}$$

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

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

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

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

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

Fig. 3. Reduction.

giving flexibility and maintaining linearity in channel usage. The use of the *n*-party synchronization in this rule captures, albeit abstractly, an *n*-party handshake that would be necessary for establishing an *n*-party link in real-world protocols. For example, we can create an arbitrary number of queues that can be dequeued and enqueued by all parties in that session.

Rules [Send], [Deleg], and [Label] respectively enqueue values, channels, and a label at the tail of the queue for s. In rule [Send], $\tilde{e} \downarrow \tilde{v}$ evaluates each expression e_i to a value v_i and its definition is left unspecified. Symmetrically, rules [Recv], [SRec], and [Branch] dequeue, from the head of the queue, values, session channels, and a label, respectively. Rules [Recv] further instantiates the received value in the continuation P, while rule [Branch] selects, from its continuation, the branch corresponding to the received label. The reduction rules [Deleg] and [SRec] are often called (session) *delegation* or *higher order session passing*.

In these communication rules, sending and receiving are mediated by a queue: Only when a message sent by (say) Alice is received by (say) Bob through a queue, can we say that an interaction between Alice and Bob has taken place. Since [Link] generates a queue for each channel, these rules entail that:

(1) A sending action is never blocked (communication asynchrony); and that
(2) two messages from the same sender to the same channel arrive in the sending order (message order preservation).

As we discussed in Section 1, these are among the main features of the well-known transport mechanisms TCP, and the message queue is introduced for modeling these transports.

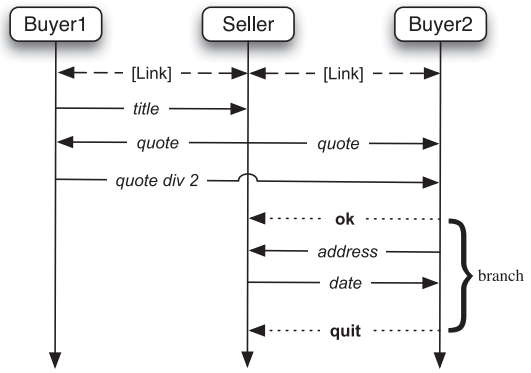
All other rules are standard: For reference, we briefly describe them. The two rules for conditional, [IfT] and [IfF], reduce to one of the branches, depending on the evaluation of the guard. Rule [Def] performs unfolding of recursion. Rules [Scop], [Par], and [Defin] close the reduction relation under hiding, parallel composition, and definition, respectively. Finally, rule [Str] says that the reduction relation is defined over processes up to $\equiv$.

Remark 2.2. The rule for delegation [SRec], originally introduced in Honda et al. [1998] for the π-calculus with sessions, uses the same session name $\tilde{t}$ without substitution for a simpler presentation. However, having [SRec] with substitution as in [Recv] breaks the subject reduction theorem and requires either two endpoint channels or bidirectional buffers [Gay and Hole 2005; Gay and Vasconcelos 2009; Yoshida and Vasconcelos 2007]. The reader can find a more detained explanation in Yoshida and Vasconcelos [2007]. Roughly speaking, a substitution creates a self-delegation where the receiver gets his own session by which the shape of the session type is changed and the subject reduction is broken. Hence, we require additional queues and restrictions on the form of the communication. The technical development of this work does not depend on this choice (see also Section 6.2).

2.3. Examples

We now report two examples that have been used for discussion within the W3C WS-CDL working group [WS-CDL 2003]. Further large examples and applications of multiparty session types are listed in Honda et al. [2008b].

Example 2.3 (*Two-buyer Protocol*). We describe the two-buyer protocol from the Introduction first by a sequence diagram, then by processes.



First Buyer1 sends a book title to Seller; then, Seller sends back a quote to Buyer1 and Buyer2; Buyer1 tells Buyer2 how much she is willing to contribute; and, finally, Buyer2 notifies Seller whether it accepts the quote or not. We can describe the behavior of Buyer1 with the following process:

$$\begin{aligned}\text{Buyer1} \stackrel{\text{def}}{=}\ & \overline{a}_{[2,3]}(b_1, b_2, b_2', s).\ s!\langle\text{“War and Peace”}\rangle;\\ & b_1?(quote);\ b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1\end{aligned}$$

Channel b_1 is for Buyer1 to receive messages: b_2 and b_2' for Buyer2 and s for Seller (we discuss soon why Buyer2 needs two receiving channels). Buyer1 is willing to contribute to half of the quote. In P_1, Buyer1 may perform the remaining transactions with Seller

and Buyer2. The remaining participants follow.

$$\begin{aligned}\text{Buyer2} \stackrel{\text{def}}{=}\ & a_{[2]}(b_1, b_2, b_2', s).\ b_2?(quote);\ b_2'?(contrib);\\ & \text{if } (quote - contrib \leq 99)\\ & \quad \text{then}\quad s \lhd \mathsf{ok}; s!\langle address\rangle; b_2?(x); P_2\\ & \quad \text{else}\quad s \lhd \mathsf{quit}; \mathbf{0}\\ \text{Seller} \stackrel{\text{def}}{=}\ & a_{[3]}(b_1, b_2, b_2', s).\ s?(title);\quad b_1, b_2!\langle quote\rangle;\\ & s \rhd \{\mathsf{ok} : s?(x); b_2!\langle date\rangle; Q,\quad \mathsf{quit} : \mathbf{0}\}\end{aligned}$$

Here, $s_1..s_m!\langle v\rangle; P$ stands for $s_1!\langle v\rangle; ..s_m!\langle v\rangle; P$, assuming $s_1..s_m$ are pairwise distinct.[1] We now explain why Buyer2 needs to use two input channels, b_2 and b_2'. The first input (for *quote*) is from Seller, whereas the second one (for *contrib*) is from Buyer1. Hence, there is no guarantee that they arrive in a fixed order, as can be easily seen by analyzing reduction paths (this is Lamport's principle [Lamport 1978]). Thus, if we were to use b_2 for both actions, the two messages can be confused, losing linear usage of a channel. The problem becomes visible after the fifth step of the following reduction. If b_2 and b_2' were the same then the contribution of the Buyer1 could be queued before the price of the book and therefore received before at Buyer2. In Section 4, we use our type discipline to detect this kind of error.

We now show an example of reductions. Let us define:

$$\begin{aligned}P \triangleq\ & \text{if } (quote - contrib \leq 99)\\ & \text{then}\ \ s \lhd \mathsf{ok}; s!\langle address\rangle; b_2?(x); P_2\\ & \text{else}\ \ s \lhd \mathsf{quit}; \mathbf{0}\end{aligned} \qquad S \triangleq s \rhd \{\mathsf{ok} : s?(x); b_2!\langle date\rangle; Q,\ \ \mathsf{quit} : \mathbf{0}\}$$

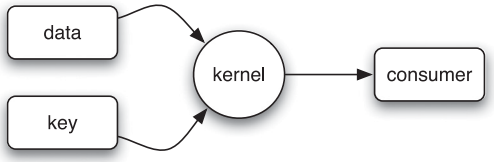
Below, a tag denotes the name of the rule from Figure 3 we apply. For simplicity, we omit [Par] and [Scop] after the second reduction.

$$\begin{aligned}& \qquad\qquad \text{Buyer1} \mid \text{Buyer2} \mid \text{Seller}\\ \rightarrow^{[\text{Link}]}\ & (\nu\, b_1, b_2, b_2', s)(\ s!\langle\text{“War and Peace”}\rangle; b_1?(quote); b_2'!\langle quote \text{ div } 2\rangle; P_1\\ & \qquad\qquad \mid b_2?(quote); b_2'?(contrib); P\\ & \qquad\qquad \mid s?(title); b_1, b_2!\langle quote\rangle; S\\ & \qquad\qquad \mid b_1 :: \emptyset \mid b_2 :: \emptyset \mid b_2' :: \emptyset \mid s :: \emptyset)\\ \rightarrow^{[\text{Send}],[\text{Par}],[\text{Scop}]}\ & (\nu\, b_1, b_2, b_2', s)(\ b_2'!\langle quote \text{ div } 2\rangle; P_1\\ & \qquad\qquad \mid b_2?(quote); b_2'?(contrib); P\\ & \qquad\qquad \mid s?(title); b_1, b_2!\langle quote\rangle; S\\ & \qquad\qquad \mid b_1 :: \emptyset \mid b_2 :: \emptyset \mid b_2' :: \emptyset \mid s :: \text{“War and Peace”})\\ \rightarrow^{[\text{Rcv}]}\ & (\nu\, b_1, b_2, b_2', s)(\ b_1?(quote); b_2'!\langle quote \text{ div } 2\rangle; P_1\\ & \qquad\qquad \mid b_2?(quote); b_2'?(contrib); P\\ & \qquad\qquad \mid b_1, b_2!\langle quote\rangle; S\\ & \qquad\qquad \mid b_1 :: \emptyset \mid b_2 :: \emptyset \mid b_2' :: \emptyset \mid s :: \emptyset)\end{aligned}$$

[1]Due to asynchrony, there is in effect no order among the sending actions at $s_1..s_m$.

$$\begin{aligned}
&\rightarrow^{[\text{Send}]} (\boldsymbol{\nu}\, b_1, b_2, b_2', s)(\ b_1?(quote); b_2'!\langle quote \text{ div } 2\rangle; P_1 \\
&\qquad\qquad\qquad\qquad\quad \mid b_2?(quote); b_2'?(contrib); P \\
&\qquad\qquad\qquad\qquad\quad \mid b_2!\langle quote\rangle; S \\
&\qquad\qquad\qquad\qquad\quad \mid b_1 :: quote \mid b_2 :: \emptyset \mid b_2' :: \emptyset \mid s :: \emptyset) \\
&\rightarrow^{[\text{Rcv}]} (\boldsymbol{\nu}\, b_1, b_2, b_2', s)(\ b_2'!\langle quote \text{ div } 2\rangle; P_1 \\
&\qquad\qquad\qquad\qquad\quad \mid b_2?(quote); b_2'?(contrib); P \\
&\qquad\qquad\qquad\qquad\quad \mid b_2!\langle quote\rangle; S \\
&\qquad\qquad\qquad\qquad\quad \mid b_1 :: \emptyset \mid b_2 :: \emptyset \mid b_2' :: \emptyset \mid s :: \emptyset) \\
&\ldots
\end{aligned}$$

Example 2.4 (*Streaming Protocol*). We next consider a simple protocol for the standard stream cipher [Schneier 1993].



Data Producer and Key Producer continuously send a data stream and a key stream, respectively, to Kernel. Kernel calculates their XOR and sends the result to Consumer.

Assuming streams are sent block by block (say, as large arrays), we can realize this protocol as communicating processes. We focus only on communication behavior. The kernel initiates a session:

$$\begin{aligned}
\text{Kernel} \stackrel{\text{def}}{=}\ &\text{def } \mathrm{K}(d, k, c) = d?(x); k?(y); c!\langle x \text{ xor } y\rangle; \mathrm{K}\langle d, k, c\rangle \\
&\text{in } \overline{a}_{[2,3,4]}(d, k, c).\mathrm{K}\langle d, k, c\rangle
\end{aligned}$$

The channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively, while c is used for Consumer to receive the encrypted data from Kernel. Data Producer and Consumer can be given as:[2]

$$\text{DataProducer} \stackrel{\text{def}}{=} \text{def } \mathrm{P}(d, k, c) = d!\langle data\rangle; \mathrm{P}\langle d, k, c\rangle \text{ in } a_{[2]}(d, k, c).\mathrm{P}\langle d, k, c\rangle$$

$$\text{Consumer} \stackrel{\text{def}}{=} \text{def } \mathrm{C}(d, k, c) = c?(data); \mathrm{C}\langle d, k, c\rangle \text{ in } a_{[3]}(d, k, c).\mathrm{C}\langle d, k, c\rangle$$

Key Producer is identical to Data Producer except it outputs at k instead of d. When three processes are composed, we can verify that messages are always consumed in the order they are produced, an essential requirement for correctness of the protocol (although processes repeatedly send and receive data using the same channel). This is because each channel is used by exactly one sender. We show how this argument can be cleanly represented and validated through session types in the next two sections.

[2]For simplicity, our description lets both Data Producer and Consumer repeatedly send the same data: Practically, this is not the case, but this simplified form is enough for our current concern (i.e., validation of communication behavior).

$$
\begin{array}{lllll}
\text{Global} & G & ::= & \mathtt{p} \to \mathtt{p}' : k\,\langle U \rangle.\, G' & \text{values} \\
 & & \mid & \mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j \in J} & \text{branching} \\
 & & \mid & G \mid G' & \text{parallel} \\
 & & \mid & \mu \mathbf{t}.G & \text{recursive} \\
 & & \mid & \mathbf{t} & \text{variable} \\
 & & \mid & \mathsf{end} & \text{end} \\
\text{Value} & U & ::= & \tilde{S} & \text{sorts} \\
 & & \mid & T@\mathtt{p} & \text{located session} \\
\text{Sort} & S & ::= & \mathsf{bool} \mid \mathsf{nat} \mid \ldots \mid \langle G \rangle &
\end{array}
$$

Fig. 4. Syntax of global types.

3. GLOBAL TYPES AND CAUSAL ANALYSIS

Developing programs for multiparty sessions demands a clear formal design since we need to program global interactions where multiple participants communicate and synchronize with each other. Programming individual participants without such a design and hoping they somehow realize a meaningful and error-free conversation is hardly practical, especially when the implementation is done by a team of several programmers. In binary session types, the type for an endpoint also served as the description of the whole conversation between two parties, but this is no longer possible for multiparty sessions. This is why we require the type abstraction to describe global conversation scenarios of multiparty sessions: The global types introduced in this section extend binary session types to be able to directly express dependencies between communications among multiple peers.

3.1. Syntax of Global Types

A global type abstracts global multiparty conversations as a type signature. It takes a similar form to cryptography protocols where a message exchange from participant $\mathtt{p}$ to participant $\mathtt{p}'$ is specified as $\mathtt{p} \to \mathtt{p}'$. For example, the protocol "`Alice` sends a message with type nat to `Bob` via channel k, then terminates the interaction" is simply described as $\mathtt{Alice} \to \mathtt{Bob} : k\,\langle \mathsf{nat} \rangle.\ \mathsf{end}$. *Unlike the standard types of process calculi, the syntax no longer describes the input and output types separately: The information exchange between two parties is directly abstracted as one interaction.*

The full syntax of *global session types*, or *global types*, denoted by $G, G', \ldots$, is given in Figure 4. In a global type, we refer to session channels with a number, denoted by $k, k', \ldots$, which corresponds to the index of a vector of session channels: If we want to refer to the k-th session channel s_k of $s_1..s_n$ (such a vector is created by a session initiation), we write k in the global type. By writing number k (like de Bruijn notation), instead of channel s_k, we avoid including binding in the syntax of global types. We call k a *session channel index*.

$U, U', \ldots$ range over *value types*, denoting types for message values. Each value type is either a vector of *sorts* or a *located type*. Sorts, written $S, S', \ldots$, are types for shared names, where $\langle G \rangle$ means communicating a shared name typed by $\langle G \rangle$. A located type $T@\mathtt{p}$ denotes the communication (delegation) of a session channel of type T (called *endpoint type*) with role $\mathtt{p}$. Both of these types (T and S) are discussed in detail in Section 4.2. For understanding this section, it suffices to assume U as a single base type (i.e., only nat or bool). We often write $\mathtt{p} \to \mathtt{p}' : k.\ G'$ for $\mathtt{p} \to \mathtt{p}' : k\,\langle \rangle.\ G'$ (i.e., U is empty).

We now give a detailed description of each term in Figure 4.

Type $\mathtt{p} \to \mathtt{p}' : k\,\langle U \rangle.\ G'$ says that an interaction between two participants over a session channel with index k must take place. In an implementation of such a behavior,

given a vector of session channels $\tilde{s}$, participant $\mathtt{p}$ would send some message of type U to participant $\mathtt{p}'$ over s_k and then the session would continue according to G'. The type U is called *carried type*. Note that the operator "." captures sequentiality. As an example, the global type

$$1 \to 3 : k\,\langle \mathsf{int} \rangle.\ 3 \to 2 : k'\,\langle \mathsf{bool} \rangle.\ \mathsf{end} \tag{3}$$

describes a protocol where, given a vector $\tilde{s}$, participant 1 sends an integer to participant 3 over session channel s_k, and then 3 sends a boolean to participant 2 over $s_{k'}$. This protocol can be written in the model introduced in the previous section as follows:

$$\underbrace{s_k!\langle 5\rangle;\mathbf{0}}_{1} \mid \underbrace{s_{k'}?(y);\mathbf{0}}_{2} \mid \underbrace{s_k?(x);s_{k'}!\langle x=5\rangle;\mathbf{0}}_{3}$$

According to the semantics given in Figure 3, the preceding processes (where, for the sake of clarity, we have labeled each process with a number corresponding to a participant) will execute according to the specification given by the global type in Equation (3). Obviously, the same channel can be used several times, as in the following type:

$$1 \to 3 : k\,\langle \mathsf{int} \rangle.\ 3 \to 2 : k'\,\langle \mathsf{bool} \rangle.\ 2 \to 1 : k\,\langle \mathsf{bool} \rangle. \tag{4}$$

A possible implementation respecting such a protocol is:

$$\underbrace{s_k!\langle 5\rangle;s_k?(z);\mathbf{0}}_{1} \mid \underbrace{s_{k'}?(y);s_k!\langle \mathsf{true}\rangle;\mathbf{0}}_{2} \mid \underbrace{s_k?(x);s_{k'}!\langle x=5\rangle;\mathbf{0}}_{3}$$

On the other hand, the following process would *not* satisfy the specification in Equation (4):

$$\underbrace{s_k!\langle 5\rangle;s_k?(z);\mathbf{0}}_{1} \mid \underbrace{s_k!\langle \mathsf{true}\rangle;s_{k'}?(y);\mathbf{0}}_{2} \mid \underbrace{s_k?(x);s_{k'}!\langle x=5\rangle;\mathbf{0}}_{3}$$

Unfortunately, due to asynchrony, it is possible that participant 3 receives a boolean while participant 1 receives, later on, an integer, causing a runtime error.

Type $\mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J}$ denotes branching of a session. Intuitively, participant $\mathtt{p}$ must send one of the labels in $\{l_j \mid j \in I\}$ on channel s_k to participant $\mathtt{p}'$. When l_i is sent, interactions described in G_i will take place. For example, the global type

$$1 \to 3 : k\,\{\quad \mathsf{five} : 3 \to 2 : k'\,\langle \mathsf{bool} \rangle.\ \mathsf{end},\quad \mathsf{notfive} : 3 \to 2 : k'\,\langle \mathsf{bool} \rangle.\ \mathsf{end}\quad \}$$

could be implemented by the process

$$\underbrace{\mathsf{if}\ e\ \mathsf{then}\ s_k \lhd \mathsf{five}\ \mathsf{else}\ s_k \lhd \mathsf{notfive}}_{1} \mid \underbrace{s_{k'}?(y)}_{2} \mid \underbrace{s_{k'} \rhd \left\{ \begin{array}{r} \mathsf{five} : s_{k'}!\langle \mathsf{true}\rangle, \\ \mathsf{notfive} : s_{k'}!\langle \mathsf{false}\rangle \end{array} \right\}}_{3}$$

Type $G \mid G'$ specifies the concurrent execution of the interactions in G and G'.

Type $\mu\mathbf{t}.G$ is a recursive type for recurring conversation structures, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way (i.e., type variables only appear under the prefixes [hence contractive]). We take an *equi-recursive* view, not distinguishing between $\mu\mathbf{t}.G$ and its unfolding $G[\mu\mathbf{t}.G/\mathbf{t}]$ [Pierce 2002]. We assume that $\langle G \rangle$ in the grammar of sorts is closed (i.e., without type variables).[3]

Type end represents the termination of the session and is often omitted. We identify "$G \mid \mathsf{end}$" and "$\mathsf{end} \mid G$" with G.

[3]In the presence of the standard recursive sorts [Honda et al. 1998], which we omit for simpler presentation, we allow sort variables to occur in $\langle G \rangle$.

We conclude this subsection by giving the following definition:

Definition 3.1 (*Action*). We say that $\mathtt{p} \to \mathtt{p}' : k$ in $\mathtt{p} \to \mathtt{p}' : k\,\langle U\rangle.\, G'$ or $\mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J}$ is an *action from* $\mathtt{p}$ *to* $\mathtt{p}'$ *at* k.

3.2. Operational Semantics for Global Types

This subsection defines semantics of global types, introducing the Labeled Transition Relation (LTS). The LTS is useful not only to give a clear justification for causal dependencies of global types defined in the next subsection, but also to prove the main theorems for the typing system later.

Definition 3.2 (*Global Type Labeled Transition Relation*). The syntax of labels ($\ell, \ell', \ldots$) of global types is defined as follows:

$$\ell \;::=\; \mathtt{p} \to \mathtt{p}' : k\,\langle U\rangle \;\;\mid\;\; \mathtt{p} \to \mathtt{p}' : k\,\langle l\rangle$$

A label ℓ denotes a communication over a channel k of some type U or label l. Then the transition relation $G \xrightarrow{\ell} G'$ is defined by the following rules:

$$[\mathrm{GR1}]\quad \mathtt{p} \to \mathtt{q}\colon k\,\langle U\rangle.\, G \xrightarrow{\mathtt{p}\to\mathtt{q}\colon k\,\langle U\rangle} G \qquad [\mathrm{GR2}]\quad \mathtt{p} \to \mathtt{q}\colon k\,\{l_i : G_i\}_{i\in I} \xrightarrow{\mathtt{p}\to\mathtt{q}\colon k\,\langle l_j\rangle} G_j$$

$$[\mathrm{GR3}]\ \frac{G_1 \xrightarrow{\ell} G_2 \qquad \mathtt{q} \notin \ell}{\mathtt{p} \to \mathtt{q}\colon k\,\langle U\rangle.\, G_1 \xrightarrow{\ell} \mathtt{p} \to \mathtt{q}\colon k\,\langle U\rangle.\, G_2} \qquad [\mathrm{GR4}]\ \frac{\forall i \in I.\, G_i \xrightarrow{\ell} G_i' \quad \mathtt{q} \notin \ell}{\mathtt{p} \to \mathtt{q}\colon k\,\{l_i : G_i\}_{i\in I} \xrightarrow{\ell} \mathtt{p} \to \mathtt{q}\colon k\,\{l_i : G_i'\}_{i\in I}}$$

$$[\mathrm{GR5}]\ \frac{G_1 \xrightarrow{\ell} G_1'}{G_1 \mid G_2 \xrightarrow{\ell} G_1' \mid G_2} \qquad [\mathrm{GR6}]\ \frac{G_2 \xrightarrow{\ell} G_2'}{G_1 \mid G_2 \xrightarrow{\ell} G_1 \mid G_2'}$$

The rules allow us to permute the order of two actions that are causally unrelated. This is defined by the condition $\mathtt{q} \notin \ell$ in [GR3,4]. Note that in [GR4], we require that each branch must be able to perform action ℓ.

As a simple example, consider $G = 1 \to 2 : k\,\langle \mathsf{int}\rangle.\ 3 \to 4 : k'\,\langle \mathsf{bool}\rangle.\,\mathsf{end}$ and let $\ell_1 = 1 \to 2 : k\,\langle \mathsf{int}\rangle$ and $\ell_2 = 2 \to 3 : k'\,\langle \mathsf{bool}\rangle$. Since the participants are pairwise distinct, we can perform the second action first. Hence, using [GR1] and [GR3] above, we have two possible transition relations from as follows:

$$G \xrightarrow{\ell_1} 3 \to 4 : k'\,\langle \mathsf{bool}\rangle.\,\mathsf{end} \xrightarrow{\ell_2} \mathsf{end} \quad \text{and} \quad G \xrightarrow{\ell_2} 1 \to 2 : k\,\langle \mathsf{int}\rangle.\,\mathsf{end} \xrightarrow{\ell_1} \mathsf{end}$$

Another interesting example is: $1 \to 2 : k\,\langle \mathsf{int}\rangle.\ 3 \to 1 : k'\,\langle \mathsf{bool}\rangle.\,\mathsf{end}$. This global type means that participant 1 is allowed to *receive* the message from participant 3 before the message from 1 is received by 2 since they are delivered to the two different channels (i.e., queues). Thus, with $\ell_3 = 2 \to 1 : k'\,\langle \mathsf{bool}\rangle$, we have:

$$G' \xrightarrow{\ell_1} 3 \to 1 : k'\,\langle \mathsf{bool}\rangle.\,\mathsf{end} \xrightarrow{\ell_3} \mathsf{end} \text{ and } G' \xrightarrow{\ell_3} 1 \to 2 : k\,\langle \mathsf{int}\rangle.\,\mathsf{end} \xrightarrow{\ell_1} \mathsf{end}$$

On the other hand, $G'' = 3 \to 1 : k'\,\langle \mathsf{bool}\rangle.\ 1 \to 2 : k\,\langle \mathsf{int}\rangle.\,\mathsf{end}$ has only one possible transition since the two inputs at the receiver $\mathtt{q}$ are ordered. Hence, we only have the following one transition from G''.

$$G'' \xrightarrow{\ell_3} 1 \to 2 : k'\,\langle \mathsf{bool}\rangle.\,\mathsf{end} \xrightarrow{\ell_1} \mathsf{end}$$

This means the message from 1 is surely received at 2 *after* 1 received the message from 3; hence, two actions are not permutable. The semantics of the permutation will be clearer when we introduce the causality relation between the actions in Section 3.5.

3.3. Action Ordering

Henceforth, we refer to the acyclic directed graph of a global type G as a standard regular tree representation [Pierce 2002]. In order to give a definition, we annotate the actions in $\mathtt{p} \to \mathtt{p}' : k\,\langle U\rangle.\ G'$ and $\mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J}$ by a node name n.

Definition 3.3 (*Regular Tree Representation*). The *regular tree representation* $\mathsf{tree}(G)$ of a global type G is defined over the annotated unfolding of G such that

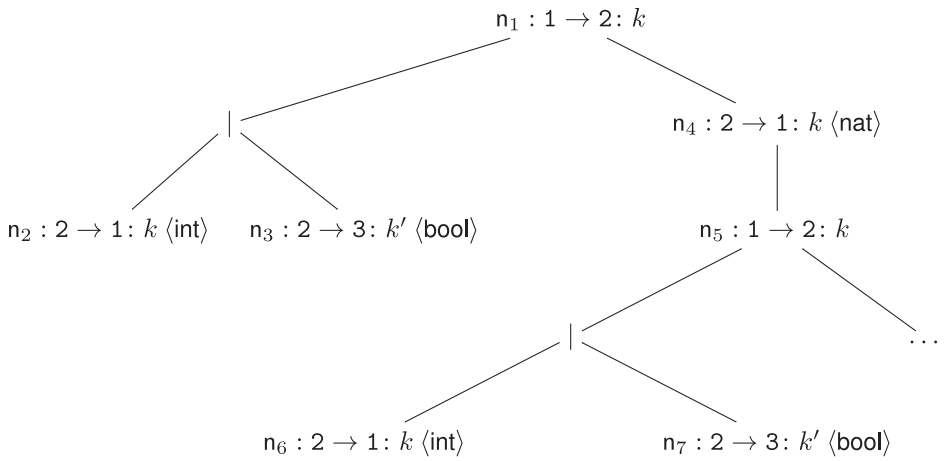
$\mathsf{tree}(\mathsf{n} : \mathtt{p} \to \mathtt{p}' : k\,\langle U\rangle.\ G')$ has root n with an edge to the root of $\mathsf{tree}(G')$
$\mathsf{tree}(\mathsf{n} : \mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J})$ has root n with edges to the roots of each $\mathsf{tree}(G_j)$ $(j \in J)$
$\mathsf{tree}(G_1 \mid G_2)$ has root $\mid$ with edges to the roots of each $\mathsf{tree}(G_i)$ $(i = 1, 2)$

Each node in $\mathsf{tree}(G)$ is labeled by the occurrence of its corresponding action, or it has no label in the case of parallel. These node names are unique in the unfolding.

As an example, the global type

$$\mu\mathbf{t}.\ 1 \to 2 : k \left\{ \begin{array}{l} l_1 : 2 \to 1 : k\,\langle\mathsf{int}\rangle.\ \mathsf{end} \mid 2 \to 3 : k'\,\langle\mathsf{bool}\rangle.\ \mathsf{end} \\ l_2 : 2 \to 1 : k\,\langle\mathsf{nat}\rangle.\ \mathbf{t} \end{array} \right\}$$

has the following (infinite) regular tree representation:



We now define:

Definition 3.4. An action from $\mathtt{p}$ to $\mathtt{p}'$ at k is in a global type G, written $\mathtt{p} \to \mathtt{p}' : k \in G$, whenever, in the regular tree representation of G, there exists some node n with label $\mathtt{p} \to \mathtt{p}' : k$. We write $\mathsf{n} = \mathtt{p} \to \mathtt{p}' : k$ if n has label $\mathtt{p} \to \mathtt{p}' : k$.

Definition 3.5. We denote:

—$\mathsf{pid}(G)$ for the set of participants occurring in G (but not in any carried types).
—$\mathsf{sid}(G)$ for the number of the set of session channel indices in G (but not in any carried types).

For example, if $G = 1 \to 3 : k\,\langle\mathsf{int}\rangle.\ 3 \to 2 : k'\,\langle\mathsf{bool}\rangle.\ \mathsf{end}$, then $\mathsf{pid}(G) = \{1, 2, 3\}$ and $\mathsf{sid}(G) = 2$.

Convention 3.6. We assume that, in each action from $\mathtt{p}$ to $\mathtt{p}'$, we have $\mathtt{p} \neq \mathtt{p}'$; that is, we prohibit reflexive interaction.

Here, we define the relation $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$, which holds whenever n_1 directly or indirectly occurs before n_2 in the regular tree representation of G. For instance, in the global type $G = \mathtt{p}_1 \to \mathtt{p}_1' : k_1 \langle U_1 \rangle.\ \mathtt{p}_2 \to \mathtt{p}_2' : k_2 \langle U_2 \rangle.\ G_2'$ we have that $\mathtt{p}_1 \to \mathtt{p}_1' : k_1 \prec \mathtt{p}_2 \to \mathtt{p}_2' : k_2 \in G$.

Definition 3.7 (*Action Ordering).* We define $\prec$ as the least partial order such that:

(1)	$\mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k \langle U \rangle.\ G'$	if $\mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k$ and $\mathsf{n}_2 \in G'$
(2)	$\mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k \{l_j : G_j\}_{j \in J}$	if $\mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k$ and $\exists i \in J.\, \mathsf{n}_2 \in G_i$
(3)	$\mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k \langle U \rangle.\ G'$	if $\mathsf{n}_1 \prec \mathsf{n}_2 \in G'$
(4)	$\mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p}_1 \to \mathtt{p}_1' : k' \{l_j : G_j'\}_{j \in J}$	if $\mathsf{n}_1 \prec \mathsf{n}_2 \in G_i'$ for some $i \in J$
(5)	$\mathsf{n}_1 \prec \mathsf{n}_2 \in G_1 \mid G_2$	if $\mathsf{n}_1 \prec \mathsf{n}_2 \in G_i$ for some $i \in \{1, 2\}$

Here, Lines 1 and 2 say that, in values and branching types, any nested action comes always after the top one for value and branch types. Lines 3 and 4 say that if two actions are related by the action ordering in a subterm of some global type G, then they are also related in G. Parallel composition and recursion are dealt with by Lines 5 and 6.

The action ordering allows us to express intended causal dependencies in global types, which is subtle under asynchronous semantics. Consider the following simple global type:

$$G = \mathtt{A} \to \mathtt{B} : k \langle U \rangle.\ \ \mathtt{A} \to \mathtt{C} : k' \langle U' \rangle.\ \mathsf{end} \tag{5}$$

where A, B, and C denote participants. We use this example to show an important difference between asynchronous and synchronous communication. In a "synchronous" interpretation of Equation (5), the ordering would mean "only after the first sending and receiving take place, the second sending and receiving take place." This is a suitable reading when sending and receiving constitute a single atomic action, as in synchronous languages, but *not* with asynchronous communication, where it is hard to impose such an ordering, since messages to distinct channels may not arrive in order (e.g., C may receive the second output from A before its first message reaches B). This corresponds to [GR3] and [GR4] in Definition 3.2, where the action ℓ can be executed before the action in the prefix.

Thus, the present theory takes a more liberal interpretation of $\prec$, imposing sequencing *only on the actions of the same participant in ordered actions*. For example, in Equation (5), A's two sending actions are ordered, but B's and C's receiving actions are not. This relation is explained in the next subsection with several examples.

3.4. Examples of Global Types

Example 3.8 (*Two-Buyer Protocol*). The following is a global type of the two-buyer-protocol in Section 2.3. We write participants and channels with legible symbols although they are actually numbers (e.g., $\mathtt{Bi} = \mathtt{i}$, $\mathtt{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$, and $s = 4$):

1. $\mathtt{B1} \to \mathtt{S} :\ \ s\langle \mathsf{string} \rangle.$
2. $\mathtt{S} \to \mathtt{B1} :\ \ b_1\langle \mathsf{int} \rangle.$
3. $\mathtt{S} \to \mathtt{B2} :\ \ b_2\langle \mathsf{int} \rangle.$
4. $\mathtt{B1} \to \mathtt{B2} :\ b_2'\langle \mathsf{int} \rangle.$
5. $\mathtt{B2} \to \mathtt{S} :\ \ s \left\{ \begin{array}{l} \mathsf{ok} : \mathtt{B2} \to \mathtt{S} : s \langle \mathsf{string} \rangle.\ \mathtt{S} \to \mathtt{B2} : b_2 \langle \mathsf{date} \rangle.\ \mathsf{end}, \\ \mathsf{quit} : \mathsf{end} \end{array} \right\}$

The type gives a clear, abstract view of the whole conversation scenario. The following are several salient points in the asynchronous interpretation of this type:

—Consider Lines 3 and 4. Since they have different senders, the sending actions are unordered in spite of their $\prec$-ordering. *Hence, if* $b_2 = b_2'$, *two messages have a conflict at* s (i.e., lose the ordering).
—Next, we consider the following causal chain from Line 1 to Line 3 to Line 5:

$$\texttt{B1} \;\rightarrow\; \texttt{S} \;\prec\; \texttt{S} \;\rightarrow\; \texttt{B2} \;\prec\; \texttt{B2} \;\rightarrow\; \texttt{S}$$

Here, $\rightarrow$ can be interpreted as the ordering given by message delivery (see previous subsection), whereas $\prec$ is the action ordering. Note in particular that two sending actions by `B1` (Line 1) and by `B2` (Line 5), both done at s, are causally ordered. By focusing on $\prec$ from the first `S` (of Line 1) to the last `S` (of Line 5), the receiving actions in Line 1 and the first $\texttt{B1} \rightarrow \texttt{S}$ in Line 5 are also ordered. Since the interaction in Line 1 will surely take place before the interaction in Line 5, no conflict occurs between these two communications in spite of their use of a common channel s.

Example 3.9 (*Streaming Protocol*). We now present the global type of the simple streaming protocol in Section 2.3. Here, we unfold its recursion once and set: $d = 1$, $k = 2$, $c = 3$, $\texttt{K} = 1$, $\texttt{DP} = 2$, $\texttt{C} = 3$ and $\texttt{KP} = 4$.

$$\begin{array}{lll}
1.\ \mu\mathbf{t}.\ \texttt{DP} \rightarrow \texttt{K} : d\,\langle \mathsf{bool} \rangle. & & 4.\ \texttt{DP} \rightarrow \texttt{K} : d\,\langle \mathsf{bool} \rangle. \\
2. \qquad \texttt{KP} \rightarrow \texttt{K} : k\,\langle \mathsf{bool} \rangle. & & 5.\ \texttt{KP} \rightarrow \texttt{K} : k\,\langle \mathsf{bool} \rangle. \\
3. \qquad \texttt{K} \rightarrow \texttt{C} : c\,\langle \mathsf{bool} \rangle. & & 6.\ \texttt{K} \rightarrow \texttt{C} : c\,\langle \mathsf{bool} \rangle.\mathbf{t}
\end{array}$$

The following arguments hold for any n-fold unfoldings.

—Lines 1 and 2 are temporally unordered in sending, but this does not cause conflict since channels d and k are distinct.
—Line 1 and its unfolding, Line 4, share d. But the two use the same sender and the same receiver, so each pair of actions are $\prec$-ordered, hence safe. Similarly for other unfolded actions.

Example 3.10 (*Instrument Controlling*). We now present another example from OOI [2015] that focuses on the usage of an instrument through repeated commands, together with checking privileges initially and later reporting the status to the central operator in charge of the instrument. The global type description involves a user `User`, an operator `Op`, and the instrument `Instr` and is given as follows.

$$\texttt{User} \rightarrow \texttt{Op} : 1\,\langle \mathsf{privilege} \rangle.$$
$$\texttt{Op} \rightarrow \texttt{User} : 2 \left\{ \begin{array}{l} \mathsf{ok} :\ \mu\mathbf{t}. \\ \qquad \texttt{User} \rightarrow \texttt{Instr} : 3 \left\{ \begin{array}{l} \mathsf{move} :\ \mathbf{t} \\ \mathsf{photo} :\ \mathbf{t} \\ \mathsf{quit} :\ \texttt{Instr} \rightarrow \texttt{Op} : 4\,\langle \mathsf{string} \rangle.\ \mathsf{end} \end{array} \right\} \\ \\ \mathsf{no} :\ \mathsf{end} \end{array} \right\}$$

Note that the protocol description given by the global type here can have several implementations. In particular, the instrument can be used with any combination of the operations Move and Photo. However, any sequence will be terminated by Quit.

(II) Good

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{C} \to \mathtt{B} : k'$
$s_k! \mid (s_k?; s_{k'}?) \mid s_{k'}!$

(II) Bad

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{C} \to \mathtt{B} : k$
$s_k! \mid (s_k?; s_k?) \mid s_k!$

(IO) Good

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{B} \to \mathtt{C} : k'$
$s_k! \mid (s_k?; s_{k'}!) \mid s_{k'}?$

(IO) Bad

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{B} \to \mathtt{C} : k$
$s_k! \mid (s_k?; s_k!) \mid s_k?$

(OO, II) Good

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{A} \to \mathtt{B} : k$
$(s_k!; s_k!) \mid (s_k?; s_k?)$

(OI) Bad

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{C} \to \mathtt{A} : k$
$(s_k!; s_k?) \mid s_k? \mid s_k!$

(OO) Bad

$\mathtt{A} \to \mathtt{B} : k$
$\mathtt{A} \to \mathtt{C} : k$
$s_k? \mid (s_k!; s_k!) \mid s_k?$

Fig. 5. Causality analysis.

Here, we give a possible implementation of each participant:

$$\text{User} \stackrel{\text{def}}{=} s_1!\langle high\rangle; s_2 \rhd \left\{ \begin{array}{l} \mathsf{ok}: \ s_3 \lhd \mathsf{move};\ s_3 \lhd \mathsf{photo};\ s_3 \lhd \mathsf{quit};\ \mathbf{0} \\ \mathsf{no}: \ \mathbf{0} \end{array} \right\}$$

$$\text{Operator} \stackrel{\text{def}}{=} s_1?(x);\ \mathsf{if}\ f(x)\ \mathsf{then}\ \ s_2 \lhd \mathsf{ok};\ \mathbf{0}\ \ \mathsf{else}\ \ s_2 \lhd \mathsf{no};\ \mathbf{0}$$

$$\text{Instrument} \stackrel{\text{def}}{=} \mu \mathbf{t}.\ s_3 \rhd \left\{ \begin{array}{l} \mathsf{move}: \ \mathbf{t} \\ \mathsf{photo}: \ \mathbf{t} \\ \mathsf{quit}: \ s_4!\langle report\rangle; \mathbf{0} \end{array} \right\}$$

3.5. A Safety Principle for Global Types: Linearity of Channels

For a conversation in a session to proceed properly, it is desirable that there is no conflict (racing) at session channels. The process $s_k!\langle \mathsf{true}\rangle \mid s_k!\langle 5\rangle \mid s_k?x;\ \mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is a typical example of a race at channel s_k: If the second output synchronizes with the first input, we have a runtime error when evaluating the guard of the conditional. To ensure absence of such races, when a *common* channel is used in two communications, their sending actions and their receiving actions should (respectively) be ordered temporally (causality) so that no confusion arises at sending or receiving. If a global type satisfies this principle, then it specifies an ordering of interactions and can be used as a basis of guaranteeing process behaviors through type-checking. The correspondence between the linearity property and the LTS of the global types defined in Definition 3.2 will be used for proving the main theorems, Subject Reduction Theorem (Theorem 5.19) and Session Fidelity Theorem (Corollary 5.23).

Causality is induced in several ways in the present asynchronous model. We summarise all essential cases in Figure 5, with concrete process instances for illustration.

In the figure, IO indicates a causal ordering from input (receiving) to output (sending), similarly for II, OO, and OI. In (II)-Bad, we demand $\mathtt{A} \neq \mathtt{C}$. We observe:

—The "good" and "bad" cases for II show that II alone is safe only when two channels differ. Similarly for IO.
—In OO,II, two outputs have the same sender and the same channel, so (by *message order-preservation*) outputs are ordered. Inputs are also ordered by $\prec$; hence, they are safe.
—There is no ordering from output to input (due to asynchrony), so OI gives us no dependency.

These observations lead to the following causal relations on global types.

Definition 3.11 (*Dependency Relations*). Fix G. The relation $\prec_\phi$, with $\phi \in \{\mathsf{II}, \mathsf{IO}, \mathsf{OO}\}$, over actions is generated from:

$$\begin{array}{l}
\mathsf{n}_1 \prec_{\mathtt{II}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2 \in G \text{ and } \mathsf{n}_i = \mathtt{p}_\mathtt{i} \to \mathtt{p} : k_i \ (i = 1, 2) \\
\mathsf{n}_1 \prec_{\mathtt{IO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2 \in G, \mathsf{n}_1 = \mathtt{p}_1 \to \mathtt{p} : k_1 \text{ and } \mathsf{n}_2 = \mathtt{p} \to \mathtt{p}_2 : k_2. \\
\mathsf{n}_1 \prec_{\mathtt{OO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2 \in G, \mathsf{n}_i = \mathtt{p} \to \mathtt{p}_\mathtt{i} : k \ (i = 1, 2).
\end{array}$$

—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 $\phi_i \in \{\mathsf{IO}\}$ for $1 \leq i \leq n-1$ and $\phi_n = \mathsf{II}$.
—An *output dependency from* n_1 *to* n_2 is a chain $\mathsf{n}_1 \prec_{\phi_1} \cdots \prec_{\phi_n} \mathsf{n}_2$ $(n \geq 1)$ such that $\phi_i \in \{\mathsf{OO}, \mathsf{IO}\}$.

Note that, in the input dependency, the last II-ordering is necessary. In fact, if we allow the dependency to end with an IO-edge, an input at n_2 is not checked. We can further clarify dependencies with the following graphical examples:

In picture (a), there is an output dependency from the first to the third line that has been marked as OO, and an input dependency through all lines. However, in (b), we only have an output dependency. It is clear that, although (a) could be implemented in an asynchronous setting, the conversation in (b) would cause problems. In fact, the messages sent by A on k_1 could be delivered in the wrong order (first to C and then to B). The notion of linearity, hereby introduced, precisely captures such inconsistencies in global types.

Definition 3.12 (*Linearity*). G is *linear* whenever, for all $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ such that $\mathsf{n}_i = \mathtt{p}_\mathtt{i} \to \mathtt{p}'_\mathtt{i} : k$ $(i = 1, 2)$, both input and output dependencies from n_1 to n_2 exist. We inductively apply this constraint to all global types which G carries.

Observe that we do not require ordering between $\mathsf{n}_i \in G_k$ and $\mathsf{n}_j \in G_h$ in $\mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j \in J}$ (for $h, k \in J$, $h \neq k$) since only one branch is performed. In fact, they cannot be

related by $\prec$ according to Definition 3.7. We further clarify the condition on branching with an example:

$$\mathtt{A} \to \mathtt{B} : t \left\{ \begin{array}{r} \mathsf{ok} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end}, \\ \mathsf{quit} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end} \end{array} \right\} \qquad \mathtt{A} \to \mathtt{B} : t. \left(\begin{array}{l} \mathtt{C} \to \mathtt{D} : s.\mathsf{end} \mid \\ \mathtt{C} \to \mathtt{D} : s.\mathsf{end} \end{array} \right)$$

(a) branching (b) parallel

The type (a) represents branching: Since only one branch is selected, there is no conflict between the two actions $\mathtt{C} \to \mathtt{D} : s$. On the other hand, (b) denotes a concurrent execution of two independent $\mathtt{C} \to \mathtt{D} : s$, so an input conflict at $\mathtt{D}$ exists.

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

$$\mu \mathbf{t}.(\mathtt{A} \to \mathtt{B} : s.\mathsf{end} \mid \mathtt{B} \to \mathtt{A} : t.\mathbf{t}).$$

This is linear in its 0-th unfolding (i.e., we replace $\mathbf{t}$ with end), but, when unfolded once, it becomes nonlinear, as follows:

$$\mathtt{A} \to \mathtt{B} : s.\mathsf{end},\ \mathtt{B} \to \mathtt{A} : t.\mu \mathbf{t}.(\mathtt{A} \to \mathtt{B} : s.\mathsf{end} \mid \mathtt{B} \to \mathtt{A} : t.\mathbf{t})$$

since the two actions $\mathtt{A} \to \mathtt{B} : s$ appear in parallel. This is witnessed by:

$$\mathsf{def}\ X\langle st \rangle = ((s! \mid t?.s!.X\langle ts \rangle) \mid s?.t!)\ \mathsf{in}\ X\langle ts \rangle,$$

where $(s! \mid t?.s!.X\langle ts \rangle)$ belongs to $\mathtt{A}$ and $s?.t!$ belongs to $\mathtt{B}$. Unfolding once is necessary also in global types that do not contain parallel global types. The example in Equation 6 shows a global type that satisfies the linearity condition:

$$\mu \mathbf{t}.\mathtt{A} \to \mathtt{B} : s.\mathtt{B} \to \mathtt{C} : s'.\mathtt{A} \to \mathtt{C} : s.\mathbf{t}. \tag{6}$$

However, when unfolded once, it is no longer linear as:

$$\mathtt{A} \to \mathtt{B} : s.\mathtt{B} \to \mathtt{C} : s'.\mathtt{A} \to \mathtt{C} : s.\mu \mathbf{t}.\mathtt{A} \to \mathtt{B} : s.\mathtt{B} \to \mathtt{C} : s'.\mathtt{A} \to \mathtt{C} : s.\mathbf{t} \tag{7}$$

since there is no input and output dependencies between $\mathtt{A} \to \mathtt{C} : s$ and $\mathtt{A} \to \mathtt{B} : s$.

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. For example, the type in Equation (6) would be first transformed into the type in Equation (7) and finally become:

$$\mathtt{A} \to \mathtt{B} : s.\mathtt{B} \to \mathtt{C} : s'.\mathtt{A} \to \mathtt{C} : s.\mu \mathbf{t}.\mathtt{A} \to \mathtt{B} : s.\mathtt{B} \to \mathtt{C} : s'.\mathtt{A} \to \mathtt{C} : s.\mathsf{end}$$

Proposition 3.13.

(1) *The one-time unfolding of a global type is linear if and only if its n-th unfolding is linear.*
(2) *The linearity of a global type is decidable.*

Proof. For Property (1), the if-direction is obvious. The only if-direction is proved by induction on n. See Appendix A for the full proofs. Property (2) is an immediate corollary of (1).

Proposition 3.14. *Suppose G is linear and $G \xrightarrow{\ell} G'$. Then G' is linear.*

Proof. By induction on the last LTS rule applied. The cases [GR1,GR2,GR5,GR6] are obvious. We prove the case [GR3]. The case [GR4] is similar. Suppose $G = \mathtt{p}_1 \to \mathtt{p}_2 : k\langle U \rangle$. $G_1 \xrightarrow{\ell} \mathtt{p}_1 \to \mathtt{p}_2 : k\langle U \rangle$. $G_2 = G'$ is derived by $G_1 \xrightarrow{\ell} G_2$ with $\mathtt{p}_2 \notin \ell$. Assume $\ell = \mathtt{q}_1 \to \mathtt{q}_2 : k'\langle U' \rangle$. We first prove if G satisfies the linearity condition, then

$k \neq k'$. Suppose by contradiction, $k = k'$. Then there should be both output and input causalities from $\mathsf{n} = \mathtt{p}_1 \to \mathtt{p}_2 : k$ to $\mathsf{n}' = \mathtt{q}_1 \to \mathtt{q}_2 : k$. If there is the IO-causality from n to n' in G, then we cannot apply [GR3]. Hence, there is only OO and II causalities from n to n'. In this case, we should have $\mathtt{p}_2 = \mathtt{q}_2$. This contradicts $\mathtt{p}_2 \notin \ell$ in [GR3]. Thus we assume $k \neq k'$. Then there are three cases.

Case (a): If $\mathtt{p}_1, \mathtt{p}_2, \mathtt{q}_1, \mathtt{q}_2$ are pairwise-distinct, then $\mathtt{p}_1 \to \mathtt{p}_2 : k \not\prec_\phi \mathtt{q}_1 \to \mathtt{q}_2 : k'$ in G. Thus, no dependency relation from $\mathtt{p}_1 \to \mathtt{p}_2 : k$ to any action $\mathsf{n}' \neq \mathtt{q}_1 \to \mathtt{q}_2 : k'$ in G_1 is changed before and after the transition. Hence, obviously, $\mathtt{p}_1 \to \mathtt{p}_2 : k \prec_\phi \mathsf{n}' \in G$ implies $\mathtt{p}_1 \to \mathtt{p}_2 : k \prec_\phi \mathsf{n}' \in G'$ for all n' such that $\mathsf{n}' \neq \mathtt{q}_1 \to \mathtt{q}_2 : k'$. Hence G' is linear.

Case (b): Suppose $\mathtt{p}_1 = \mathtt{q}_2$ and others are pairwise distinct. Then $\mathtt{p}_1 \to \mathtt{p}_2 : k \not\prec_\phi \mathtt{q}_1 \to \mathtt{p}_1 : k'$ in G again. Hence, by the same reasoning as previously, $\mathtt{p}_1 \to \mathtt{p}_2 : k \prec_\phi \mathsf{n}' \in G$ implies $\mathtt{p}_1 \to \mathtt{p}_2 : k \prec_\phi \mathsf{n}' \in G'$ in $\mathsf{n}' \neq \mathtt{q}_1 \to \mathtt{q}_2 : k'$.

Case (c): Suppose $\mathtt{p}_1 = \mathtt{q}_1$ and others are pairwise distinct. Then, again, we have $\mathtt{p}_1 \to \mathtt{p}_2 : k \not\prec_\phi \mathtt{q}_1 \to \mathtt{q}_2 : k'$ in G. The remaining is the same as the previous cases. □

4. TYPE DISCIPLINE FOR MULTIPARTY SESSIONS

4.1. Programming Methodology for Multiparty Interactions

Once given global types as a description of global interactions among communicating processes, we can consider the following development steps for programs with multiparty sessions.

Step 1. A programmer describes an intended interaction scenario as global type G and checks that it is linear.
Step 2. She develops code, one for the local behavior of each participant, incrementally validating its conformance to the projection of G onto each participant by efficient type-checking.

The local behaviors might be developed by a team of programmers (who may well be distributed geographically), in which case the use of a clear, precise global description is all the more essential. When programs are executed, their interactions are guaranteed to follow the stipulated scenario. Furthermore, when transport issues interfere with communication, the global type gives a basic criteria by which communications are monitored and (in)validated at runtime. The type specification also serves as a basis for debugging, maintenance and upgrade.

For all these purposes, we need a type discipline that relates global types to communication behavior of individual (endpoint) programs and guarantees key properties such as communication safety. This section introduces such a type discipline.

4.2. Endpoint Types

Syntax. *Endpoint session types* or *endpoint types*, ranged over by $T, T', \ldots$, are types for the endpoint behavior of processes, acting as a link between the global types in Section 3, which give intended conversation structures of multiparty sessions, and processes in Section 2.1. The grammar is given in Figure 6 (the grammars for U and S are repeated from Figure 4). All constructs come from binary session types [Honda et al. 1998] except for the following major changes for multiparty interactions.

—Since a process uses multiple channels for addressing multiple parties, a session type records the identity (number) of the session channel it uses at each action type.
—Since a type is used for type-checking each participant, we use a notation $T@\mathtt{p}$ (called *located type*) representing an endpoint type T assigned to participant $\mathtt{p}$. A located type is also used for delegation.

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

Fig. 6. Syntax of endpoint session types.

The remainder remains identical to the original session types [Honda et al. 1998]. Type $k?\langle U \rangle; T$ represents the behavior of inputting values of type U at s_k (assume $s_1 \ldots s_n$ is shared at initialization), then performing the actions represented by T. Similarly, $k!\langle U \rangle; T$ is for sending.

Type $k\&\{l_i : T_i\}_{i \in I}$ describes a branching (external choice): It waits with n options at k, and then behaves as type T_i if the i-th label is selected; type $k \oplus \{l_i : T_i\}_{i \in I}$ represents the behavior that selects one of the labels, say l_i at k, then behaves as T_i (internal choice). These four are *action prefixes* in endpoint types. We call send and selection types *output types* and receive and branching *input types*. The rest is the same as the global types, demanding that type variables occur guarded by a prefix and taking an equi-recursive approach for recursive types. We often omit end. Note that endpoint types do not contain parallel composition, hence retaining simplicity.

Projection and Coherence. The following defines the projection of a global type to endpoint types at each participant.

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

$$
- \quad (\mathtt{p}_1 \to \mathtt{p}_2 : k\,\langle U \rangle.\, G') \upharpoonright \mathtt{p} = \begin{cases} k!\langle U \rangle; (G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_1 \neq \mathtt{p}_2 \\ k?\langle U \rangle; (G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_2 \neq \mathtt{p}_1 \\ (G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} \neq \mathtt{p}_2 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \end{cases}
$$

$$
- \quad (\mathtt{p}_1 \to \mathtt{p}_2 : k\,\{l_j : G_j\}_{j \in J}) \upharpoonright \mathtt{p} = \begin{cases} k \oplus \{l_j : (G_j \upharpoonright \mathtt{p})\}_{j \in J} & \text{if } \mathtt{p} = \mathtt{p}_1 \neq \mathtt{p}_2 \\ k\&\{l_j : (G_j \upharpoonright \mathtt{p})\}_{j \in J} & \text{if } \mathtt{p} = \mathtt{p}_2 \neq \mathtt{p}_1 \\ (G_1 \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} \neq \mathtt{p}_2 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \\ & \text{and } \forall i, j \in J. G_i \upharpoonright \mathtt{p} = G_j \upharpoonright \mathtt{p} \end{cases}
$$

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

$$
- \quad (\mu \mathbf{t}.G) \upharpoonright \mathtt{p} = \begin{cases} \mu \mathbf{t}.(G \upharpoonright \mathtt{p}) & \text{if } G \upharpoonright \mathtt{p} \neq \mathsf{end} \\ \mathsf{end} & \end{cases} \qquad \mathbf{t} \upharpoonright \mathtt{p} = \mathbf{t} \qquad \mathsf{end} \upharpoonright \mathtt{p} = \mathsf{end}.
$$

When none of the side conditions holds, the map is undefined.

We regard the map to act on the syntax of global types. In the branching clause, all the projections of those participants whose behavior does not depend on the branching should generate an identical endpoint type (otherwise undefined); and, in parallel composition, $\mathtt{p}$ should be contained in at most a single type, ensuring each type is single-threaded. Note that, for the sake of clarity, we forbid reflexive interactions directly in

the definition of projection, making Convention 3.6 redundant. Here, in Property (2), the term $T_{\mathtt{p}}@\mathtt{p}$ was introduced at the beginning of Section 4.2.

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.*

PROOF. By Proposition 3.13 Property (2), noting that the projection is only applied to a given global type without unfolding. A complexity analysis is given in Deniélou and Yoshida [2010]. □

PROPOSITION 4.4. *Assume G is coherent and* $G \xrightarrow{\ell} G'$. *Then G′ is coherent.*

PROOF. By Proposition 3.14, we only have to prove if G is projectable, then G' is projectable. This can be done by induction on the last LTS rule applied. Cases [GR1,GR2] are straightforward by definition of projection, whereas [GR3,GR4,GR5,GR6] follow immediately by induction hypothesis. □

If the projection mapping is undefined, a global type is not coherent. Linearity guarantees linear channel usage including message-order preservation. The next examples demonstrate the need of these conditions.

4.3. Examples of Coherence

The following global type is linear but *not* coherent because the projection is undefined:

$$\mathtt{A} \to \mathtt{B} : k\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool}\rangle,\ \mathsf{quit} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{nat}\rangle\}. \tag{8}$$

Intuitively, when we project this type onto C or D, regardless of the choice made by A, they should behave in the same way: Participants C and D should be independent threads. If we change the nat to bool as:

$$\mathtt{A} \to \mathtt{B} : k\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool}\rangle, \mathsf{quit} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool}\rangle\}, \tag{9}$$

we can define the coherent projection as follows:

$$\{k \oplus \{\mathsf{ok} : \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}@\mathtt{A},\ k\&\{\mathsf{ok} : \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}@\mathtt{B}\\ k'!\langle \mathsf{bool}\rangle@\mathtt{C},\ k'?\langle \mathsf{bool}\rangle@\mathtt{D}\}.$$

As examples of endpoint types that are not coherent, consider processes in the second case of Figure 5:

$$(\mathsf{II})\ \text{Bad}\ \{s!\langle\rangle@\mathtt{A},\ s?\langle\rangle; s?\langle\rangle@\mathtt{B},\ s!\langle\rangle@\mathtt{C}\}.$$

This process is not coherent since the corresponding global type $\mathtt{A} \to \mathtt{B} : s.\mathtt{C} \to \mathtt{B} : s$ is not linear.

4.4. Typing System

The purpose of the typing system is to efficiently type behaviors *that are built by programmers* and hence that do not include runtime elements such as queues.

Environments and Type Algebra. 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:

$$\Gamma\ ::=\ \emptyset\ \mid\ \Gamma, u : S\ \mid\ \Gamma, X : \tilde{S}\tilde{T} \qquad \Delta\ ::=\ \emptyset\ \mid\ \Delta, \tilde{s} : \{T@\mathtt{p}\}_{\mathtt{p}\in I.}$$

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to sorts or from process variables to sequences of sorts and types. *Typing* $(\Delta, \Delta', ..)$ records linear usage of session channels. In binary sessions types, it assigned a type to a single channel; now, it assigns a family of located types to a vector of session channels.

Notation 4.5.

—We write Δ, Δ' to denote a typing made from the disjoint union of Δ and Δ', always assuming their domains contain disjoint sets of session channels.
—We write $\tilde{s} : T@\mathtt{p}$ for a singleton typing $\tilde{s} : \{T@\mathtt{p}\}$.

A family of located types is needed to link a set of session types to types of a set of processes created by the session initialization.

Typing System. The type assignment system for processes is given in Figure 7. We use the following judgments for processes and expressions, respectively:

$$\Gamma \vdash P \triangleright \Delta \qquad\qquad \Gamma \vdash e : S.$$

These read "under the environment Γ, process P has typing Δ" and "under the environment Γ, expression e has type S". If we set $|\tilde{s}| = 1$ and $\mathtt{n} = 2$, and we delete $\mathtt{p}$ from located type, the rules are essentially identical to those for the original binary session [Yoshida and Vasconcelos 2007]. Here, we explain the key rules.

[Name], [Bool], [Or] are the rules for the expressions and identical with [Yoshida and Vasconcelos 2007].

[Mcast] is the rule for session request. The condition $\Gamma \vdash a : \langle G \rangle$ says that sessions established on shared channel a will execute according to global type G. Therefore, $\tilde{s}$ must be used in the body P as the *first* projection of G. Note how $\tilde{s}$ are bound in $\overline{a}[2..\mathtt{n}](\tilde{s}).P$ and therefore disappear from the typing. [Macc] is for the session accept, taking the $\mathtt{p}$-th projection. The endpoint type $(G \upharpoonright \mathtt{p})@\mathtt{p}$ means that the participant $\mathtt{p}$ has $G \upharpoonright \mathtt{p}$, which is the projection of G onto $\mathtt{p}$ as its endpoint type. In both rules, condition $|\tilde{s}| = \mathsf{sid}(G)$ (see Definition 3.5) ensures the number of session channels meets those in G. The typing $\tilde{s} : T@\mathtt{p}$ (stands for $\tilde{s} : \{T@\mathtt{p}\}$) means that each prefix does not contain parallel threads which share $\tilde{s}$.

[Send] and [Rcv] are the rules for sending and receiving values. Since the k-th name $s[k]$ of $\tilde{s}$ is used as the subject, we record k in the type. Hence, vector $\tilde{s}$ has type $k!\langle \tilde{S} \rangle; T@\mathtt{p}$ in [Send] and type $k?\langle \tilde{S} \rangle; T@\mathtt{p}$ in [Rcv], under the assumption that it is used as $T@\mathtt{p}$ by the subterm P. Note how the relevant type prefixes $(k!\langle \tilde{S} \rangle$ for the output and $k?\langle \tilde{S} \rangle$ for the input) are composed. In both rules, "$\mathtt{p}$" in $T@\mathtt{p}$ ensures that P is (being inferred as) the behavior for participant $\mathtt{p}$, and its domain should be $\tilde{s}$.

[Deleg] and [SRec] are the rules for delegation of a session and its dual. Delegation of a multiparty session passes the whole remaining capability to participate in a multiparty session: Thus, operationally, we send the whole vector of session channels. The carried type T' is located, making sure that the behavior by the receiver at the passed channels takes the role of a specific participant (here $\mathtt{p}'$) in the delegated multiparty session. The rest follows the standard delegation rule [Yoshida and Vasconcelos 2007], observing [Deleg] says that $\tilde{t} : T'@\mathtt{p}'$ does not appear in P, symmetrically to [SRec], which uses the channels in P.

[Sel] and [Branch], identical with Yoshida and Vasconcelos [2007], are the rules for selection and branching.

[Conc] composes two processes if their endpoint types are disjoint.

[If], [Inact], [Var], and [Def] are standard. [NRes] is the restriction rule for shared name a.

In [Inact] and [Var], "end only" means Δ only contains end as session types.

$$\Gamma, a : S \vdash a : S \qquad \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{bool} \qquad \frac{\Gamma \vdash e_i \triangleright \mathsf{bool}}{\Gamma \vdash e_1 \mathsf{or}\ e_2 : \mathsf{bool}} \qquad \text{[NAME], [BOOL], [OR]}$$

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright \mathtt{1})@\mathtt{1} \quad \{\mathtt{1}, \ldots, \mathtt{n}\} = \mathsf{pid}(G) \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash \overline{a}_{[\mathtt{2..n}]}\,(\tilde{s}).P \triangleright \Delta} \qquad \text{[MCAST]}$$

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright \mathtt{p})@\mathtt{p} \quad \mathtt{p} \in \mathsf{pid}(G) \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash a_{[\mathtt{p}]}\,(\tilde{s}).P \triangleright \Delta} \qquad \text{[MACC]}$$

$$\frac{\forall j.\ \Gamma \vdash e_j : S_j \qquad \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@\mathtt{p}}{\Gamma \vdash s[k]!\langle \tilde{e} \rangle; P \triangleright \Delta, \tilde{s} : k!\,\langle \tilde{S} \rangle; T@\mathtt{p}} \qquad \text{[SEND]}$$

$$\frac{\Gamma, x : \tilde{S} \vdash P \triangleright \Delta, \tilde{s} : T@\mathtt{p}}{\Gamma \vdash s[k]?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k?\,\langle \tilde{S} \rangle; T@\mathtt{p}} \qquad \text{[RCV]}$$

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

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{s} : T@\mathtt{p},\ \tilde{t} : T'@\mathtt{p}'}{\Gamma \vdash s[k]?((\tilde{t})); P \triangleright \Delta, \tilde{s} : k?\,\langle T'@\mathtt{p}' \rangle; T@\mathtt{p}} \qquad \text{[SREC]}$$

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

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

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} \qquad \text{[CONC]}$$

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

$$\frac{\Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\boldsymbol{\nu}\, a)P \triangleright \Delta} \qquad \text{[INACT],[NRES]}$$

$$\frac{\Gamma \vdash \tilde{e} : \tilde{S} \qquad \Delta\ \mathsf{end\ only}}{\Gamma, X : \tilde{S}\tilde{T} \vdash X\langle \tilde{e}\tilde{s}_1..\tilde{s}_n \rangle \triangleright \Delta, \tilde{s}_1 : T_1@\mathtt{p}_1, .., \tilde{s}_n : T_n@\mathtt{p}_n} \qquad \text{[VAR]}$$

$$\frac{\Gamma, X : \tilde{S}\tilde{T}, \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{s}_1 : T_1@\mathtt{p}_1..\tilde{s_n} : T_n@\mathtt{p}_n \quad \Gamma, X : \tilde{S}\tilde{T} \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\tilde{s}_1..\tilde{s}_n) = P\ \mathsf{in}\ Q \triangleright \Delta} \qquad \text{[DEF]}$$

Fig. 7. Typing system for expressions and processes.

As for binary session types, the type-checking problem for programs is decidable. Here, we say Γ is *well-formed* when the global types it uses are all coherent. A program (or a process) is *annotated* when each of its ν-bound shared names is annotated by a well-formed global type.

PROPOSITION 4.6. *Let Γ be well-formed and P be an annotated program. Then it is decidable whether $\Gamma \vdash P \triangleright \emptyset$ is derivable or not.*

PROOF. By annotation, in each typing rule in Figure 7, the conclusion uniquely determines its premise(s). Note also, by well-formedness, projection of global types P may use is always well-defined. □

In this article, we leave open the generalization of the result to non-annotated programs and the corresponding result for type inference.

4.5. Typing Examples

Two-Buyer Protocol. Write $\overline{a}_{[2,3]}(b_1, b_2, b_2', s).Q_1$ and $a_{[2]}(b_1, b_2, b_2', s).Q_2$ for Buyer1 and Buyer2 in Section 2.3. Then, Q_1 and Q_2 have the following typing under $\Gamma = \{a : \langle G \rangle\}$ where G is given in the corresponding example in Section 3.4, letting $\mathsf{Bi} = i$, $\mathsf{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$, and $s = 4$ and assuming P_1, P_2, Q are $\mathbf{0}$:

$$\Gamma \vdash Q_1 \triangleright \tilde{s} : s!\langle \mathsf{string} \rangle; b_1?\langle \mathsf{int} \rangle; b_2'!\langle \mathsf{int} \rangle @ \mathtt{B1}$$
$$\Gamma \vdash Q_2 \triangleright \tilde{s} : b_2?\langle \mathsf{int} \rangle; b_2'?\langle \mathsf{int} \rangle; s \oplus \{\mathsf{ok} : s!\langle \mathsf{string} \rangle; b_2?\langle \mathsf{date} \rangle; \mathsf{end},\ \ \mathsf{quit} : \mathsf{end}\} @ \mathtt{B2}$$

Similarly for Seller. After prefixing at a, we can compose all three by [CONC].

A Streaming Protocol. Let $\Gamma = \{a : \langle G' \rangle\}$ where G' is from the streaming example in Section 3.4. Let $d = 1$, $k = 2$, $c = 3$, $\mathtt{K} = 1$, $\mathtt{DP} = 2$, $\mathtt{C} = 3$, and $\mathtt{KP} = 4$. Write R_1, R_2, R_3, and R_4 for the processes under the initial prefixes of Kernel, DataProducer, Consumer, and KeyProducer, respectively. We can type them as:

$$\Gamma \vdash R_1 \triangleright dkc : \mu \mathbf{t}.d?\langle \mathsf{bool} \rangle; k?\langle \mathsf{bool} \rangle; c!\langle \mathsf{bool} \rangle; \mathbf{t} @ \mathtt{K}$$
$$\Gamma \vdash R_2 \triangleright dkc : \mu \mathbf{t}.d!\langle \mathsf{bool} \rangle; \mathbf{t} @ \mathtt{DP} \qquad \Gamma \vdash R_4 \triangleright dkc : \mu \mathbf{t}.c?\langle \mathsf{bool} \rangle; \mathbf{t} @ \mathtt{C}$$

(R_4 is similar as R_2). Note that these types correspond to the projection of G' onto respective participants: Thus, Kernel, DataProducer, Consumer, and KeyProducer are typable programs under Γ, which can be composed to make the initial configuration.

Delegation. One source of the expressiveness of the session types comes from a facility of *delegation* (often called *higher order session passing*). We type the example in Section 3.4 and see the relationship with global and endpoint types. Consider the following three participants:

$$\begin{array}{lcl} \text{Alice} & \stackrel{\text{def}}{=} & \overline{a}_{[2]}(t_1, t_2).\overline{b}_{[2,3]}(s_1, s_2).t_1!\langle\!\langle s_1, s_2 \rangle\!\rangle; \mathbf{0} \\ \text{Bob} & \stackrel{\text{def}}{=} & a_{[2]}(t_1, t_2).b_{[1]}(s_1, s_2).t_1?(\!(s_1, s_2)\!); s_1!\langle 1 \rangle; \mathbf{0} \\ \text{Carol} & \stackrel{\text{def}}{=} & b_{[2]}(s_1, s_2).s_1?(x); P, \end{array}$$

where Alice delegates its capability to Bob. Since there are two multicastings, there are two global specifications, one for a and another for b as follows:

$$\begin{array}{rcl} G_a & = & \mathtt{A} \to \mathtt{B} : t_1\ \langle s_1!\langle \mathsf{int} \rangle @ \mathtt{A} \rangle.\ \mathsf{end} \\ G_b & = & \mathtt{A} \to \mathtt{C} : s_1\ \langle \mathsf{int} \rangle.\ \mathsf{end}, \end{array}$$

where the type $s_1!\langle \mathsf{int} \rangle @ \mathtt{A}$ means the capability to send an integer from participant $\mathtt{A}$ via channel s_1. This capability is passed to $\mathtt{B}$ so that $\mathtt{B}$ behaves as $\mathtt{A}$. However, since the two specifications are independent, $\mathtt{C}$ does not have to know who would pass the capability.

Let $(\text{Alice} \mid \text{Bob} \mid \text{Carol}) \to (\nu \tilde{t}\tilde{s})(A \mid B \mid C \mid R)$ where A, B, C are the processes of Alice, Bob, and Carol after the initial multicasting and R is the generated queues. Let

$s_1 = 1, t_1 = 1, \mathtt{A} = 1, \mathtt{B} = 2, \mathtt{C} = 3$. We have the following typings under Γ with $P \equiv \mathbf{0}$:

$$\Gamma \vdash A \triangleright \tilde{t} : t_1!\langle s_1!\langle \mathsf{int}\rangle@\mathtt{A}\rangle@\mathtt{A},\ \tilde{s} : s_1!\langle \mathsf{int}\rangle@\mathtt{A}$$
$$\Gamma \vdash B \triangleright \tilde{t} : t_1?\langle s_1!\langle \mathsf{int}\rangle@\mathtt{A}\rangle@\mathtt{B}$$
$$\Gamma \vdash C \triangleright \tilde{s} : s_1?\langle \mathsf{int}\rangle@\mathtt{C},$$

where each endpoint type reflects the original global specifications (e.g., Carol does not know Alice passed the capability to Bob, and Bob behaves as Alice). These types give projections of G_a and G_b.

5. SAFETY AND PROGRESS

This section establishes the fundamental behavioral properties of typed processes. We follow three technical steps:

(1) We extend the typing rules to include those for runtime processes that involve message queues.
(2) We define reduction over session typings, which eliminates a pair of minimal complementary actions from endpoint types.
(3) We then relate the reduction of processes and that of typings: Showing the latter follows the former gives us *subject reduction* (Theorem 5.19), *safety* (Theorem 5.22), and *session fidelity* (Corollary 5.23), whereas showing the former follows the latter under a certain condition gives us *progress* (Corollary 5.30).

By the correspondence between endpoint types and global types, these results guarantee that interactions between typed processes exactly follow the conversation scenario specified in a global type.

Note that the typing system for runtime processes that we introduce in this section is used solely for establishing the behavioral properties of typed processes, tracing how typability is preserved during reduction. This is in contrast to the simple typing system in Section 4, which is for typing programs and program phrases.

5.1. Typing Runtime

How to Type a Queue. We first illustrate a key idea underlying our runtime typing using the following example:

$$\underbrace{s!\langle 3\rangle; s!\langle \mathsf{true}\rangle; \mathbf{0}}_{1} \ \mid\ s :: \emptyset \ \mid\ \underbrace{s?(x); s?(y); \mathbf{0}}_{2}. \tag{10}$$

Here, process 1 sends an integer and a boolean to process 2 through queue $s :: \emptyset$. Process 1 can be typed with $s : 1!\langle \mathsf{nat}\rangle; 1!\langle \mathsf{bool}\rangle; \mathsf{end}@\mathtt{p}$, whereas process 2 by $s : 1?\langle \mathsf{nat}\rangle; 1?\langle \mathsf{bool}\rangle; \mathsf{end}@\mathtt{q}$. After a reduction, (10) changes into:

$$s!\langle \mathsf{true}\rangle; \mathbf{0} \ \mid\ s :: 3 \ \mid\ s?(x); s?(y); \mathbf{0}. \tag{11}$$

Note that Equation (11) is identical to Equation (10) except that an output prefix in Equation (10) changes its place to the queue. Thus, we can go back from Equation (11) to Equation (10) by placing this message on the top of the process. A key idea in our runtime typing is *to carry out this "rollback of a message" in typing* using an endpoint type with a hole (a type context) for typing a queue. For example, we type the queue in Equation (11) as:

$$s : \{\ 1!\langle \mathsf{nat}\rangle; [\ ]@\mathtt{p},\ [\ ]@\mathtt{q}\ \}, \tag{12}$$

where [ ] indicates a hole (this will be formalized in Definition 5.2). Each of the holes should be filled by the remaining endpoint type of s at $\mathtt{p}$ and $\mathtt{q}$. Hence, we cover the

type $1!\langle \mathsf{bool}\rangle;\mathsf{end}$ with the type context for p just given, $1!\langle \mathsf{nat}\rangle;[\ ]$, obtaining the type $1!\langle \mathsf{nat}\rangle;1!\langle \mathsf{bool}\rangle;\mathsf{end}$ for p, restoring the original typing.

Labels in a queue are also typed using a type context. For example, $k : l_1 \cdot \mathsf{true} \cdot l_2$ can be typed with

$$k \oplus l_1 :\ k!\langle \mathsf{bool}\rangle;\ k \oplus l_2 : [\,], \tag{13}$$

omitting braces for a singleton selection. Now consider reduction

$$s_i \lhd \mathsf{ok}; P \mid s_i : \emptyset \quad \rightarrow \quad P \mid s_i : \mathsf{ok}. \tag{14}$$

Assume we type the left-hand side as

$$\tilde{s} : k \oplus \{\mathsf{ok} : T, \mathsf{quit} : T'\}@\mathrm{p}. \tag{15}$$

After the reduction, we obtain the type for P as

$$\tilde{s} : T@\mathrm{p}. \tag{16}$$

and the type for the queue as:

$$\tilde{s} : k \oplus \{\mathsf{ok} : [\ ]\}@\mathrm{p}. \tag{17}$$

By combining Equations (16) and (17) as before, we obtain

$$\tilde{s} : k \oplus \{\mathsf{ok} : T\}@\mathrm{p}. \tag{18}$$

We now observe that the located type in Equation (18) is a *subtype* of the located type in Equation (15) in the standard session subtyping [Gay and Hole 2005; Carbone et al. 2007, 2012], which is formally defined as [Pierce and Sangiorgi 1996]:

Definition 5.1. The *subtyping* over endpoint types, denoted $\leq_{\mathsf{sub}}$, is the maximal fixed point of function $\mathcal{F}$ that maps each binary relation $\mathcal{R}$ on endpoint types as regular trees to $F(\mathcal{R})$ given as:

—if $(T, T') \in \mathcal{R}$, then $(k!\langle U\rangle;T, k!\langle U\rangle;T') \in \mathcal{F}(\mathcal{R})$ and $(k?\langle U\rangle;T, k?\langle U\rangle;T') \in \mathcal{F}(\mathcal{R})$
—if $(T_i, T_i') \in \mathcal{R}$ for each $i \in I \subset J$, then $(\oplus\{l_i : T_i\}_{i\in I}, \oplus\{l_j : T_j'\}_{j\in J}) \in \mathcal{F}(\mathcal{R})$ and $(\&\{l_j : T_j\}_{i\in J}, \&\{l_i : T_i'\}_{i\in I}) \in \mathcal{F}(\mathcal{R})$.

If $T \leq_{\mathsf{sub}} T'$, then T is a *subtype* of T' whereas T' is a *supertype* of T.

Note that we do not have a subsumption rule for a program in Figure 7. On the other hand, we require a subtying relation between located types to type runtime processes.

Since $k \oplus \{\mathsf{ok} : T\} \leq_{\mathsf{sub}} k \oplus \{\mathsf{ok} : T, \mathsf{quit} : T'\}$, we can type the reductum in Equation (14) using the located type given in Equation (15), which is a supertype of the located type in Equation (18) through the standard subsumption, thus achieving the required rollback.

Type Contexts. Here, we formalize the notion of type context used in the previous section.

Definition 5.2. The type contexts $(\mathcal{T}, \mathcal{T}', \ldots)$ and the extended session typing $(\Delta, \Delta', \ldots$ as before) are given as:

$$\begin{array}{lcl} \mathcal{T} & ::= & [\ ] \mid k!\langle U\rangle;\mathcal{T} \mid k \oplus l_i : \mathcal{T} \\ H & ::= & T \mid \mathcal{T} \\ \Delta & ::= & \emptyset \mid \Delta, \tilde{s} : \{H_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p}\in I}. \end{array}$$

Thus, a type context represents a sequence of outputs and singleton selections that ends with a hole. As before, the notation "Δ, Δ'" denotes the union, assuming the domains should not include a common channel name. The *isomorphism* $\approx$ *on type contexts* is generated from permutations given here:

Definition 5.3 (*Permutation*). In addition to the folding/unfolding of recursive types, we consider endpoint types up to the following isomorphism (closed under all type constructors):

$$k!\langle U\rangle; k'!\langle U'\rangle; T \;\approx\; k'!\langle U'\rangle; k!\langle U\rangle; T \qquad (k \neq k') \tag{19}$$

$$k \oplus \{l_i : k' \oplus \{l'_j : T_{ij}\}_{j\in J}\}_{i\in I} \;\approx\; k' \oplus \{l'_j : k \oplus \{l_i : T_{ij}\}_{i\in I}\}_{j\in J} \qquad (k \neq k') \tag{20}$$

$$k \oplus \{l_i : k'!\langle U\rangle; T_i\}_{i\in I} \;\approx\; k'!\langle U\rangle; k \oplus \{l_i : T_i\}_{i\in I} \qquad (k \neq k') \tag{21}$$

The equations permute two consecutive outputs or selections with different subjects, capturing asynchrony in communication.

Assignments in Δ may contain both endpoint types and type contexts. Here, we define the partial commutative algebra $\circ$ where $\mathsf{sid}(\mathcal{T})$ are the channel numbers in $\mathcal{T}$:

$$T \circ \mathcal{T} = \mathcal{T} \circ T = \mathcal{T}[T]$$
$$\mathcal{T} \circ \mathcal{T}' = \mathcal{T}[\mathcal{T}'] \quad (\mathsf{sid}(\mathcal{T}) \cap \mathsf{sid}(\mathcal{T}') = \emptyset).$$

In the first rule, we place the output types of message queues on that of a process. In the second, we compose the type contexts for two sets of messages from the mutually disjoint sets of queues. Note $\mathcal{T} \circ \mathcal{T}'$ is defined if and only if $\mathcal{T}' \circ \mathcal{T}$ is defined and in which case we have $\mathcal{T}[\mathcal{T}'] \approx \mathcal{T}'[\mathcal{T}]$. Note also that $T \circ T'$ is never defined.

Here, we define a simple algebra of environments for runtime processes:

Definition 5.4 (*Type Algebra*). A partial operator $\circ$ is defined as:

$$\{H_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p}\in I} \circ \{H'_{\mathrm{p}'}@\mathrm{p}'\}_{\mathrm{p}'\in J} \;=\; \{(H_{\mathrm{p}} \circ H'_{\mathrm{p}})@\mathrm{p}\}_{\mathrm{p}\in I\cap J} \cup \{H_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p}\in I\setminus J} \cup \{H'_{\mathrm{p}'}@\mathrm{p}'\}_{\mathrm{p}'\in J\setminus I},$$

assuming each $\circ$ on the right-hand side is defined. Otherwise the operation is undefined. Then we say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if for all $\tilde{s}_i \in \mathsf{dom}(\Delta_i)$ such that $\tilde{s}_1 \cap \tilde{s}_2 \neq \emptyset$, $\tilde{s} = \tilde{s}_1 = \tilde{s}_2$ and $\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s})$ 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:

$$\Delta_1 \circ \Delta_2 \;=\; \{\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) \mid \tilde{s} \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \cup \Delta_1 \setminus \mathsf{dom}(\Delta_2) \cup \Delta_2 \setminus \mathsf{dom}(\Delta_1).$$

The operation $\Delta \circ \Delta'$ is undefined if $\Delta \asymp \Delta'$ does not hold.

5.2. Typing Rules for Runtime

To guarantee that there is at most one queue for each channel, we use the typing judgment refined as:

$$\Gamma \vdash P \triangleright_{\tilde{s}} \Delta,$$

where $\tilde{s}$ (regarded as a set) records the session channels associated with the message queues. The typing rules for runtime are given in Figure 8. [Subs] allows subsumption ($\leq_{\mathsf{sub}}$ is extended pointwise from types). [QNil] starts from the empty hole for each participant, recording the session channel in the judgment. [QVal] says that when we enqueue $\tilde{v}$, the type for $\tilde{v}$ is added at the tail. [QSess] and [QSel] are the corresponding rules for delegated channels and a label.

[Inact] allows weakening for empty queue types, whereas [Conc] is refined to prohibit duplicated message queues. The rule does not use coherence (cf. Def. 4.2 (2)) since coherence is meaningful only when all participants and queues are ready.

In [CRes], since we are hiding session channels, we now know no other participants can be added. Hence, we check that all message queues are composed and the given configuration at $\tilde{s}$ is coherent.

For the rest, we refine the original typing rules in Figure 7 not appearing in Figure 8 as follows (the full typing rules are listed in Appendix B):

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'} \qquad \frac{\Delta \text{ end only}}{\Gamma \vdash s[k]:\emptyset \ \triangleright_{s[k]} \ \tilde{s}:\{[\,]@\mathtt{p}\}_{\mathtt{p}} \circ \Delta} \qquad [\textsc{Subs}],[\textsc{QNil}]$$

$$\frac{\Gamma \vdash v_i : S_i \quad \Gamma \vdash s[k]:\tilde{h} \triangleright_{s[k]} \Delta, \tilde{s}:(\{\mathcal{T}@\mathtt{q}\} \cup R) \qquad R = \{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I}}{\Gamma \vdash s[k] : \tilde{h} \cdot \tilde{v} \ \triangleright_{s[k]} \ \Delta, \tilde{s}:(\{\mathcal{T}[k!\langle \tilde{S}\rangle;[\,]]@\mathtt{q}\} \cup R)} \qquad [\textsc{QVal}]$$

$$\frac{\Gamma \vdash s[k]:\tilde{h} \triangleright_{s[k]} \Delta, \tilde{s}:\{\mathcal{T}@\mathtt{q}\} \cup R \qquad R = \{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I}}{\Gamma \vdash s[k]:\tilde{h} \cdot \tilde{t}' \triangleright_{s[k]} \Delta, \tilde{s}:(\{\mathcal{T}[k!\langle T'@\mathtt{p}'\rangle;[\,]]@\mathtt{q}\} \cup R), \tilde{t}':T'@\mathtt{p}'} \qquad [\textsc{QSess}]$$

$$\frac{\Gamma \vdash s[k] : \tilde{h} \triangleright_{s[k]} \Delta, \tilde{s} : \{\mathcal{T}@\mathtt{q}\} \cup R \qquad R = \{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I}}{\Gamma \vdash s[k] : \tilde{h} \cdot l \ \triangleright_{s[k]} \ \Delta, \tilde{s}:(\{\mathcal{T}[k \oplus l : [\,]]@\mathtt{q}\} \cup R)} \qquad [\textsc{QSel}]$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta \quad \Gamma \vdash Q \triangleright_{\tilde{t}_2} \Delta' \quad \tilde{t}_1 \frown \tilde{t}_2 = \emptyset \quad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \Delta \circ \Delta'} \qquad [\textsc{Conc}]$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s}:\{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I} \quad \tilde{s} \in \tilde{t} \quad \{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I} \text{ coherent}}{\Gamma \vdash \ (\boldsymbol{\nu}\, \tilde{s})P \triangleright_{\tilde{t}\setminus\tilde{s}} \Delta} \qquad [\textsc{CRes}]$$

Fig. 8. Selected typing rules for runtime processes.

—For [Mcast], [Macc], [Rcv], [SRec], [Branch], and [Def], we replace $\Gamma \vdash P \triangleright \Delta$ with $\Gamma \vdash P \triangleright_{\emptyset} \Delta$.
—[Var] is similar to [Inact] (so that a queue can never occur in processes realizing participants).
—For both [Def] and [NRes], we replace $\Gamma \vdash P \triangleright \Delta$ by $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$.

Using these typing rules, we can check that the configurations at the beginning of this section, in Equations (10) and (11), are given an identical typing by "rolling back" the type of the message in the queues; similarly, for the next redex and reductum pair in the same page, Equations (15) and (16).

The typability in the original system in Section 4 and the one in this system coincide for processes without runtime elements.

Proposition 5.5. *Let P be a program phrase and Δ be without a type context. Then, $\Gamma \vdash P \triangleright \Delta$ in the typing system in Section 4 if and only if $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ is derived without using [*Subs*] in the typing system in this section.*

Proof. See Appendix B. □

Proposition 5.6. *If $\Gamma \vdash P \triangleright_{s[1..m]} \Delta$, then P has a unique queue at $s[i]$ $(1 \leq i \leq m)$, no other queue at a free channel occurs in P, and no queue in P is under any prefix.*

Proof. It is routine by rule induction, see Appendix B.2. □

5.3. Type Reduction

Next, we introduce a reduction relation over session typings, which abstractly represents interaction in processes at session channels. Here, we assume well-formedness of types and typing.

Definition 5.7 (*Type Reduction*). The syntax of labels $(\ell, \ell', \ldots)$ of local types is defined as follows:

$$\ell \ ::= \ \mathtt{p} \to \mathtt{p}' : k\langle U\rangle \ \mid \ \mathtt{p} \to \mathtt{p}' : k\langle l\rangle \ \mid \ \mathtt{p} \to \mathtt{p}' : s[k]\langle U\rangle \ \mid \ \mathtt{p} \to \mathtt{p}' : s[k]\langle l\rangle.$$

We generate $\Delta \xrightarrow{\ell} \Delta'$ by the following rule:

$$k!\langle U\rangle; H@\mathtt{p},\ k?\langle U\rangle; T@\mathtt{q} \xrightarrow{\mathtt{p}\to\mathtt{q}:k\langle U\rangle} H@\mathtt{p},\ T@\mathtt{q} \qquad \text{[TR-Com]}$$

$$k \oplus \{l : H, \ldots\}@\mathtt{p},\ k\&\{l : T, \ldots\}@\mathtt{q} \xrightarrow{\mathtt{p}\to\mathtt{q}:k\langle l\rangle} H@\mathtt{p},\ T@\mathtt{q} \qquad \text{[TR-Bra]}$$

$$\frac{H_1@\mathtt{p}_1,\ H_2@\mathtt{p}_2 \xrightarrow{\ell} H_1'@\mathtt{p}_1,\ H_2'@\mathtt{p}_2 \quad \mathtt{p}_1, \mathtt{p}_2 \in I \quad k \in \ell}{\tilde{s} : \{H_1@\mathtt{p}_1,\ H_2@\mathtt{p}_2, \ldots\}_{i\in I}, \Delta \xrightarrow{\ell[s[k]/k]} \tilde{s} : \{H_1'@\mathtt{p}_1,\ H_2'@\mathtt{p}_2, \ldots\}_{i\in I}, \Delta} \qquad \text{[TR-Context]}$$

$$\frac{\Delta \approx \Delta_0 \quad \Delta_0 \xrightarrow{\ell} \Delta_0' \quad \Delta_0' \approx \Delta'}{\Delta \xrightarrow{\ell} \Delta'} \qquad \text{[TR-Iso]}$$

In the sequel, we investigate the relationship between the LTS semantics of global and local types to prove the key properties for the main theorems.

Definition 5.8 (*Full Projection*). Assume G is coherent. Then the *full projection of* G, denoted by $[\![G]\!]$, is defined as the set $\{(G \upharpoonright \mathtt{p})@\mathtt{p} \mid \mathtt{p} \in \mathsf{pid}(G)\}$. We write $[\![G]\!] \xrightarrow{\ell} [\![G']\!]$ if $\tilde{s} : [\![G]\!] \xrightarrow{\ell[s[k]/k]} \tilde{s} : [\![G']\!]$.

Definition 5.9 (*Coherence and Partial Coherence of Typings*). (1) We say Δ *is coherent* if $\Delta(\tilde{s})$ is coherent for each $\tilde{s} \in \mathsf{dom}(\Delta)$. (2) Δ is *partially coherent* if for some Δ' we have $\Delta \asymp \Delta'$, and $\Delta \circ \Delta'$ is coherent.

The following lemma states that for any type reduction in the local types projected from G, its corresponding action $\mathtt{p}_i \to \mathtt{p}_j : k$ in G is the minimum with respect to $\prec_\phi$.

Lemma 5.10 (Projection and Causality). *Assume* $[\![G]\!] = \{T_i@\mathtt{p}_i\}_{i\in I}$ *and there exists* $i, j \in I$ *such that* $T_i@\mathtt{p}_i, T_j@\mathtt{p}_j \xrightarrow{\ell} T_i'@\mathtt{p}_i, T_j'@\mathtt{p}_j$ *with* $k \in \ell$. *Then there is no action* $\mathtt{q} \to \mathtt{q}' : k' \in G$ *such that* $(\mathtt{q} \to \mathtt{q}' : k') \prec_\phi (\mathtt{p}_i \to \mathtt{p}_j : k) \in G$ *with either (i)* $\phi \in \{\mathsf{II}, \mathsf{IO}\}$ *or (ii)* $\phi = \mathsf{OO}$ *and* $k = k'$.

Proof. By the linearity of G, if T_i is the output type at k, then there is no output type at k except T_i. Similarly if T_j is an input type at k, there is no input type at k except T_j in $[\![G]\!]$. Then it is obvious by the definition of $\prec$. □

The key lemma that states the one-to-one correspondence between the semantics of a global type and the semantics of its projected local types follows.

Lemma 5.11 (Global and Local Types). *Suppose* G *is coherent. Then* $G \xrightarrow{\ell} G'$ *if and only if* $[\![G]\!] \xrightarrow{\ell} [\![G']\!]$.

Proof. The only-if direction is straightforward by definition of $[\![G]\!]$. We prove the if direction by induction on the derivation of $[\![G]\!] \xrightarrow{\ell} [\![G']\!]$.

Let us first analyze the case where either [TR-Com] or [TR-Bra] are in the premise of [TR-Context]. If $\mathtt{p}_1$ and $\mathtt{p}_2$ are top level in G, then we can straightforwardly use [GR1] and [GR2] from Definition 3.2. Otherwise, if [TR-Bra] is in the premise, then it must be the case that both $\mathtt{p}_1$ and $\mathtt{p}_2$ are not top level in G. This follows by the definition of projection and applicability of [TR-Context] with [TR-Iso] not in the premise. In such a case, by definition of projection, all roles different from $\mathtt{p}_1$ and $\mathtt{p}_2$ must behave the same on each branch. Hence, the precondition of [GR4] is satisfied, and we can apply such rule. Note that the global type obtained after reduction can be projected to the

reductum of [TR-Context] as expected. The case where [T-Com] is used in the premise of [TR-Context] is similar.

If [TR-Iso] is in the premise of [TR-Context], then we must have done a permutation of some outputs/selections. We show that such a behavior can be emulated by the global type semantics. Suppose that

$$\tilde{s} : [\![G]\!] = \tilde{s} : \{k!\langle U\rangle; k'!\langle U'\rangle; T_1@1, k?\langle U\rangle; T_2@2, k'?\langle U'\rangle; T_3@3\}$$
$$\xrightarrow{\ell_0} \tilde{s} : [\![G']\!] = \tilde{s} : \{k'!\langle U'\rangle; T_1@1, T_2@2, k'?\langle U'\rangle; T_3@3\}$$

where $G = 1 \to 2 : k\langle U\rangle.1 \to 3 : k'\langle U'\rangle.G_1$ and $G' = 1 \to 3 : k'\langle U'\rangle.G_1$ with $\ell_0[s[k]/k] = \ell$.

Now, suppose

$$\tilde{s} : [\![G]\!] \approx \tilde{s} : \{k'!\langle U'\rangle; k!\langle U\rangle; T_1@1, k?\langle U\rangle; T_2@2, k'?\langle U'\rangle; T_3@3\}$$
$$\xrightarrow{\ell'_0} \tilde{s} : [\![G_0]\!] = \tilde{s} : \{k!\langle U\rangle; T_1@1, k?\langle U\rangle; T_2@2, T_3@3\}$$

by Equation (19) in Definition 5.3, where $G_0 = 1 \to 2 : k\langle U\rangle.G_1$. By the definition of LTS ([GR3] in Definition 3.2), we can obtain $G_1 \xrightarrow{\ell'} G_0$ with $\ell'_0[s[k]/k] = \ell'$, as required. Other cases are similar. □

The following lemma states that the transitions from global types and projected local types are deterministic.

Lemma 5.12 (Determinacy).

(1) *Suppose G is coherent. Then $G \xrightarrow{\ell} G_1$ and $G \xrightarrow{\ell} G_2$ imply $G_1 \approx G_2$.*
(2) *Suppose Δ is coherent. Then $\Delta \xrightarrow{\ell} \Delta_1$ and $\Delta \xrightarrow{\ell} \Delta_2$ imply $\Delta_1 \approx \Delta_2$.*
(3) *Suppose G is coherent, and $G \xrightarrow{\ell_1} G_1$ and $G \xrightarrow{\ell_2} G_2$ with $k \in \ell_1, \ell_2$. Then $\ell_1 = \ell_2$.*
(4) *Suppose Δ is coherent, and $\Delta \xrightarrow{\ell_1} \Delta_1$ and $\Delta \xrightarrow{\ell_2} \Delta_2$ with $s[k] \in \ell_1, \ell_2$. Then $\ell_1 = \ell_2$.*

Proof. Condition (1) is immediate, noting that if $G_1 \mid G_2 \xrightarrow{\ell} G'_1 \mid G_2$ then $G_2 \overset{\ell}{\not\to}$ (since the participants are disjoint between G_1 and G_2). Condition (2) is by Condition (1) and Lemma 5.10. Conditions (3) and (4) are similar to Conditions (1) and (2), respectively. □

The following proposition states that (Condition 1) transitions of Δ are closed under $\asymp$; (Conditions 2, 3) Δ is invariant with regard to partial and coherence; and (Condition 4), the transition of a global type and its mapping have exact correspondence.

Proposition 5.13.

(1) *$\Delta_1 \xrightarrow{\ell} \Delta'_1$ and $\Delta_1 \asymp \Delta_2$ imply $\Delta'_1 \asymp \Delta_2$ and $\Delta_1 \circ \Delta_2 \xrightarrow{\ell} \Delta'_1 \circ \Delta_2$.*
(2) *Let Δ be coherent. Then $\Delta \xrightarrow{\ell} \Delta'$ implies Δ' is coherent.*
(3) *Let Δ be partial coherent. Then $\Delta \xrightarrow{\ell} \Delta'$ implies Δ' is partial coherent.*
(4) *Let Δ be coherent and $\Delta(\tilde{s}) = [\![G]\!]$. Then $\Delta \xrightarrow{\ell} \Delta'$ with $s[k] \in \ell$ iff $G \xrightarrow{\ell[k/s[k]]} G'$ with $\Delta'(\tilde{s}) = [\![G']\!]$.*

Proof. For Condition (1), suppose $\Delta_1 \xrightarrow{\ell} \Delta'_1$ with $s[k] \in \ell$ and $\Delta_1 \asymp \Delta_2$. Note by definition of $\Delta_1 \asymp \Delta_2$, each pair of vectors of channels from $\Delta_{1,2}$ either coincide or are disjoint; that is, (a) $s[k] \in \tilde{s} \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$ or (b) $s[k] \in \tilde{s} \in \mathsf{dom}(\Delta_i)$ and $\tilde{s} \notin \mathsf{dom}(\Delta_j)$ with $i \neq j$. For case (a), since the typed reduction only erases the top input

and output pair in Δ_1, we have $\Delta_1' \asymp \Delta_2$ by the inductive hypothesis and Lemma 5.12 (Condition 2). Then $\Delta_1 \circ \Delta_2 \xrightarrow{\ell} \Delta_1' \circ \Delta_2$ is by definition.

Case (b) is vacuous since the reduction does not relate to the domain of Δ_2. Hence $\Delta_1' \asymp \Delta_2$.

For Condition (2), suppose Δ is coherent and $\Delta \xrightarrow{\ell} \Delta'$. Suppose the associated redex is in $\Delta(\tilde{s})$. By coherence, we can write $\Delta(\tilde{s})$ as $[\![G]\!]$ for some coherent G. By Lemma 5.11, there exists G' such $G \xrightarrow{\ell'} G'$ such that $\ell'[s[k]/k] = \ell$ and $[\![G']\!] = \Delta'(\tilde{s})$. Then, by Proposition 4.4, G' is coherent. Hence, $[\![G']\!]$ and $\Delta'(\tilde{s})$ are both coherent.

Implication (3) is immediate from Conditions (1) and (2).

Finally the only if-direction of Condition (4) follows directly from Definition 5.8, whereas the if direction is immediate by Lemma 5.11. □

5.4. Subject Reduction and Communication Safety

For subject reduction, we use the following lemmas. In Lemma 5.14, we say that two typings, Δ_1 and Δ_2, *share a common target channel in their type contexts* when, for some $\tilde{s}$ and k, we have: (1) $\mathcal{T}_1@\mathrm{p} \in \Delta_1(\tilde{s})$ and $\mathcal{T}_2@\mathrm{p} \in \Delta_2(\tilde{s})$; and (2) $k!\langle U\rangle$ or $k \oplus l$ occurs in $\mathcal{T}_1$ and $k!\langle U'\rangle$ or $k \oplus l'$ occurs in $\mathcal{T}_2$ (i.e., they have an output/selection type at a shared channel).[4]

Lemma 5.14 (Partial Commutativity and Associativity of $\circ$). *$\circ$ on typings is partially commutative and associative with identity $\emptyset$ under the condition that, whenever we compose two typings, they never share a target channel in their type contexts (in the above sense).*

Proof. See Appendix B.3. □

Lemma 5.15. *Assume $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. Then, all free names and free variables in P occur in Γ, and all free channels in P occur in Δ.*

Here, a *derivation of* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ is a derivation tree of the typing rules for runtime processes (fully listed in Appendix B) whose conclusion is $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$.

Lemma 5.16 (Permutation). (1) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [*Subs*] at its last two steps. Then, $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ has a derivation identical with the original one except its last two steps are replaced by a single application of [*Subs*].* (2) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [*Subs*] as its last rule and another rule that is not one of [*Subs*], [*Sel*], and [*Branch*]. Then ,$\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ has a derivation that is identical to the original one except that the last two rules used are permuted.*

Proof. Condition (1) is immediate from the transitivity of [Subs]. Condition (2) is routine. □

Lemma 5.17 (Queue). *The following rules are admissible in the typing system for runtime processes. Here, let $\tilde{s} = s[1..k..n]$ and let us assume that occurrences of $\circ$ in the*

[4]Whenever we compose two processes, their typings never share a common target channel in their type contexts in this sense because, by the disjointness of mentioned channels for queues, target channels in type contexts can never coincide.

premise of each rule are well-defined.

$$\frac{\Gamma \vdash s[k] :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\} \quad \Gamma \vdash \tilde{v} \triangleright \tilde{S}}{\Gamma \vdash s[k] :: \tilde{h} \cdot \tilde{v} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}[k!\langle \tilde{S}\rangle; [\,]]@\mathrm{p}\}} \quad [\mathrm{Q}\textsc{val}]$$

$$\frac{\Gamma \vdash s[k] :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\} \quad \{\tilde{t}\}\ \mathit{fresh}}{\Gamma \vdash s :: \tilde{h} \cdot \tilde{t} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}[k!\langle T@\mathrm{p}'\rangle; [\,]]@\mathrm{p}\},\ \tilde{t} : \{T@\mathrm{p}'\}} \quad [\mathrm{Q}\textsc{sess}]$$

$$\frac{\Gamma \vdash s :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\}}{\Gamma \vdash s :: \tilde{h} \cdot l \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}[k \oplus \{.., l : [\,], ..\}]@\mathrm{p}\}} \quad [\mathrm{Q}\textsc{sel}]$$

$$\frac{\Gamma \vdash s :: \tilde{v} \cdot \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k!\langle \tilde{S}\rangle; \mathcal{T}@\mathrm{p}\}@\mathrm{p}}{\Gamma \vdash s :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\}} \quad [\mathrm{Q}\textsc{valdq}]$$

$$\frac{\Gamma \vdash s :: \tilde{t} \cdot \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k!\langle T@\mathrm{p}'\rangle; \mathcal{T}@\mathrm{p}\},\ \tilde{t} : \{T@\mathrm{p}'\}@\mathrm{p}'}{\Gamma \vdash s :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\}} \quad [\mathrm{Q}\textsc{sessdq}]$$

$$\frac{\Gamma \vdash s :: l \cdot \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k \oplus l : \mathcal{T}@\mathrm{p}\}}{\Gamma \vdash s :: \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\mathcal{T}@\mathrm{p}\}} \quad [\mathrm{Q}\textsc{seldq}]$$

Proof. See Appendix B.2. □

Here, we do not require the substitution lemmas for session channels and process variables (cf. Yoshida and Vasconcelos [2007]).

Lemma 5.18 (Substitution and Weakening). (1) *(substitution)* $\Gamma, x : S \vdash P \triangleright_{\tilde{s}} \Delta$ *and* $\Gamma \vdash v : S$ *imply* $\Gamma \vdash P[v/x] \triangleright_{\tilde{s}} \Delta$. (2) *(weakening) Whenever* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *is derivable, then its weakening,* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta, \Delta'$ *for disjoint* Δ', *where* Δ' *contains only empty type contexts and for types* end, *is also derivable.*

Proof. Standard, see Yoshida and Vasconcelos [2007]. □

Among the preceding lemmas, the lemmas for queues are needed for treating reduction involving queues in the present asynchronous operational semantics. We can now establish subject reduction.

Subject Reduction, Communication Safety, and Session Fidelity. By the preceding proposition and the substitution lemma, we obtain:

Theorem 5.19 (Subject Congruence and Reduction).

(*1*) $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *and* $P \equiv P'$ *imply* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta$.
(*2*) $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *such that* Δ *is coherent and* $P \to P'$ *imply* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ *where* $\Delta = \Delta'$ *or* $\Delta \xrightarrow{\ell} \Delta'$ *for some* ℓ.
(*3*) $\Gamma \vdash P \triangleright_{\emptyset} \emptyset$ *and* $P \to P'$ *imply* $\Gamma \vdash P' \triangleright_{\emptyset} \emptyset$.

Proof. See Appendix B.4. □

Remark 5.20. Theorem 5.19, Condition (3) and the subsequent results (in particular Theorem 5.22 and Corollary 5.30) tell us, through Proposition 5.5, that the typing system in Section 4, which is for programs and program phrases, guarantees type safety and other significant behavioral properties for typable programs, noting that typability of (annotated) programs is decidable by Proposition 4.6.

Theorem 5.19 immediately entails the lack of the standard type errors in expressions (such as $\mathsf{true} + 3$). The type discipline also satisfies, as in the preceding session type disciplines [Honda et al. 1998], communication error freedom, including linear usage of channels. We first introduce the reduction context $\mathcal{E}$ as follows:

$$\mathcal{E} \quad ::= \quad \mathcal{E} \mid P \quad | \quad P \mid \mathcal{E} \quad | \quad (\nu n)\mathcal{E} \quad | \quad \mathsf{def}\ D\ \mathsf{in}\ \mathcal{E}$$

We also state:

—A prefix is *at* s (*at* a, respectively) if its subject (i.e., its initial channel) is s (a, respectively). Furthermore, a prefix is *emitting* if it is request, output, delegation, or selection; otherwise it is *receiving*.
—A prefix is *active* if it is not under a prefix or an `if` branch, after any unfoldings by [Def]. We write $P\langle\!\langle s \rangle\!\rangle$ if P contains an active subject at s after applying [Def], and we write $P\langle\!\langle s! \rangle\!\rangle$ (resp. $P\langle\!\langle s? \rangle\!\rangle$) if P contains an emitting (receiving, respectively) active prefix at s.
—P has a *redex at* s if it has an active prefix at s among its redexes.

Here and henceforth, we safely confuse a channel (as a number) in a typing and the corresponding free session channel of a process.

Lemma 5.21. *Assume* $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ *s.t.* $\Delta \circ \Delta_0$ *is coherent for some* Δ_0.

(*1*) *If* $P\langle\!\langle s \rangle\!\rangle$, *then* P *contains either a unique active prefix at* s *or a unique active emitting prefix and a unique active receiving prefix at* s.
(*2*) *If* P *contains an active emitting (receiving, respectively) prefix at* s, *then* Δ *contains an emitting (receiving, respectively) minimal prefix at* s.

Proof. By easy rule induction, see Appendix B.6. □

The following result adapts the standard properties for synchronous session types [Takeuchi et al. 1994; Honda et al. 1998; Yoshida and Vasconcelos 2007] to multiparty asynchronous session types. Note that reductions may go wrong for several reasons. Traditional problems include non-boolean values in a conditional, as in $\mathsf{if}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q$, and arity mismatch for process definitions as in $\mathsf{def}\ X(yx) = P\ \mathsf{in}\ X\langle\mathsf{true}\rangle$. Here, we are instead interested in *communication safety*, which ensures there is no error when participants interact with each other. Since interactions always happen at session channels, we focus on the linearity property (no races) and the interactions between processes and their corresponding queue. Here, we assume the standard bound name convention.

Theorem 5.22 (Communication Safety). *Suppose* $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ *s.t.* Δ *is coherent and* P *has a redex at free* s. *Then:*

(*1*) *(linearity)* $P \equiv \mathcal{E}[s :: \tilde{h}]$ *such that either*
  (*a*) $P\langle\!\langle s? \rangle\!\rangle$, s *occurs exactly once in* $\mathcal{E}$ *and* $\tilde{h} \neq \emptyset$; *or*
  (*b*) $P\langle\!\langle s! \rangle\!\rangle$ *and* s *occurs exactly once in* $\mathcal{E}$; *or*
  (*c*) $P\langle\!\langle s? \rangle\!\rangle$, $P\langle\!\langle s! \rangle\!\rangle$, *and* s *occurs exactly twice in* $\mathcal{E}$.
(*2*) *(error-freedom) if* $P \equiv \mathcal{E}[R]$ *with* $R\langle\!\langle s? \rangle\!\rangle$ *being a redex:*
  (*a*) *If* $R \equiv s?(\tilde{y}); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{v} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{v}| = |\tilde{y}|$.
  (*b*) *If* $R \equiv s?(\!(\tilde{s})\!); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{t} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{s}| = |\tilde{t}|$.
  (*c*) *If* $R \equiv s \triangleright \{l_i : Q_i\}_{i \in I}$ *then* $P \equiv \mathcal{E}'[s : l_j \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $j \in I$.

Proof. For Condition (1), let $P \equiv (\nu \tilde{n})(P_0 | s : \tilde{h} | Q)$ where P_0 does not contain a queue and Q only contains queues (by Proposition 5.6). By Lemma 5.21, we know P_0 has either a single active prefix or a pair of a receiving active prefix and an emitting active prefix. So we have three cases:

—$P_0\langle\langle s?\rangle\rangle$ and there is no other active prefixes at s: If so because there is a redex in P the queue cannot be empty.
—$P_0\langle\langle s!\rangle\rangle$ and there is no other active prefixes at s: Then this gives us a redex.
—$P_0\langle\langle s!\rangle\rangle$ and $P_0\langle\langle s?\rangle\rangle$. Then at least the former gives a redex, but the latter can also give a redex.

Hence, as required.

For Condition (2), if P satisfies the stated condition, then we can write $P \equiv \mathcal{E}'[s : \tilde{h}|R]$ and $S \stackrel{\text{def}}{=} s : \tilde{h}|R$ form a redex, with the same typing by Theorem 5.7 Condition (1). Since this should have a partially coherent typing, it means, in particular, that the pair of active prefixes at s in the tying of S should be complementary. The rest is by the direct correspondence between the type constructors and the prefixes. □

By Theorems 5.19 and 5.22, a typed process "never goes wrong" in the sense that its interaction at a multiparty session channel is always one-to-one and that each delivered value matches the receiving prefix.

By Lemma 5.21 Condition (2) and by the typing of the associated queue, this delivery precisely corresponds to a redex in the session typing.

As the corollary of Theorem 5.19 Condition (2) and Proposition 5.13 Condition (4), we obtain *session fidelity*: The interactions of a typable process exactly follow the specification described by its global type.

COROLLARY 5.23 (SESSION FIDELITY). *Assume* $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ *such that* Δ *is coherent and* $\Delta(\tilde{s}) = [\![G]\!]$. *If*

(*1*) $P\langle\langle s[k]?\rangle\rangle \to P'$ *at the redex of* $s[k]$, *then* $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ *with* $G \stackrel{\ell}{\to} G'$ *with* $k \in \ell$ *and* $[\![G']\!] = \Delta'(\tilde{s})$, *or*
(*2*) $P\langle\langle s[k]!\rangle\rangle \to P'$ *at the redex of* $s[k]$, *then* $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta$.

PROOF. In Conditioin (1), the conclusion $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ where $\Delta = \Delta'$ or $\Delta \stackrel{\ell}{\to} \Delta'$ follows directly from Theorem 5.19 Condition (2). The second conclusion $G \stackrel{\ell}{\to} G'$ with $k \in \ell$ and $[\![G']\!] = \Delta'(\tilde{s})$ follow directly from Lemma 5.12 Condition (3) and Proposition 5.13 Condition (4). If not, a sender puts some value in the queue. Hence, Condition (2) obviously holds. □

5.5. Progress

Communication safety says that if a process ever does a reduction, it conforms to the typing and it is linear. If interactions within a session are not hindered by initialization and communication of *different* sessions, then the converse holds: The reduction predicted by the typing surely takes place, that is the standard progress property in binary session types [Dezani-Ciancaglini et al. 2006; Honda et al. 1998]. First, we define:

Definition 5.24. Let $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. Then, P is *queue-full* when $\{\tilde{s}\}$ coincide with the set of session channels occurring in Δ.

A process is queue-full when it has a queue for each session channel. The following precludes interleaving of other sessions (including initializations and communications) that can introduce deadlock. For example, two session initializations $a[2](s).b[2](t).s?;t!$ and $\overline{a}[2](s).\overline{b}[2](t).t?;s!$ cause deadlock. Observe, because we have multiparty sessions, that there is less need to use interleaved sessions.

Definition 5.25 (*Simple*). A process P is *simple* when it is typable with a type derivation where the session typing in the premise and the conclusion of each prefix rule in Figure 7 is restricted to at most a singleton. That is, *(1)*Δ of [MCAST], [MACC],

[Send], [Rcv], [Branch] and [Var] are empty. *(2)* Neither [Rcv] nor [Deleg] is used. *(3)* Δ of [If], [Inact], [NRes] and [Def] contains at most a singleton, and, in [Conc], either Δ, Δ' contains at most a singleton.

Thus, each prefixed subterm in a simple process has only a unique session.

Proposition 5.26. *Let P_0 be simple and $P_0 \to^* P$. Then, no delegation prefix (input or output) occurs in P and for each prefix with a shared name in P, say $a[\mathtt{i}](\tilde{s}).P'$ or $\overline{a}[2..\mathtt{n}](\tilde{s}).P'$, there is no free session channels in P' except $\tilde{s}$.*

Proof. See Appendix B.7. □

Another element that can hinder progress is when interactions at shared names cannot proceed.

Definition 5.27 (*Well-Linked*). We say P is *well-linked* when for each $P \to^* Q$, whenever Q has an active prefix whose subject is a (free or bound) shared name, then it is always part of a redex.

Thus, in a simple well-linked P, each session is never hindered by other sessions nor by a name prefixing. The key lemma for simple processes follows. Here, we safely confuse a channel in a typing and the corresponding free session channel of a process.

Lemma 5.28. *Let $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ and P is simple. If there is an active receiving (active emitting, respectively) prefix in Δ at s and none of prefixes at s in P is under a prefix at a shared name or under an* `if`*-branch, then $P \langle\!\langle s? \rangle\!\rangle$ (either $P \langle\!\langle s! \rangle\!\rangle$ or the queue at s is not empty, respectively).*

Proof. By rule induction using Proposition 5.26, see Appendix B.8. □

Proposition 5.29. *Let $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$, Δ is coherent, P is simple, well-linked and queue-full. Then:*

(1) If $P \not\equiv \mathbf{0}$, then $P \to P'$ for some P'.

(2) If $\Delta(\tilde{t}) = [\![G]\!]$ and $G \xrightarrow{\ell} G'$ with $k \in \ell$, then $P \to^+ P'$ at the redex at t_k such that $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ with $\Delta'(\tilde{t}) = [\![G']\!]$.

Proof. Let P be simple, queue-full, and well-linked, and $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ such that Δ is coherent. Without loss of generality we can assume P does not have hidings (we can just take off and the result is again simple, queue-full, well-linked, and coherent). Since Δ is coherent, if Δ contains any prefix then, by, Proposition 5.26, it should form a redex (together with another prefix to form the image of an identical set). By Lemma 5.28 and Theorem 5.22 (Conditions 1 and 2) and by the well-linkedness, either there is an if-branch above the prefix or P has an active prefix (or prefixes) at s in P. For the former, this if-branch itself cannot be under any prefix since that violates the activeness at s in Δ. So this if-branch can reduce; hence, we conclude the case.

If not, then, by Lemma 5.28, there are the following cases:

(a) $P \equiv \mathcal{E}[Q\langle\!\langle s! \rangle\!\rangle | s : \tilde{h} | R\langle\!\langle s? \rangle\!\rangle]$, in which case there is at least one redex in P between the emitting prefix and the queue.
(b) $P \equiv \mathcal{E}[s : \tilde{h} | R\langle\!\langle s? \rangle\!\rangle]$ with $\tilde{h}$ non-empty, in which case there is a redex between the non-empty queue and the receiving redex.
(c) $P \equiv \mathcal{E}[Q\langle\!\langle s! \rangle\!\rangle | s : \tilde{h}]$, in which case there is a redex as in (a).

In each case, there is a reduction; hence, done. □

(2) above gives the converse of Corollary 5.23: if the global type has a reduction, then the process can always realize it.

COROLLARY 5.30 (PROGRESS). *Let P be a simple and well-linked program. Then P has the progress property in the sense that $P \to^* P'$ implies either $P' \equiv \mathbf{0}$ or $P' \to P''$ for some P''.*

PROOF. Immediate from Proposition 5.26, Lemma 5.28, and Theorem 5.29. □

A simple application of Theorems 5.19 Condition (3) and 5.22 and Corollary 5.30 for processes from Section 2.3 follow. Here, *communication mismatch* stands for the violation of the conditions given in Theorem 5.22 Condition (2).

PROPOSITION 5.31 (PROPERTIES OF TWO PROTOCOLS).

(*1*) *Let $Buyer1|Buyer2|Seller \to^* P$. Then P is well-typed, simple, and well-linked; P has no communication mismatch; and either $P \equiv \mathbf{0}$ or $P \to P'$ for some P'.*
(*2*) *Similarly for $DataProducer|KeyProducer|Kernel|Consumer$.*

PROOF. Immediate from Corollary 5.30 because these two configurations are typable programs each of which loses its shared name in the initial reduction (at which point all the occurrences of the shared name are used). □

The significance of the progress result under these constraints is that, if a typable program ever gets stuck during reduction, then its causes are other than the structure of individual typed conversations: Thus, we are ensured that the causes of deadlock (if any) in typed interactions do not lie in each conversation structure itself, allowing their well-articulated analysis.

6. EXTENSIONS AND RELATED WORK

We outline applications and several possible extensions of the presented framework, then discuss related works. We also summarize recent results and applications of multiparty session types after the publication of the extended abstract [Honda et al. 2008a].

6.1. Applications and Extensions

Applications. As we already discussed (cf. Section 1 and Section 4.1), the type discipline we explored in the present article is intended to be used as a typed foundation for the development of communication-centered software in various ways and at different development stages. Types will also serve as a core specification on which other formal specifications and techniques such as program analyses and assertions may be built.

A global type serves as an agreement of a protocol following which each end-point program will execute its communication. An automatic method to check well-formedness of the global types (linearity, by Proposition 3.13, and coherence, by Proposition 4.3) guarantees the behaviors specified by the global specification. Development of individual programs for endpoint communications, which materialize a global conversation, is assisted in several ways: first, the projection of a global type to each participant (well-defined by coherence) directly suggests the possible shape of end-point interactional behavior. Second, during development, a programmer can check whether her program conforms to the agreed global type through type-checking the program against an appropriate projection of the global type (Proposition 4.6). The global type and its projections may also be used as a basis of the debugging/testing process, including automatic generation of test suites.

Once the development of all programs is complete, their typability ensures, in the absence of systems errors (such as transport-level failure), that the runtime behavior of the deployed programs satisfies the key properties including communication safety, session fidelity, and progress through the theorems in Section 5 (cf. Theorems 5.19 and 5.22 and Corollary 5.30). Since global types and their projections specify possible

legitimate interaction sequences of the deployed programs, they can be used for runtime monitoring, flagging those communications that go out of expected conversation sequences and thus signaling the existence of system-level errors (which is another direct consequence of the theorems in Section 5), thus helping to locate the cause of such errors. These static and dynamic validations of programs may add further precision by using refined specifications, such as logical assertions following (global and local) session type structures, as we recently did in Bocchi et al. [2010].

The effectiveness of these applications hinges on the exactitude with which global types and the associated type discipline can assure basic properties of programs and thus is underpinned by the formal results discussed in this article. At the same time, to put these ideas into practise, the presented framework may need various extensions as well as engineering experiments. Some possible extensions of the presented type discipline are discussed in the following.

Existing Extensions of Binary Session Types. In the literature, several extensions of binary session type disciplines have been proposed, including subtyping [Gay and Hole 2005], bounded polymorphism [Gay 2008], integration with security annotations to guarantee authentication properties [Bonelli et al. 2005], and integration with higher order π-calculus [Mostrous and Yoshida 2007, 2009]. We believe that integrations with these extensions should be possible and will enrich the expressive power and applicability of the theory.

Multithreaded Participants. Another straightforward extension is to allow a multithreaded participant, so that a single participant can perform parallel conversations with others during a session. For this extension, we need to augment endpoint types with the parallel composition $T_1 \mid T_2$, equipped with the following isomorphism (using type contexts in Section 5): $\mathcal{T}[T_1] \mid T_2 \approx \mathcal{T}[T_1 \mid T_2]$ if for no k there is an output at k in both $\mathcal{T}$ and T_2 (such a prefix adds false OO-dependency), as well as commutativity and associativity. Linearity between T_1 and T_2 in T_1, T_2 is given by coherence via projection. This extension has been recently studied with more advanced dynamic roles in Deniélou and Yoshida [2011].

Graph-Based Global Types and Type Inference. The syntax of global types uses the standard abstract syntax tree. We can further generalize this tree-based syntax to graph structures to obtain a strictly more expressive type language, thus enlarging typability. Consider the two endpoint processes $P \equiv s!.t?$ and $Q \equiv t!.s?$: Their parallel composition does not introduce conflict, hence, it is linear and safe. This situation, however, cannot be represented in the current global types since two "prefixes" criss-cross each other. Interestingly, our linearity conditions in Section 3.5, based on input/output dependencies, can directly capture the safety of this configuration. All we need do is to take the graphs of prefixes and II, IO, and OO-edges (cf. Figure 5) under the linearity condition (precisely following Section 3.5) as global types, augmented with an acyclicity condition on chains of these causal edges. All other definitions and results stay the same.

Our recent work [Mostrous et al. 2009] studies the generation of graph-based global types from endpoint types, where we also use such graph-based types for solving the type inference problem for (the generalized version of) the presented type discipline. This is further extended in Deniélou and Yoshida [2012] and Lange et al. [2015], making a connection with Communicating Automata. See Section 6.3.

Synchrony and Asynchrony. Most of the session types currently studied are binary and synchronous [Honda et al. 1998]. In some computing environments (e.g., tightly coupled SMP), synchrony would be more suitable. Adding synchrony means we have more causality: OO-dependency between different names as well as the OI-dependency

(i.e., the dependency from output to input, cf. Figure 5), which in asynchrony never arises Section 3.4. Our subsequent work [Bejleri and Yoshida 2009] studies a synchronous multiparty session type.

A different direction is to consider asynchronous message passing without order preservation [Honda and Tokoro 1991], which is also used in some computing environments (although in many environments we have efficient order-preserving transports such as TCP). Again, we can use our modular articulation by taking off OO-edges to obtain a consistent theory for pure asynchrony.

Multicast Primitives for Sessions Communication. The two-buyer protocol uses a multicasting prefix notation $s, t!\langle V\rangle$. The present work treats it as a macro for $s!\langle V\rangle; t!\langle W\rangle$, which has an essentially identical abstract semantics. Having proper multicasting primitives for session communication is, however, useful especially in the case of sessions involving a large number of participants using multicast protocols such as IP-multicast through APIs. It also enriches the type structures: We extend $\mathtt{p} \to \mathtt{p}' : k$ in the prefix of global types to $\mathtt{p} \to \mathtt{p}_1, \ldots, \mathtt{p}_n : \{k_1, \ldots, k_n\}$ (with a practical adaptation such as group addressing), representing the multicast of a message to $\mathtt{p}_1, \ldots, \mathtt{p}_n$ via channels $k_1, \ldots, k_n$ by participant p; similarly, we extend endpoint session types to $\tilde{k}!\langle U\rangle$ from $k!\langle U\rangle$. Causality analysis remains the same by decomposing each multicasting prefix into its unicasting elements and considering causality for each of them. Our subsequent work [Bettini et al. 2008; Coppo et al. 2015a, 2015b] uses multicasting and proves the progress properties in asynchronous multiparty sessions.

6.2. Related Work

There is a large literature on session types for both process calculi (in particular π-calculi) and programming languages. Here, we discuss some of the most closely related works.

Asynchronous Session Types. Multiparty session types are based on message-order preserving asynchronous communication. Operational semantics of binary sessions based on asynchronous communication was first considered by Neubauer and Thiemann [2004b]. Recently, Gay and Vasconcelos [2009] studied the asynchronous version of binary sessions for an ML-like language [Vasconcelos et al. 2006]. In Gay and Vasconcelos [2009], message queues are given two endpoint channels and a direction.

Coppo et al. [2007] study asynchronous binary session types for Java, extending the previous work [Dezani-Ciancaglini et al. 2006] and proving progress by introducing an effect system. The resulting system does not allow interleaving sessions so that interactions involving more than two parties—such as our examples in Section 2.3—cannot be represented. Our theorem establishes the progress property on multiple session channels, significantly enlarging the framework in Coppo et al. [2007]. Recently, Dezani-Ciancaglini et al. [2007] propose a typing system for progress in binary synchronous interleaving sessions. There, typable processes obey the partial orders of shared and session channels inferred during type-checking. Because of the use of global types, processes typed by our multiparty session typing do not have to follow such ordering; on the other hand, the system in Dezani-Ciancaglini et al. [2007] does not require the simpleness condition (Definition 5.25). In Dezani-Ciancaglini et al. [2007], a progress property is defined as a typable process that never reduces to a process which contains open sessions (this amounts to containing session channels) and that is irreducible in any inactive context (represented by another inactive process running in parallel). A combination of this and our multiparty session typing systems will enlarge typability, guaranteeing progress in many situations. See also Section 6.3.

The concurrent work done by Bonelli and Compagnoni [2007], which is independently conceived and developed, studies a multiparty session typing for asynchronous

communication. Although treating the common topic, the technical direction of their work is different from that of the present work. Instead of global types, they solely use what we call (recursion-free) endpoint types. In type-checking, endpoint types are projected to each binary session so that type safety can be ensured using duality. Since we lose sequencing information in this way, the progress property is not guaranteed. The use of global types in the present work leads to transparent treatment of type structures such as recursion, the guarantee of stronger behavioral properties such as progress, and (arguably) more intelligible description of multiparty interaction structures.

Global Description of Session Types. Two recent works studied global descriptions of sessions in the context of Web services and business protocols [Carbone et al. 2007, 2012; Bhargavan et al. 2009]. Our work [Carbone et al. 2007, 2012] presented an *executable language* for directly describing Web interactions from a global viewpoint and provided the framework for projecting a description in the language to endpoint processes. The use of global description for *types* and its associated theories have not been developed in Carbone et al. [2007]. The type disciplines for the two (global and endpoint) calculi studied in Carbone et al. [2007] are based on binary synchronous session types; hence, safety and progress for multiparty interactions are not considered. See also Section 6.3 for further extensions of Carbone et al. [2007, 2012].

Bhargavan et al. [2009] investigates approaches to cryptographically protecting session execution from both external attackers in networks and malicious session principals. Their session specification models an interaction sequence between two or more constituent *roles*, an abstraction of network peers. The description is given as a graph whose node represents a specific state of a role in a session and whose edge denotes a dyadic communication and control flow. The purpose of the message flow graphs in Bhargavan et al. [2009] is more to serve as a model for systems and programs than to offer a type discipline for programming languages.

First, their work does not (aim to) present compositional typing rules for processes. Second, their flow graphs do not (try to) represent such elements as local control flow (e.g., prefixing), channel-based communication, and delegation. Third, their operational structures may not be oriented toward type abstractions: For example, their choice structures are based on transitions of flow graphs rather than on additive structures realizable by branching and selection.

Integration of their and our approaches is an interesting further topic: For example, we may consider developing a runtime validation method for multiparty sessions using flow models induced by our global types.

With a similar intent to address secure implementation of multiparty sessions, the works in Carbone and Guttman [2009a, 2009b] provide an abstract semantics for global types without parallel composition and recursion into the Strand Spaces model [Thayer et al. 1999]. The semantic function exploits a projection similar to ours.

Semantics of Delegation. For a simpler presentation, we used the operational semantics of delegation from Honda et al. [1998], which demands that delegated channels do not occur in the receiver. This prevents a process from acting as two or more participants in the same session, which usually leads to a deadlock. The duplication check is easily implementable in a way analogous to the standard mechanism of firewalls. The more generous rule [Gay and Hole 2005 and Yoshida and Vasconcelos 2007] allows substitution of session channels, as in [Recv], which also satisfies type safety and progress through annotations on channels and types. This annotation extends the method in Gay and Hole [2005] and Yoshida and Vasconcelos [2007]: Instead of polarities, we use indices by participants to annotate each usage of channels. With this change, the whole theories remain intact with exactly the same operational semantics and typing for programs. We study this delegation in Bettini et al. [2008] and Bejleri and Yoshida [2009].

Linear and Behavioral Types for Mobile Processes. Among many works on types for mobile processes, session type disciplines in general and the present work in particular are most closely related with linear/IO-typed π-calculi with causality information. The session type disciplines are related with linear and IO-typed π-calculi with causality information. The causality analysis in global types is partly inspired by the graph-based linear types developed in Yoshida [1996] and Yoshida et al. [2001], where ordering among multiple linear names (which correspond to session channels) guarantees deadlock freedom of typed processes. Several works [Kobayashi 2006; Igarashi and Kobayashi 2004] study generalized forms of linear typing for guaranteeing different kinds of deadlock freedom incorporating synchronizations and locking.

A main difference of session type disciplines from these and other preceding works in this field is a notion of rigorously structured conversations and their direct type abstraction. See Acciai and Boreale [2008] and Dezani-Ciancaglini et al. [2007] for detailed discussions, including comparisons between the session-based and the behavioral-based ones [Yoshida 1996; Yoshida et al. 2001; Kobayashi 2006]; in Acciai and Boreale [2008], Dezani-Ciancaglini et al. [2007], and Bettini et al. [2008], structured session primitives help to provide simpler typing systems for progress for binary sessions.

By raising the level of abstraction through the use of structured primitives such as separate session initiation, branching, and recursion, session types can describe complex interaction structures more intelligibly and enable efficient type-checking. These features would have direct applicability for the design of programming languages with communication [Hu et al. 2008; Carbone et al. 2007, 2012; Honda et al. 2007; Sackman and Eisenbach 2008; Pucella and Tov 2008; Scribble 2008].

One of the novelties of the present work is the introduction of global descriptions as types and a use of their projection for type-checking. They offer a modular and systematic causality analysis rather than directly working on individual syntax and operational semantics, with adaptations to asynchronous and synchronous communications. Composability of multiple programs is transparent through projection of a common global type, whereas complex syntax of types and typing are required in the traditional approach. To our knowledge, this method has not been investigated so far in the types of mobile processes.

Advanced Process Calculi and Types. Several process calculi for broadcasting have been investigated to model and analyze broadcasting networks including (recently) mobile ad-hoc networks, starting from Prasad's thesis [Prasad 2001]. Recent works focus on behavioral equivalences with lts [Merro 2007; Mezzetti and Sangiorgi 2006; Prasad 2006] and static analysis [Nanz et al. 2007] to investigate a number of different broadcasting methods. None studied the typing system and provided a strong progress guarantee as ensured by our session types. Our session types use a static participant information in the syntax and types. Recent advanced typing systems for location-based distributed processes [Hennessy 2007] use the similar notion for types $T@\mathtt{p}$, allowing one to dynamically instantiate locations into the capabilities using dependent type techniques. Since our aim is to prove the simplest extension of the original session types to multiparty sessions, static participants are enough even for delegations. A valuable further study will be to investigate a dynamic change of participant numbers during session initialization (without explicitly declaring $\mathtt{p}$ in the syntax) by using channel-dependent types [Mostrous and Yoshida 2007] or polymorphism.

Other Recent Service-Oriented Calculi. A vast amount of formal work for service-orientation has been done using process calculi and session types. The reader can refer to two recent surveys [Dezani-Ciancaglini and de' Liguoro 2010; Castagna et al. 2011] for more comparisons. We focus on the most related recent work. Different approaches to the description of service-oriented multiparty communications are taken in Bravetti

and Zavattaro [2007] and Bruni et al. [2008]. In Bravetti and Zavattaro [2007], the global and local views of protocols are described using a synchronous CCS-based calculus as a contract language and testing-preorders to check subcontract compliance; Bruni et al. [2008] propose a distributed calculus that provides communications either inside sessions or inside locations, modeling merging running sessions. *Contracts* [Castagna and Padovani 2009] use a process-based specification of protocols in which conformance means Must-preorder (so that we can ensure liveness). The system in Castagna and Padovani [2009] can type more processes than session types, thanks to the flexibility of process syntax for describing protocols. However, typable processes themselves in Castagna and Padovani [2009] may not always satisfy the properties of session types such as progress: It is proved later by checking whether the type meets a certain form. Hence, proving progress with contracts effectively requires an exploration of all possible paths (interleavings, choices) of a protocol.

Caires and Vieira [2010] propose a proof system that builds a well-founded ordering on events to enforce progress for processes of the Conversation Calculus [Vieira et al. 2008] where dynamic join and leave of participants are treated. These recent works do not treat a prescription of protocols offered by the global types, with the efficient projection and type-checking, which can ensure strong safety properties. Our recent work [Deniélou and Yoshida 2011] extends a dynamic join and leaving mechanism based on the multiparty session types by introducing a notion of *roles* that represent a unit of local behaviors.

6.3. Recent Works Based on Multiparty Session Types

This subsection summarizes works based on Multiparty Session Types published after the extended abstract [Honda et al. 2008a] of this article.

Theoretical Studies on Multiparty Session Types. Extensions of the original multiparty session types [Honda et al. 2008a] have been proposed, often motivated by use cases resulting from industry applications. Such extensions include a subtyping for asynchronous multiparty session types to enhance efficiency [Mostrous et al. 2009] motivated by financial protocols and multicore algorithms; parametrized global types for parallel programming and Web service descriptions [Deniélou et al. 2012]; communication buffered analysis [Deniélou and Yoshida 2010]; extensions to the sumtype and its encoding [Nielsen et al. 2010] for describing healthcare workflows; exception handling for multiparty conversations [Capecchi et al. 2016] for Web services and financial protocols; and a liveness-preserving refinement for multiparty session types [Padovani 2014b].

Multiparty session types can be extended with logical assertions following the design-by-contract framework [Bocchi et al. 2010]. This framework is enriched in Bocchi et al. [2012] to handle stateful logical assertions, whereas Chen and Honda [2012] offer more fine-grained property analysis for multiparty session types with these stateful assertions.

In Deniélou and Yoshida [2011], roles are inhabited by an arbitrary number of participants that can dynamically join and leave a session. Swamy et al. [2011] show that the multirole session types [Deniélou and Yoshida 2011] can be naturally represented in a dependent-typed language.

To enhance expressivity and flexibility of multiparty session types, Demangeon and Honda [2012] propose nested, higher order multiparty session types, and Castagna et al. [2012] study a generalization of choices and parallelism. Carbone and Montesi [2013] directly type a global description language [Carbone et al. 2012] by multiparty session types without using local types. This direct approach can type processes that are untypable in the original multiparty session typing (i.e., the communication type

system in this article). Montesi and Yoshida [2013] extend the work in Carbone and Montesi [2013] to compositional global description languages.

As another line of the study, we extend the multiparty session types to express temporal properties [Bocchi et al. 2014b]. In this work, the global times are enriched with time constraints in a way similar to timed automata.

A type system enforcing a stronger correspondence between nondeterministic choices expressed in multiparty session types and the behavior of processes involved in multiparty sessions has been investigated in Bocchi et al. [2014a].

Progress and Session Interleaving. Multiparty session types are a convenient methodology for ensuring the progress of systems of communicating processes. However, progress is only guaranteed within a *single* session [Honda et al. 2008a; Dezani-Ciancaglini and de' Liguoro 2010; Deniélou and Yoshida 2011], not when multiple sessions are interleaved. The first papers considering progress for interleaved sessions required the nesting of sessions in Java [Dezani-Ciancaglini et al. 2006; Coppo et al. 2007]. These systems can guarantee progress for only one single active binary session. Coppo et al. [2015b] develop a static interaction type system for global progress in dynamically interleaved and interfered multiparty sessions. A type inference algorithm for this system has been studied in Coppo et al. [2013], although for finite types only. Padovani [2014a, technical report] presents a type system for the linear π-calculus that can ensure progress even in presence of session interleaving, exploiting an encoding similar to that described in Dardha et al. [2012] of sessions into the linear π-calculus. However, not *all* multiparty sessions can be encoded into well-typed linear π-calculus processes. In this respect, the richer structure of multiparty session types increases the range of systems for which nontrivial properties such as progress can be guaranteed.

Security. Enforcement of *integrity* properties in multiparty sessions using session types has been studied in Bhargavan et al. [2009] and Planul et al. [2009]. These papers propose a compiler that, given a multiparty session description, implements cryptographic protocols that guarantee session execution integrity.

Capecchi et al. [2010] and in an extended version Capecchi et al. [2014] propose a session type system for a calculus of multiparty sessions enriched with security levels and adding access control and secure information flow requirements in the typing rules; they show that this type system guarantees preservation of data confidentiality during session execution. In Capecchi et al. [2015], this calculus is equipped with a monitored semantics that blocks the execution of processes as soon as they attempt to leak information and raises an error.

Behavioral Semantics. Typed behavioral theory has been one of the central topics in the study of the π-calculus throughout its history (e.g., reasoning about various encodings into the typed π-calculi [Pierce and Sangiorgi 1996; Yoshida 1996; Kouzapas et al. 2016]). In the context of typed bisimulations and reduction-closed theories, Kouzapas and Yoshida [2014] show that unique behavioral theories can be constructed based on the multiparty session types. The behavioral theory in Kouzapas and Yoshida [2014] treats the mutual effects of multiple choreographic sessions that are shared among distributed participants as their common knowledge or agreements, reflecting the origin of choreographic frameworks [WS-CDL 2003]. These features related to multiparty session type discipline make the theory distinct from any type-based bi-simulations in the literature and are also applicable to a real choreographic use case from a large-scale distributed system. This bisimulation is called *globally governed* since it uses global multiparty specifications to regulate the conversational behavior of distributed processes. It is an interesting future work to extend this work toward more scalable session bi-simulations for the eventful session types and the higher order π-calculus studied in Kouzapas et al. [2015].

Runtime Monitoring and Adaptations. Multiparty session types were originally developed to be used for the static type-checking of communicating processes. Via collaborations with Ocean Observatories Initiative [OOI 2015], it was discovered that the framework of multiparty session types can be naturally extended to runtime type-checking (monitoring). A formulation of the runtime monitoring (dynamic or runtime type-checking) was firstly proposed in Chen et al. [2012]. Later, Bocchi et al. [2013] formally proved its correctness and properties guaranteed by the runtime monitoring based on multiparty session types.

Works addressing adaptation for multiparty communications include Dalla Preda et al. [2014] and Coppo et al. [2014]. Dalla Preda et al. [2014] propose a choreographic language for distributed applications. Adaptation follows a rule-based approach in which all interactions, under all possible changes produced by the adaptation rules, proceed as prescribed by an abstract model. In Coppo et al. [2014], a calculus based on global types, monitors, and processes is introduced, and adaptation is triggered after the execution of the communications prescribed by a global type in reaction to changes of the global state.

Linkages with Other Frameworks. Deniélou and Yoshida [2012] give a linkage between communicating automata [Brand and Zafiropulo 1983] and a general graphical version of multiparty session types, proving a correspondence between the safety properties of communicating automata and multiparty session types. This work [Deniélou and Yoshida 2012] studies more detailed semantics for global and local types, relating with other frameworks such as model checking and logical verification for contracts [Villard 2011; Basu et al. 2012] (see Deniélou and Yoshida [2012, §5] for detailed comparisons).

Deniélou and Yoshida [2013] study the sound and complete characterization of the multiparty session types in communicating automata (called *multiparty compatibility*) and apply the result to the synthesis of multiparty session types. The inference of global types from a set of local types is also studied in Lange and Tuosto [2012]. The techniques developed in Deniélou and Yoshida [2013] and Lange and Tuosto [2012] are extended to a synthesis of general graphical multiparty session types in Lange et al. [2015]. This connection is extended to timed communicating automata [Krcál and Yi 2006] and Bocchi et al. [2015] propose general conditions of progress and non-zero properties of timed communicating automata at the top of multiparty compatibility.

Fossati et al. [2014] study the relationship of multiparty session types with Petri Nets. They propose a conformance relation between global session nets and endpoint programs and prove its safety.

A recent work [Carbone et al. 2015] studies a relationship with Linear Logic and multiparty session types along the line of Wadler [2012] and Caires and Pfenning [2010].

Implementations Based on Multiparty Session Types. We are currently designing and implementing a modeling and specification language with multiparty session types [SAV 2010; Scribble 2008] in collaboration with some industrial partners [Honda et al. 2011, 2014]. This protocol language is called Scribble. An article [Yoshida et al. 2013] also explains the origin and recent development of Scribble.

Java protocol optimization [Sivaramakrishnan et al. 2010] based on multiparty session types and the generation of multiparty cryptographic protocols [Bhargavan et al. 2009] are also studied. The multiparty session type theory is applied to healthcare workflows [Henriksen et al. 2013]. Its prototype implementation (the multiparty session π-processes with sumtypes) is available from Apims [2014].

Based on the runtime type-checking theory, we are implementing runtime monitoring [Demangeon et al. 2015; Hu et al. 2013; Neykova et al. 2013] under collaborations with Ocean Observatories Initiative [OOI 2015]. Demangeon et al. [2015] and Hu et al. [2013] allow interruptions in Scribble and prove the correctness of this extension.

Furthermore, we generalize the Python implementation to the Actor framework [Neykova and Yoshida 2014]. In order to express temporal properties studied in timed multiparty session types [Bocchi et al. 2014b], Neykova et al. [2014] extend Scribble with timed constrains and implement the runtime monitoring in Python.

We also apply the multiparty session types to high-performance parallel programming in C [Ng et al. 2012, 2012] and MPI [Ng and Yoshida 2014]. A parameterized version of Scribble [Ng and Yoshida 2014; Ng et al. 2013] based on the theory of parameterized multiparty session types [Deniélou et al. 2012] is developed. This extension, called Pabble, is used for automatically generating MPI parallel programs from sequential C code in Ng et al. [2015].

7. CONCLUSION

One of the main open problems of the session types is whether binary sessions can be extended to n-party sessions and, if they can, to determine their additional expressiveness and benefits. This article answers the question affirmatively. The present theory can guarantee stronger conformance to stipulated conversation structures than binary sessions when a protocol involves more than two parties. We proposed a new efficient type-checking system and proved type safety and progress extended to multiparty interactions. The central technical underpinning of the present work is the introduction of global types that offer an intuitive syntax for describing multiparty conversation structures from a global viewpoint and the use of their projection for efficient type-checking, thus proposing a new effective methodology for programming multiparty interactions in distributed environments. Global types also offer a basis for a clean modular causal analysis systematically applicable to both synchronous and asynchronous communications to ensure progress and session fidelity.

There are several significant future topics on the theory and applications of the proposed theory. We are currently starting to use this generalized session type structure as one of the formal foundations for the next version of a web service description language (based on an idea from WS-CDL [2003]) developed in Scribble from JBoss Red Hat [Scribble 2008], a message scheme for financial protocols; for a testable architecture, SAVARA from JBoss Red Hat [SAV 2010]; for a specification for message middleware from AMQP [AMQ 2015]; for a specification for large distributed systems from Ocean Observatories Initiative [OOI 2015]; and for a software development life cycle from Zero Diviation Life Cycle (ZDLC) [zdl 2015]. In particular, we are currently designing and implementing a modeling and specification language with multiparty session types [Scribble 2008] for these standards with our industry collaborators. This consists of three layers: The first layer is a global type that corresponds to a signature of class models in UML; the second one is for conversation models in which signatures and variables for multiple conversations are integrated; and the third layer includes extensions of the existing languages (such as Java [Hu et al. 2008, 2010; Ng et al. 2011]) that implement conversation models. Other future topics include tools assistance for the design and elaboration of global types, incorporation of typed exceptions into sessions, and the integration of the type discipline with diverse specification concerns including security and monitoring for distributed messages by the assertional methods [Bocchi et al. 2010].

APPENDIX

A. PROOF OF PROPOSITION 3.13

Here, the proofs of both Conditions (1) and (2) induce concrete algorithms. Global types are generally treated as regular trees (except, e.g., when we consider substitution). We first introduce the following notation.

Notation A.1.

(i) In the following, we write $G(0), G(1), \ldots, G(n), \ldots$ for the result of n-times unfolding of each recursion in G. For example, if G is $\mu \mathbf{t}.G'$ and this is the only recursion in G, then $G(0)$ is given as $G'[\mathsf{end}/\mathbf{t}]$, $G(1)$ is given as $G'[G(0)/\mathbf{t}]$, and, for each n, $G(n+1)$ is given as $G'[G(n)/\mathbf{t}]$. If G contains more than one recursion, we perform the unfolding of each of its recursions. For convenience, we set $G(-1)$ to be the empty graph.
(ii) Observing each $G(n+1)$ is the result of adding zero or more unfoldings to $G(n)$, so that $G(n+1)$ contains the exact copy of $G(n)$; we write $G(n+1)\backslash G(n)$ to denote the newly added (unfolded) part of $G(n+1)$.
(iii) Given a node n in $G(m+1)\backslash G(m)$, we can jump back from n once to reach its "original" in $G(m)\backslash G(m-1)$ (which is $G(0)$ if $m=0$). This exact copy of n that was created "one unfolding ago" is called the *one-time folding of* n, or simply the *folding of* n. In the same way, we define *the i-th folding of* n, which is in $G(m-i+1)\backslash G(m-i)$ (which is $G(0)$ if $i = m+1$). Note that there are $m+1$ such "foldings" of n in $G(m+1)\backslash G(m)$.

Proof of (1). Here, we say there are input/output dependencies from n_1 to n_2 when there is an input dependency *and* an output dependency from n_1 to n_2.

Claim. (A) Suppose $\mathsf{n}_{1,2}$ and their respective i-th foldings $\mathsf{n}'_{1,2}$ are in $G(m)$. Then, there are both input/output dependencies from n_1 to n_2 if and only if there are both input/output dependencies from n'_1 to n'_2. **(B)** Let n' be the folding of n. Then there is always both input and output dependencies from n' to n.

Proof of Claim. Condition (A) is immediate since the graph structure of the foldings is identical to that of the originals (i.e., we can simply "fold" the original two onto their foldings and all prefix relations coincide). Condition (B) is obvious since there always exist both II and OO dependencies by the definition of linearity. □

We now prove the statement. Fix a global type G and assume $G(1)$ is linear. We show by induction on n ($n \geq 1$) that each $G(n)$ is linear. Henceforth, we ignore nodes in carried types.

Base step. This is linearity of $G(1)$, which is the assumption itself.

Induction Step. Suppose $G(n)$ is linear. Then take two nodes n_1 and n_2 in $G(n+1)$ (but not in carried types) that happen to share a common channel. We show there are input/output dependencies from n_1 to n_2, or the same holds in the reverse direction. We say such $\mathsf{n}_{1,2}$ are *conflict-free* for brevity. We do case analysis depending on the position of these nodes in $G(m+1)$.
(i) If $\mathsf{n}_{1,2}$ are in $G(n)$, then they already have input/output dependencies by induction hypothesis.
(ii) If n_1 is in $G(n)\backslash G(n-1)$ and n_2 is in $G(n+1)\backslash G(n)$, then take their two foldings (say n'_1 and n'_2 , respectively). By induction hypothesis, they are conflict-free by a pair of dependency chains. By Claim A, we are done.
(iii) If n_1 is in $G(n-i)$ ($i \geq 1$) and n_2 is in $G(n+1)\backslash G(n)$, then take the folding of n_2 (say n'_2) which is in $G(n)$. By induction, we know n_1 and n'_2 are conflict-free.
By Claim B, there are both input and output dependencies from n_2 to n'_2. Thus, we have both input and output dependencies from n_1 to n'_2 and n'_2 to n_2 (hence, n_1 to n_2). Now we connect these chains and we are done. □

B. FULL TYPING RULES FOR RUNTIME PROCESSES

This appendix first presents the full typing rules, except those for expressions.

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : (G \upharpoonright 1)@1 \quad \{1, \ldots, \mathrm{n}\} = \mathsf{pid}(G) \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash_\emptyset \overline{a}_{[2..\mathrm{n}]}(\tilde{s}).P \triangleright \Delta} \quad [\textsc{Mcast}]$$

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : (G \upharpoonright \mathrm{p})@\mathrm{p} \quad \mathrm{p} \in \mathsf{pid}(G) \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash_\emptyset a_{[\mathrm{p}]}(\tilde{s}).P \triangleright \Delta} \quad [\textsc{Macc}]$$

$$\frac{\Gamma \vdash e_j : S_j \qquad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_\emptyset s[k]!\langle \tilde{e} \rangle; P \triangleright \Delta, \tilde{s} : k! \langle \tilde{S} \rangle; T@\mathrm{p}} \qquad \frac{\Gamma, \tilde{x} : \tilde{S} \vdash P_\emptyset \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_\emptyset s[k]?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k? \langle \tilde{S} \rangle; T@\mathrm{p}} \quad [\textsc{Send}], [\textsc{Rcv}]$$

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_\emptyset s[k]!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright \Delta, \tilde{s} : k! \langle T'@\mathrm{p}' \rangle; T@\mathrm{p}, \ \tilde{t} : T'@\mathrm{p}'} \qquad \frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T@\mathrm{p}, \ \tilde{t} : T'@\mathrm{p}'}{\Gamma \vdash_\emptyset s[k]?(\!(\tilde{t})\!); P \triangleright \Delta, \tilde{s} : k? \langle T'@\mathrm{p}' \rangle; T@\mathrm{p}} \quad [\textsc{Deleg}], [\textsc{SRec}]$$

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T_j@\mathrm{p} \qquad j \in I}{\Gamma \vdash_\emptyset s[k] \lhd l; P \triangleright \Delta, \tilde{s} : k \oplus \{l_i : T_i\}_{i \in I}@\mathrm{p}} \qquad \frac{\Gamma \vdash_\emptyset P_i \triangleright \Delta, \tilde{s} : T_i@\mathrm{p} \qquad \forall i \in I}{\Gamma \vdash_\emptyset \ s[k] \rhd \{l_i : P_i\}_{i \in I} \triangleright \Delta, \tilde{s} : k \,\&\{l_i : T_i\}_{i \in I}@\mathrm{p}} \quad [\textsc{Sel}], [\textsc{Branch}]$$

$$\frac{\Gamma \vdash_\emptyset e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash_\emptyset \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta} \quad [\textsc{If}]$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta \quad \Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta' \quad \tilde{t}_1 \cap \tilde{t}_2 = \emptyset \quad \Delta \asymp \Delta'}{\Gamma \vdash_{\tilde{t}_1 \cdot \tilde{t}_2} P \mid Q \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \Delta \circ \Delta'} \quad [\textsc{Conc}]$$

$$\frac{\Delta \text{ end only} \quad \Delta' \text{ [ ] only}}{\Gamma \vdash \mathbf{0} \triangleright_\emptyset \Delta, \Delta'} \qquad \frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'} \quad [\textsc{Inact}], [\textsc{Subs}]$$

$$\frac{\Gamma, a : \langle G \rangle \vdash_{\tilde{t}} P \triangleright \Delta}{\Gamma \vdash_{\tilde{t}} (\nu\, a)P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : \{T_\mathrm{p}@\mathrm{p}\}_{\mathrm{p} \in I} \quad \tilde{s} \in \tilde{t} \quad \{T_\mathrm{p}@\mathrm{p}\}_{\mathrm{p} \in I} \text{ coherent}}{\Gamma \vdash_{\tilde{t} \setminus \tilde{s}} (\nu\, \tilde{s})P \triangleright \Delta} \quad [\textsc{NRes}], [\textsc{CRes}]$$

$$\frac{\Gamma \vdash \tilde{e} : \tilde{S} \qquad \Delta \text{ end only}}{\Gamma, X : \tilde{S}\tilde{T} \vdash_\emptyset X\langle \tilde{e}\tilde{s}_1..\tilde{s}_n \rangle \triangleright \Delta, \tilde{s}_1 : T_1@\mathrm{p}_1, \ldots, \tilde{s}_n : T_n@\mathrm{p}_\mathrm{n}} \quad [\textsc{Var}]$$

$$\frac{\Gamma, X : \tilde{S}\tilde{T}, \tilde{x} : \tilde{S} \vdash_\emptyset P \triangleright \tilde{s}_1 : T_1@\mathrm{p}_1..\tilde{s_n} : T_n@\mathrm{p}_\mathrm{n} \quad \Gamma, X : \tilde{S}\tilde{T} \vdash_{\tilde{t}} Q \triangleright \Delta}{\Gamma \vdash_{\tilde{t}} \mathsf{def}\ X(\tilde{x}\tilde{s}_1..\tilde{s}_n) = P\ \mathsf{in}\ Q \triangleright \Delta} \quad [\textsc{Def}]$$

The typing rules for queues are from Figure 8.

B.1. Proof of Proposition 5.5

Suppose P is a program phrase. By definition, P is without queues and without bound channels. We show two implications.

(1) $\Gamma \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright_\emptyset \Delta$: Suppose P is typable in the original typing rules (for program phrases). Since the typing rules for runtime processes subsume the original rules, they can type P with the same derivation.

(2) $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ without [Subs] implies $\Gamma \vdash P \triangleright \Delta$: Suppose P is typable in the refined system as $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ without type contexts in Δ and without using [Subs]. By the lack of [Subs] in the derivation, the derivation precisely follows the structure of P. We inspect the potential differences between the original rules and the refined rules.

—(Use of Type Contexts in Derivation) Suppose the derivation uses a type context. The only place it can be taken off is [Conc]. Since there is no queue in P, this means the type context has been empty as the result of weakening by [Inact]. Hence its use can be taken off from the derivations.
—(Use of Refined Constraints on Queue Channels in Judgments) Since the only rule that decreases the number of mentioned queue channels in the judgment (as in $\triangleright_{\tilde{s}}$) is [CRes], we know each judgment in the derivation has the $\emptyset$ as its mentioned queue channels. Hence the constraint on queue channels in [Conc] and other rules are never used.

Thus, this derivation for P in the refined rules offers the derivation in the original rules as is; hence, done. □

B.2. Proof of Proposition 5.6

Assume $\Gamma \vdash P \triangleright_{s[1..m]} \Delta$. We call $s_1 ... s_m$ in $\Gamma \vdash P \triangleright_{s[1..m]} \Delta$, the judgment's *mentioned queue channels* or simply *queue channels*.

We first show there is one-to-one correspondence between the free queues in P and the mentioned queue channels by inspecting each rule.

Case [Inact]: Zero queue channel to zero queue.

Case [Qnil]: It connects precisely one channel to one queue.

Case [Qval], [Qsess], [Qsel]: These "enqueue" rules leave the number of channels one assigned to the unique queue channel.

Case [Mcast], [Macc], [Send], [Rcv], [Deleg], [SRec], [Sel], and [Branch], [If], [Var], [Def]: Each of these process construction rules leaves the queue channels unchanged (empty).

Case [Conc]: In the premise, assume $\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta$ and $\Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta'$ the free queues in P have channels $\tilde{t}_1$ whereas the free queues in Q have channels $\tilde{t}_2$. Since we assume $\tilde{t}_1 \cap \tilde{t}_2 = \emptyset$ and $P|Q$ have exactly the sum of their respective queues.

Case [NRes]: The rule leaves both the queues and the queue channels unchanged.

Case [CRes]: The rule precisely takes off those channels whose channels become bound.

Case [Subs]: No change in the process and no change in the queue. This exhausts all cases.

By the preceding case analysis, we conclude that free queues and mentioned queue channels precisely correspond to each other. Furthermore, the case analysis also shows that each prefix rule assumes that the process has no free queue before prefixing (in the premise). Furthermore, a program phrase cannot have channel restriction, so that all of its existing queues should be recorded in queue channels. We can now conclude that no queue can be under a prefix. □

B.3. Proof of Lemma 5.14

By the definition of $\circ$ on Δ, it suffices to show the commutativity and associativity at the level of types and type contexts, assuming that combined type contexts never share a target channel (in the sense defined just before Lemma 5.14, page 33).

We first show the commutativity. We write $H_1 \asymp H_2$ (which we read: "H_1 and H_2 are coherent") when $H_1 \circ H_2$ is defined. Note $H_1 \circ H_2$ means either:

—both of $H_{1,2}$ are type contexts and they do not share a target channel; or
—one of $H_{1,2}$ is a type context and the other is a type.

Here, the designation "context-context" means the case when we compose two contexts, similarly for others.

Case Context-Context: We consider the composition of $\mathcal{T}_{1,2}$ that are disjoint in targets (by our assumption). Then we always have:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{22}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2 = \mathcal{T}_1[\mathcal{T}_2] \tag{23}$$

By the symmetry of $\asymp$ (or equivalently by the assumption on target channels) we have:

$$\mathcal{T}_2 \asymp \mathcal{T}_1 \tag{24}$$
$$\mathcal{T}_2 \circ \mathcal{T}_1 = \mathcal{T}_2[\mathcal{T}_1] \tag{25}$$

Because of the isomorphism by the permutation equivalence for target-disjoint type contexts (cf. Section 5.1, paragraph **Type contexts**: recall $\approx$ is extended to type contexts unlike $\leq_{\mathsf{sub}}$), we have $\mathcal{T}_1[\mathcal{T}_2] \approx \mathcal{T}_2[\mathcal{T}_1]$; hence, we are done.

Case Type-Context: Immediate since, by definition, $\mathcal{T} \asymp T$ and $T \asymp \mathcal{T}$ always and $\mathcal{T} \circ T = T \circ \mathcal{T} = \mathcal{T}[T]$.

Case Context-Type: Symmetric to the preceding case.

Case Type-Type: Never defined, hence vacuous.

This exhausts all cases.

Next we show associativity.

Case Context-Context-Context: We consider the composition of $\mathcal{T}_{1,2,3}$, showing $(\mathcal{T}_1 \circ \mathcal{T}_2) \circ \mathcal{T}_3$ and $\mathcal{T}_1 \circ (\mathcal{T}_2 \circ \mathcal{T}_3)$ coincide in definedness and their resulting values. Assume $\mathcal{T}_{1,2}$ are mutually disjoint in target channels, similarly for $\mathcal{T}_1[\mathcal{T}_2]$ and $\mathcal{T}_3$. Then, automatically:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{26}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp \mathcal{T}_3 \tag{27}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ \mathcal{T}_3 = \mathcal{T}_1[\mathcal{T}_2][\mathcal{T}_3] \tag{28}$$

By Equation (27) we have:

$$\mathcal{T}_2 \asymp \mathcal{T}_3 \tag{29}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[\mathcal{T}_3] \tag{30}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[\mathcal{T}_3] = \mathcal{T}_1[\mathcal{T}_2[\mathcal{T}_3]] \tag{31}$$

Since $\mathcal{T}_1[\mathcal{T}_2][\mathcal{T}_3] = \mathcal{T}_1[\mathcal{T}_2[\mathcal{T}_3]]$, we are done. The other direction is symmetric.

Case Context-Context-Type: We consider the composition of $\mathcal{T}_{1,2}$ and T, showing that the definedness and the resulting value of $(\mathcal{T}_1 \circ \mathcal{T}_2) \circ T$ and $\mathcal{T}_1 \circ (\mathcal{T}_2 \circ T)$ coincide. This case is not symmetric; hence, we show both directions. First, if $\mathcal{T}_{1,2}$ are disjoint then automatically:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{32}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp T \tag{33}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ T = \mathcal{T}_1[\mathcal{T}_2][T] \tag{34}$$

We also always have:

$$\mathcal{T}_2 \asymp T \tag{35}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{36}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[T] = \mathcal{T}_1[\mathcal{T}_2[T]] \tag{37}$$

Since $\mathcal{T}_1[\mathcal{T}_2][T] = \mathcal{T}_1[\mathcal{T}_2[T]]$, we are done. For the other direction, we first compose $\mathcal{T}_2$ and T, then compose $\mathcal{T}_1$. As noted, we always have

$$\mathcal{T}_2 \asymp T \tag{38}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{39}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[T] = \mathcal{T}_1[\mathcal{T}_2[T]] \tag{40}$$

By our assumption, $\mathcal{T}_1$ and $\mathcal{T}_2$ do not share a target channel. Hence:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{41}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp T \tag{42}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ T = \mathcal{T}_1[\mathcal{T}_2][T] \tag{43}$$

Again, we note $\mathcal{T}_1[\mathcal{T}_2[T]] = \mathcal{T}_1[\mathcal{T}_2][T]$; hence, we are done.

Case Type-Context-Context, Context-Type-Context: By the preceding case Context-Context-Type and commutativity.

Since we can never combine two types, this exhausts all cases. □

B.4. Proof of Subject Reduction Theorem (Theorem 5.19)

Condition (1) is by rule induction on $\equiv$ showing, in both ways, that if one side has a typing, then the other side has the same typing. In the following, we safely ignore uninteresting (permutable) final applications of [Subs] in derivations by way of Lemma 5.16.

Case $P \mid \mathbf{0} \equiv P$: First assume $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. By $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \emptyset$ and by applying [Conc] to these two sequents, we immediately obtain $\Gamma \vdash P|\mathbf{0} \triangleright_{\tilde{s}} \Delta$, as required. For the converse direction, assume $\Gamma \vdash P|\mathbf{0} \triangleright_{\tilde{s}} \Delta$. We can safely assume (via Lemma 5.16) that the last rule applied is [Conc]. Thus, we can set $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta_1$ and $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. Note that we can safely regard $\Gamma \vdash \mathbf{0} \triangleright \Delta_2$ as being inferred by the axiom [Inact] since applying [Subs] to empty types and empty type contexts again leads to the empty types and empty type contexts: Thus Δ_2 consists of only empty types and empty type contexts. Thus, in the composition $\Delta_1 \circ \Delta_2$, the empty types and some of the empty type contexts from Δ_2 are added to Δ_1 to generate Δ. Let this added part be Δ_2'. Since we can weaken Δ_1 in the first sequent with Δ_2' using Lemma 5.18 Condition (2), we are done.

Case $P \mid Q \equiv Q \mid P$: By symmetry of the rule, we have only to show one direction. Suppose $\Gamma \vdash P|Q \triangleright_{\tilde{s}} \Delta$. We can safely assume the last rule applied is [Conc]. We can thus set $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$ and $\Gamma \vdash Q \triangleright_{\tilde{r}} \Delta_2$ such that $\Delta_1 \asymp \Delta_2$, $\Delta_1 \circ \Delta_2 = \Delta$ and $\tilde{t} \uplus \tilde{r} = \tilde{s}$. By Lemma 5.14, we know $\Delta_2 \asymp \Delta_1$ and $\Delta_2 \circ \Delta_1 = \Delta$; hence, by applying [Conc] with the premises reversed we are done.

Case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$: By the establishment of the previous case again we have only to show one direction. Suppose $\Gamma \vdash (P \mid Q) \mid R \triangleright_{\tilde{s}} \Delta$. We can safely assume $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$, $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta_2$ and $\Gamma \vdash P \triangleright_{\tilde{q}} \Delta_3$ such that $\Delta_1 \asymp \Delta_2$, $(\Delta_1 \circ \Delta_2) \asymp \Delta_3$ and $(\Delta_1 \circ \Delta_2) \circ \Delta_3 = \Delta$, as well as $\tilde{t} \uplus \tilde{r} \uplus \tilde{q} = \tilde{s}$. By the last condition, no two of Δ_1, Δ_2, and Δ_3 share a common target channel in their type contexts (in the sense given just before Lemma 5.14, page 33) because if the queue for a certain channel does not exist in a sequent then it cannot be used as a target channel in a type context in its typing. Thus,

we can apply Lemma 5.14 to know $\Delta_2 \asymp \Delta_3$, $\Delta_1 \asymp (\Delta_2 \circ \Delta_3)$ and $\Delta_1 \circ (\Delta_2 \circ \Delta_3) = \Delta$. By applying [Conc] in an appropriate order, we are done.

The remaining rules are reasoned exactly as in Yoshida and Vasconcelos [2007] (note the arguments for congruence rules are direct from the compositionality of the typing rules). This concludes the proof of Condition (1).

For Condition (2), we establish the following stronger claim by rule induction:

Claim. Suppose $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ and Δ is partially coherent (cf. Definition 5.9). Then, $P \to P'$ implies $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta'$ such that either $\Delta \xrightarrow{\ell} \Delta'$ or $\Delta = \Delta'$.

All results on reduction on coherent typing are immediately applicable to partially coherent typing by Proposition 5.13 Condition (1). Furthermore, by Proposition 5.13 Condition (3), Δ' above is again partially coherent. Here, we again ignore irrelevant final application of [Subs] through Lemma 5.16. All rule names are those of the typing rules.

Case [Link]: Let $R \stackrel{\text{def}}{=} \overline{a}_{[2..n]}(\tilde{s}).P_1 \mid a_{[2]}(\tilde{s}).P_2 \mid \cdots \mid a_{[n]}(\tilde{s}).P_n$ which is a redex of [Link]. We write R_1 for $\overline{a}_{[2..n]}(\tilde{s}).P_1$ and R_i for $a_{[i]}(\tilde{s}).P_i$ $(2 \leq i \leq n)$. Assume:

$$\Gamma \ \vdash \ R \ \triangleright \ \Delta. \tag{44}$$

By Lemma 5.15, we know $a \in \mathsf{dom}(\Gamma)$. Let $\Gamma(a) = G$. Since Equation (44) can only be inferred by the sequence of [Conc] (up to permutable [Subs], similarly in the following), we know $\Gamma \ \vdash \ R_i \ \triangleright \ \Delta_i$ $(1 \leq i \leq n)$ such that $\Delta_1 \circ \ldots \circ \Delta_n = \Delta$. By [Mcast] and [Macc] this means:

$$\Gamma \ \vdash \ P_i \ \triangleright \ \Delta_i, \tilde{s} : \{(G \upharpoonright i)@i\} \tag{45}$$

for each $1 \leq i \leq n$. Hence, by the successive applications of [Conc] we reach:

$$\Gamma \ \vdash \ (\Pi_i P_i) \mid (\Pi_i s_i :: \emptyset) \ \triangleright_{\tilde{s}} \ \Delta, \tilde{s} : \{(G \upharpoonright i)@i\}_{1 \leq i \leq n}. \tag{46}$$

Since $\{(G \upharpoonright i)@i\}_i$ collects all projections of G, we can apply [CRes] to obtain:

$$\Gamma \ \vdash \ (\nu \tilde{s})(\ (\Pi_i P_i) \mid (\Pi_i s_i :: \emptyset)\ ) \ \triangleright \ \Delta \tag{47}$$

for a reductum of [Link]. Note that the typing does not change.

Case [Send]: We use the first rule of Lemma 5.17 for "rolling back" a message. Suppose we have:

$$\Gamma \ \vdash \ s!\langle \tilde{e} \rangle; P \mid s :: \tilde{h} \ \triangleright_s \ \Delta. \tag{48}$$

Since [Conc] is the only rule to derive this process we can set

$$\Gamma \ \vdash \ s!\langle \tilde{e} \rangle; P \ \triangleright_{\emptyset} \ \Delta_1 \tag{49}$$

and

$$\Gamma \ \vdash \ s :: \tilde{h} \ \triangleright_s \ \Delta_2 \tag{50}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since Equation (49) can only be inferred from [Send] we know, first:

$$\Gamma \vdash e_j : S_j \tag{51}$$

for each e_j in $\tilde{e}$; and, second, for some p and for some $\tilde{s}$ that includes s,

$$\Delta_1 \ = \ \Delta_1' \circ \tilde{s} : k! \langle \tilde{S} \rangle; T@\mathrm{p} \tag{52}$$

and moreover

$$\Gamma \ \vdash \ P \ \triangleright_{\emptyset} \ \Delta_1' \circ \tilde{s} : T@\mathrm{p}. \tag{53}$$

On the other hand, by $\Delta_1 \asymp \Delta_2$ and Equation (50), we know:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}@\mathtt{p}. \tag{54}$$

Now assume $\tilde{e} \downarrow \tilde{v}$. Notice that by Equation (51) we have $\Gamma \vdash v_j : S_j$ for each v_j in $\tilde{v}$. Thus, by Lemma 5.17, [QVAL], we infer:

$$\Gamma \;\vdash\; s::\tilde{h} \cdot \tilde{v} \;\triangleright\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[k!\,\langle \tilde{S}\rangle; [\,]]@\mathtt{p}. \tag{55}$$

By the algebra of located types and type contexts:

$$\begin{aligned}(\Delta_1' \circ \tilde{s} : T@\mathtt{p}) \;\circ\; &(\Delta_2' \circ \tilde{s} : \mathcal{T}[k!\,\langle \tilde{S}\rangle; [\,]]@\mathtt{p}) \\ &= \;\; (\Delta_1' \circ \tilde{s} : k!\,\langle \tilde{S}\rangle; T@\mathtt{p}) \circ (\Delta_2' \circ \tilde{s} : \mathcal{T}[\,]@\mathtt{p}) = \Delta\end{aligned}$$

Thus, by applying [CONC] to Equations (49) and (50), we obtain:

$$\Gamma \;\vdash\; P \mid s::\tilde{h} \cdot \tilde{v} \;\triangleright\; \Delta, \tag{56}$$

which gives the expected typing for the reductum of [SEND], with no type change.

Case [DELEG]: Similar to [SEND], using the second rule of Lemma 5.17; see Appendix B.5.

Case [LABEL]: We use the third rule of Lemma 5.17 together with the subtyping $\leq_{\mathsf{sub}}$. Suppose we have:

$$\Gamma \;\vdash\; s \lhd l; P \mid s::\tilde{h} \;\triangleright_s\; \Delta, \tag{57}$$

which is the redex of [LABEL]. Since [CONC] is the only rule to derive this process, we can set, without loss of generality:

$$\Gamma \;\vdash\; s \lhd l; P \;\triangleright_\emptyset\; \Delta_1 \tag{58}$$

and

$$\Gamma \;\vdash\; s::\tilde{h} \;\triangleright_s\; \Delta_2 \tag{59}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since Equation (58) can only be inferred from [SEL] as the last rule (up to permutable applications of [SUBS]), we know, for some $\mathtt{p}$ and for some $\tilde{s}$ that includes s and for some $\{l_i\}$ that includes l,

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s} : k \oplus \{l_i : T_i\}_{i\in I}@\mathtt{p} \tag{60}$$

and, moreover,

$$\Gamma \;\vdash\; P \;\triangleright_\emptyset\; \Delta_1' \circ \tilde{s} : T_i@\mathtt{p}, \quad \text{for } i \in I. \tag{61}$$

On the other hand, we can write:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}@\mathtt{p}. \tag{62}$$

By Equations (59) and (62) and Lemma 5.17, [QSEL], we infer:

$$\Gamma \;\vdash\; s::\tilde{h} \cdot l \;\triangleright\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[k \oplus l : [\,]]@\mathtt{p}. \tag{63}$$

By the algebra of located types and type contexts together with subtyping:

$$\begin{aligned}(\Delta_1' \circ \tilde{s} : T_i@\mathtt{p}) \;\circ\; &(\Delta_2' \circ \tilde{s} : \mathcal{T}[k \oplus l : [\,]]@\mathtt{p}) \\ &= \Delta_1' \circ \Delta_2' \circ \tilde{s} : \mathcal{T}[k \oplus l : T_i]@\mathtt{p} \\ &\leq_{\mathsf{sub}} \Delta_1' \circ \Delta_2' \circ \tilde{s} : \mathcal{T}[k \oplus \{l_i : T_i\}_{i\in I}]@\mathtt{p} \\ &= (\Delta_1' \circ \tilde{s} : k \oplus \{l_i : T_i\}_{i\in I}@\mathtt{p}) \circ (\Delta_2' \circ \tilde{s} : \mathcal{T}@\mathtt{p}) \\ &= \Delta\end{aligned}$$

Thus we obtain, by applying [Conc] to Equations (61) and (63) then applying [Subs] (the subsumption rule):

$$\Gamma \vdash P \mid s :: \tilde{h} \cdot l \triangleright \Delta, \tag{64}$$

which gives the expected typing for the reductum of [Send], with no type change.

Case [Recv]: By the first of the latter three rules of Lemma 5.17 together with Lemma 5.18. Suppose

$$\Gamma \vdash s?(\tilde{x}); P \mid s :: \tilde{v} \cdot \tilde{h} \triangleright_s \Delta. \tag{65}$$

Since [Conc] is the only possible last rule (up to permutable [Subs]), we can set

$$\Gamma \vdash s?(\tilde{x}); P \triangleright_\emptyset \Delta_1 \tag{66}$$

and

$$\Gamma \vdash s :: \tilde{v} \cdot \tilde{h} \triangleright_s \Delta_2 \tag{67}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since Equation (66) can only be inferred from [Rcv], we know, for some p and for some $\tilde{s}$ that includes s,

$$\Delta_1 = \Delta_1' \circ \tilde{s} : k? \langle \tilde{S} \rangle; T@\mathrm{p} \tag{68}$$

and, moreover,

$$\Gamma, \tilde{x} : \tilde{S} \vdash P \triangleright_\emptyset \Delta_1' \circ \tilde{s} : T@\mathrm{p}. \tag{69}$$

By Lemma 5.18, we obtain:

$$\Gamma \vdash P[\tilde{v}/\tilde{x}] \triangleright_\emptyset \Delta_1' \circ \tilde{s} : T@\mathrm{p}. \tag{70}$$

Furthermore, by $\Delta_1 \asymp \Delta_2$ and Equation (67), we know:

$$\Delta_2 = \Delta_2' \circ \tilde{s} : k! \langle \tilde{S} \rangle . \mathcal{T}@\mathrm{p}. \tag{71}$$

By Lemma 5.17, [Qvaldq], we infer:

$$\Gamma \vdash s :: \tilde{h} \triangleright \Delta_2' \circ \tilde{s} : \mathcal{T}@\mathrm{p}. \tag{72}$$

Then we obtain:

$$\begin{aligned} \Delta &\stackrel{\text{def}}{=} (\Delta_1' \circ \tilde{s} : k? \langle \tilde{S} \rangle . T@\mathrm{p}) \circ (\Delta_2' \circ \tilde{s} : k! \langle \tilde{S} \rangle . \mathcal{T}@\mathrm{p}) \\ &\stackrel{\ell}{\to} (\Delta_1', \tilde{s} : T@\mathrm{p}) \circ (\Delta_2' \circ \tilde{s} : \mathcal{T}@\mathrm{p}) \quad (\stackrel{\text{def}}{=} \Delta'). \end{aligned}$$

Thus, by applying [Conc] to Equations (66) and (67), we obtain:

$$\Gamma \vdash P[\tilde{v}/\tilde{x}] \mid s :: \tilde{h} \triangleright \Delta' \tag{73}$$

such that $\Delta \stackrel{\ell}{\to} \Delta'$, as required. Note that this case demands reduction of typings.

Case [SRec], [Branch]: Similar to [Recv], using the latter two rules of Lemma 5.17; see Appendix B.5.

Case [IfT], [IfF], [Def], [Defin]: Standard; cf. Yoshida and Vasconcelos [2007]. No difference in the typing.

Case [Scop]: When a shared name is hidden, assume

$$\Gamma \vdash (\nu a) P \triangleright_{\tilde{s}} \Delta \tag{74}$$

and $P \to P'$. Then, we can set

$$\Gamma, a : \langle G \rangle \vdash P \triangleright_{\tilde{s}} \Delta. \tag{75}$$

By induction hypothesis, we know

$$\Gamma, a : \langle G \rangle \ \vdash \ P' \ \triangleright_{\tilde{s}} \ \Delta' \tag{76}$$

such that either $\Delta \xrightarrow{\ell}{}^{0,1} \Delta'$. Hence, by [NRES] we have

$$\Gamma \ \vdash \ (\nu a)P' \ \triangleright_{\tilde{s}} \ \Delta' \tag{77}$$

as required. When session channels are hidden, suppose

$$\Gamma \ \vdash \ (\nu \tilde{s})P \ \triangleright_{\tilde{t} \setminus \tilde{s}} \ \Delta \tag{78}$$

and $P \to P'$. We can set:

$$\Gamma \ \vdash \ P \ \triangleright_{\tilde{t}} \ \Delta, \tilde{s} : \{T_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p} \in I} \tag{79}$$

where $\{T_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p} \in I}$ is coherent. By induction hypothesis,

$$\Gamma \ \vdash \ P' \ \triangleright_{\tilde{t}} \ \Delta', \tilde{s} : \{T'_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p} \in I}, \tag{80}$$

where either $\Delta \xrightarrow{\ell}{}^{0,1} \Delta'$ or $\{s\} : \{T_p@p\}_{p \in I} \to^{0,1} \{s\} : \{T'_p@p\}_{p \in I}$. By Proposition 5.13 Condition (2), $\{T'_p@p\}_{p \in I}$ is again coherent. Hence, by [CRES], we obtain

$$\Gamma \ \vdash \ (\nu \tilde{s})P' \ \triangleright_{\tilde{t} \setminus \tilde{s}} \ \Delta' \tag{81}$$

as required.

Case [PAR]: Suppose we have $\Gamma \ \vdash \ P|Q \ \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \ \Delta$ and $P \to P'$. By [CONC], we have $\Gamma \ \vdash \ P \ \triangleright_{\tilde{t}_1} \ \Delta_1$ and $\Gamma \ \vdash \ Q \ \triangleright_{\tilde{t}_2} \ \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. By induction hypothesis, we have $\Gamma \ \vdash \ P' \ \triangleright_{\tilde{t}_1} \ \Delta'_1$ such that $\Delta_1 \to^{0,1} \Delta'_1$. By Proposition 5.13 Condition (1), we have $\Delta'_1 \asymp \Delta_2$; hence, $\Gamma \ \vdash \ P'|Q \ \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \ \Delta'_1 \circ \Delta_2$. Noting that Proposition 5.13 Condition (1) also says that $(\Delta_1 \circ \Delta_2) \to^{0,1} (\Delta'_1 \circ \Delta_2)$, we are done.

Case [STR]: Immediate from Subject Congruence (the first clause of this theorem). This exhausts all cases for Condition (2).

Condition (3) is because the empty typing $\emptyset$ is always coherent. □

B.5. Remaining Cases of Theorem 5.19

Case [DELEG]: We use the second rule of Lemma 5.17. Suppose we have:

$$\Gamma \ \vdash \ s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \mid s :: \tilde{h} \ \triangleright_s \ \Delta. \tag{82}$$

Since [CONC] is the only rule to derive this process, we can set

$$\Gamma \ \vdash \ s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \ \triangleright_{\emptyset} \ \Delta_1 \tag{83}$$

and

$$\Gamma \ \vdash \ s :: \tilde{h} \ \triangleright_s \ \Delta_2 \tag{84}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since Equation (83) can only be inferred from [DELEG], we know, for some p and for some $\tilde{s}$ that includes s,

$$\Delta_1 \ = \ \Delta'_1 \circ (\tilde{s} : k! \langle T'@\mathrm{p}' \rangle.T@\mathrm{p}, \ \tilde{t} : T'@\mathrm{p}') \tag{85}$$

and, moreover,

$$\Gamma \ \vdash \ P \ \triangleright_{\emptyset} \ \Delta'_1, \ \tilde{s} : T@\mathrm{p}. \tag{86}$$

On the other hand, by $\Delta_1 \asymp \Delta_2$ and Equation (50), we know:

$$\Delta_2 \ = \ \Delta'_2 \ \circ \ \tilde{s} : \mathcal{T}[\ ]@\mathrm{p}. \tag{87}$$

By Lemma 5.17, [Qsess], we infer:

$$\Gamma \ \vdash \ s::\tilde{h}\cdot\tilde{t} \ \triangleright_{\tilde{s}'} \ \Delta_2' \circ \tilde{s} : \{\mathcal{T}[k!\langle T@\mathrm{p}'\rangle.[\,]]@\mathrm{p}\},\ \tilde{t} : \{T@\mathrm{p}'\}. \tag{88}$$

By the algebra of located types and type contexts:

$$\begin{aligned}(\Delta_1',\ \tilde{s} : T@\mathrm{p}) \ &\circ \ (\Delta_2' \circ \tilde{s} : \{\mathcal{T}[k!\langle T@\mathrm{p}'\rangle.[\,]]@\mathrm{p}\},\ \tilde{t} : \{T@\mathrm{p}'\}) \\ &= (\Delta_1' \circ (\tilde{s} : k!\langle T'@\mathrm{p}'\rangle.T@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}')) \ \circ \ (\Delta_2' \ \circ \ \tilde{s} : \mathcal{T}[\,]@\mathrm{p}) \\ &= \Delta\end{aligned}$$

Thus, by applying [Conc] to Equations (83) and (84), we obtain:

$$\Gamma \ \vdash \ P \mid s::\tilde{h}\cdot\tilde{t} \ \triangleright \ \Delta, \tag{89}$$

which gives the expected typing for the reductum of [Deleg], with no type change.

Case [SRec]: By the second to the last rule of Lemma 5.17. Suppose

$$\Gamma \ \vdash \ s?((\tilde{t}\,)); P \mid s::\tilde{t}\cdot\tilde{h} \ \triangleright_s \ \Delta. \tag{90}$$

Since [Conc] is the only possible last rule (up to permutable [Subs]), we can set

$$\Gamma \ \vdash \ s?((\tilde{t}\,)); P \ \triangleright_{\emptyset} \ \Delta_1 \tag{91}$$

and

$$\Gamma \ \vdash \ s::\tilde{t}\cdot\tilde{h} \ \triangleright_s \ \Delta_2 \tag{92}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since Equation (91) can only be inferred from [SRec], we know, for some p and for some $\tilde{s}$ that includes s,

$$\Delta_1 \ = \ \Delta_1' \circ \tilde{s} : k?\langle T'@\mathrm{p}'\rangle.T@\mathrm{p} \tag{93}$$

and, moreover,

$$\Gamma \ \vdash \ P \ \triangleright_{\emptyset} \ \Delta_1' \circ \tilde{s} : T@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}'. \tag{94}$$

By $\Delta_1 \asymp \Delta_2$ and Equation (92), we know:

$$\Delta_2 \ = \ \Delta_2' \ \circ \ \tilde{s} : k!\langle T'@\mathrm{p}'\rangle.\mathcal{T}@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}'. \tag{95}$$

By Lemma 5.17, [Qsessdq], we infer:

$$\Gamma \ \vdash \ s::\tilde{h} \ \triangleright \ \Delta_2' \ \circ \ \tilde{s} : \mathcal{T}@\mathrm{p}. \tag{96}$$

Then we obtain:

$$\begin{aligned}\Delta \stackrel{\mathrm{def}}{=} (\Delta_1' \ \circ \ \tilde{s} : k?\langle T'@\mathrm{p}'\rangle.T@\mathrm{p}) \circ (\Delta_2' \ \circ \ \tilde{s} : k!\langle T'@\mathrm{p}'\rangle.\mathcal{T}@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}') \\ \stackrel{\ell}{\to} (\Delta_1' \ \circ \ \tilde{s} : T@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}') \circ (\Delta_2' \ \circ \ \tilde{s} : \mathcal{T}@\mathrm{p}) \quad (\stackrel{\mathrm{def}}{=} \Delta')\end{aligned}$$

Thus, by applying [Conc] to Equations (91) and (92), we obtain:

$$\Gamma \ \vdash \ P \mid s::\tilde{h} \ \triangleright \ \Delta' \tag{97}$$

such that $\Delta \stackrel{\ell}{\to} \Delta'$, as required. Note that this case again demands reduction of typings.

Case [Branch]: By the last rule of Lemma 5.17. Suppose

$$\Gamma \ \vdash \ s \rhd \{l_i : P_i\}_{i\in I} \mid s::l_j\cdot\tilde{h} \ \triangleright_s \ \Delta, \tag{98}$$

where we assume $j \in I$. Since [Conc] is the only possible last rule (up to permutable [Subs]), we can set

$$\Gamma \ \vdash \ s \rhd \{l_i : P_i\}_{i\in I} \ \triangleright_{\emptyset} \ \Delta_1 \tag{99}$$

and

$$\Gamma \vdash s::l_j \cdot \tilde{h} \triangleright_s \Delta_2 \tag{100}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. First, for Δ_2 we know, for some p and for some $\tilde{s}$ that includes s:

$$\Delta_2 = \Delta_2' \circ \tilde{s} : k \oplus l_j : \mathcal{T}@\mathrm{p}, \tag{101}$$

where by assumption we have $j \in I$. Since Equation (99) can only be inferred from [Branch] and by $\Delta_1 \asymp \Delta_2$, we also know:

$$\Delta_1 = \Delta_1' \circ \tilde{s} : k \,\&\, l_j : T_j@\mathrm{p} \tag{102}$$

(where $\& l_j : T_j$ is the singleton notation as in selection) and, moreover,

$$\Gamma \vdash P_i \triangleright_\emptyset \Delta_1' \circ \tilde{s} : T_i@\mathrm{p} \tag{103}$$

for each $i \in I$ (so Equation (102) is inferred using [Subs]). By Lemma 5.17, [QseldQ], we infer:

$$\Gamma \vdash s::\tilde{h} \triangleright \Delta_2' \circ \tilde{s} : \mathcal{T}@\mathrm{p}. \tag{104}$$

Then we obtain:

$$\begin{aligned}\Delta &\stackrel{\text{def}}{=} (\Delta_1' \circ \tilde{s} : k \,\&\, l_j : T_j@\mathrm{p}) \circ (\Delta_2' \circ \tilde{s} : k \oplus l_j : \mathcal{T}@\mathrm{p}) \\ &\stackrel{\ell}{\to} \quad (\Delta_1',\ \tilde{s} : T_j@\mathrm{p}) \circ (\Delta_2' \circ \tilde{s} : \mathcal{T}@\mathrm{p}) \quad (\stackrel{\text{def}}{=} \Delta').\end{aligned}$$

Thus, by applying [Conc] to Equations (99) and (100), we obtain:

$$\Gamma \vdash P \mid s::\tilde{h} \triangleright \Delta' \tag{105}$$

such that $\Delta \stackrel{\ell}{\to} \Delta'$, as required. Again, we need a reduction of typings. □

B.6. Proof of Lemma 5.21

Proof of Conditions (1) and (2). We prove the following claim, which implies both Conditions (1) and (2) by rule induction on the typing rules. Here and henceforth, we are confusing a free session channel and its numeric representation in the typing. Recall that Δ is partially coherent when for some Δ_0 we have $\Delta \asymp \Delta_0$ and $\Delta \circ \Delta_0$ is coherent.

Claim. Assume $\Gamma \vdash P \triangleright_t \Delta$ such that Δ is partially coherent and there is no queue at s. Assume $P\langle\!\langle s \rangle\!\rangle$. Then, one of the following conditions holds:

(a) P contains a unique active receiving (emitting, respectively) prefix at s and Δ contains a unique minimal receiving (emitting, respectively) prefix at s (Δ may contain another minimal prefix at s).
(b) P contains a unique minimal receiving prefix at s and a unique minimal emitting prefix at s. Moreover, Δ contains a unique minimal receiving prefix at s and a unique minimal emitting prefix at s.

Case [Mcast], [Macc]: Vacuous since in this case the unique active prefix in the process is at a shared name.

Case [Send], [Rcv], [Deleg], [SRec], [Sel], and [Branch]: Immediate since there can only be a unique active channel name, which is by the given prefixing.

Case [Inact], [If], [Var], [Def], [Qnil], [Qval], [Qsess], [Qsel]: Vacuous.

Case [Conc]: Suppose

$$\Gamma \vdash P \triangleright_{t_1} \Delta, \qquad \Gamma \vdash_{t_2} Q \triangleright \Delta' \tag{106}$$

such that $\tilde{t}_1 \cap \tilde{t}_2 = \emptyset$ and $\Delta \asymp \Delta'$. Observe that if $\Delta \circ \Delta'$ is partially coherent, then Δ and Δ', respectively, are partially coherent by definition. By induction hypothesis, we can assume P and Q satisfy the required condition.

(1) If only one party has an active prefix at s there is nothing to prove.
(2) If both are active at s, suppose both processes, hence Δ and Δ', have receiving active prefixes at s. Then this cannot be partially coherent since, if so, then the assumed completion of $\Delta \circ \Delta'$ to a coherent typing should also contain two minimal receiving prefixes, which is impossible by the definition of $\circ$. Similarly when two include active emitting prefixes at s; hence, as required.

Note that this pair may *not* be a redex: We do not (have to) validate coherence until we hide channels; however, it is important that there is one output and one input, otherwise there will be a conflict at s.

Case [NRes]: Vacuous since there is no change either in the process nor in the typing.

Case [CRes]: Vacuous since there is no difference in the typing for s nor in the activeness in prefixes.

Case [Subs]: Vacuous again. □

B.7. Proof of Proposition 5.26

We show the following logically equivalent result:

Claim. (1) If P is simple then

(1-a) no delegation prefix (input or output) occurs in P and
(1-b) for each prefix with a shared name in P, say $a[\mathtt{i}](\tilde{s}).P'$ or $\overline{a}[2..\mathtt{n}](\tilde{s}).P'$, there is no free session channel in P' except $\tilde{s}$.

(2) If P is simple and $P \to P'$, then P' is again simple.

We first show Condition (1) by rule induction on typing rules.

Case [Mcast]: The rule reads:

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : (G \upharpoonright 1)@1 \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash_\emptyset \overline{a}[2..\mathtt{n}]\,(\tilde{s}).P \triangleright \Delta}$$

First, by simplicity, we know $\Delta = \emptyset$ (since, if not, the premise has at least a doubleton typing). Condition (1-a) is immediate from the induction hypothesis since the rule does not add a delegation prefix: For Condition (1-b) if P' in $a[i](\tilde{s}).P'$ (resp. $\overline{a}[2..\mathtt{n}](\tilde{s}).P'$) has free session channels, then we cannot have $\Delta = \emptyset$, violating simplicity.

Case [Macc]: The rule reads:

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : (G \upharpoonright \mathtt{p})@\mathtt{p} \quad |\tilde{s}| = \mathsf{sid}(G)}{\Gamma \vdash_\emptyset a[\mathtt{p}]\,(\tilde{s}).P \triangleright \Delta}$$

Again, $\Delta = \emptyset$, and the remaining reasoning is precisely the same as in [Mcast].

Case [Send]: The rule reads:

$$\frac{\Gamma \vdash e_j : S_j \qquad \Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T@\mathtt{p}}{\Gamma \vdash_\emptyset s[k]!\langle \tilde{e} \rangle; P \triangleright \Delta, \tilde{s} : k!\,\langle \tilde{S} \rangle; T@\mathtt{p}}$$

Again, $\Delta = \emptyset$. Condition (1-a) is immediate from the induction hypothesis since the rule does not add any delegation prefix. Condition (1-b) is again immediate from the induction hypothesis since the rule doe not add a shared-name prefix.

Case [RCV]: The rule reads:

$$\frac{\Gamma, \tilde{x} : \tilde{S} \vdash P_{\emptyset} \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_{\emptyset} s[k]?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k?\langle \tilde{S} \rangle; T@\mathrm{p}}$$

Precisely the same as in [SEND].

Case [DELEG]: The rule reads:

$$\frac{\Gamma \vdash_{\emptyset} P \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_{\emptyset} s[k]!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright \Delta, \tilde{s} : k!\langle T'@\mathrm{p}' \rangle; T@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}'}$$

Even if $\Delta = \emptyset$, the conclusion's typing becomes a doubleton; hence, this rule cannot be applied.

Case [SREC]: The rule reads:

$$\frac{\Gamma \vdash_{\emptyset} P \triangleright \Delta, \tilde{s} : T@\mathrm{p},\ \tilde{t} : T'@\mathrm{p}'}{\Gamma \vdash_{\emptyset} s[k]?(\!(\tilde{t})\!); P \triangleright \Delta, \tilde{s} : k?\langle T'@\mathrm{p}' \rangle; T@\mathrm{p}}$$

which is again impossible to apply (the premise's typing becomes a doubleton).

Case [SEL],[BRANCH]: Similar with [SEND] and [RCV].

Case [IF], [CONC], [CRES], [NRES], [SUBS], [DEF]: By the shape of these rules, in each rule, there is no addition or removal of a prefix from the premise to the conclusion. Hence, both Conditions (1-a/b) are immediate from the induction hypothesis.

Case [INACT], [VAR], [QNIL], [QVAL], [QSESS], [QSEL]: Vacuous since no prefixes are involved.

Hence, as required.

For Condition (2), suppose a derivation of P is simple. By the proof of Theorem 5.19, if $P \to P'$, then we have essentially the same derivation for both P and P' except:

—taking off the last pair of prefixes from that of P (three pair of prefix rules);
—one of the branches is chosen (conditional)
—copying some part from the derivation for P to that of P' (for recursion)

In each case, clearly, the simplicity of the derivation for P implies that of P', as required. □

B.8. Proof of Lemma 5.28

Suppose:

(C1). $\Gamma \vdash P \triangleright \Delta$.
(C2). P is simple.
(C3). Δ has a minimal receiving (emitting, respectively) prefix at s.
(C4). none of the prefixes at s in P is under a shared name.
(C5). none of the prefixes at s in P is under a conditional branch.

Under these conditions, we show that P has an active receiving prefix (has an active emitting prefix or a non-empty queue, respectively). We use rule induction on typing rules.

Case [MCAST], [MACC]: By Proposition 5.26, there can be no free session channels; hence, vacuous (since Condition (C3) is not satisfied).

Case [SEND]: The "simple" rule reads:

$$\frac{\Gamma \vdash e_j : S_j \qquad \Gamma \vdash_\emptyset P \triangleright \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_\emptyset s[k]!\langle \tilde{e} \rangle; P \triangleright \tilde{s} : k! \langle \tilde{S} \rangle; T@\mathrm{p}}$$

Observe that there can be no other minimal prefix in the typing in the conclusion than the newly introduced prefix itself: This corresponds to the unique minimal prefix in the typing.

Case [RCV]: The "simple" rule reads:

$$\frac{\Gamma, \tilde{x} : \tilde{S} \vdash P_\emptyset \triangleright \Delta, \tilde{s} : T@\mathrm{p}}{\Gamma \vdash_\emptyset s[k]?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k? \langle \tilde{S} \rangle; T@\mathrm{p}}$$

Same as [SEND].

Case [SREC], [DELEG]: By Proposition 5.26, these rules are not used in derivation of a simple process; hence, vacuous.

Case [SEL],[BRANCH]: Similar with [SEND],[RCV].

Case [IF]: Vacuous since Condition (C5) does not hold.

Case [CONC]: The rule reads:

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta \quad \Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta' \quad \tilde{t}_1 \cap \tilde{t}_2 = \emptyset \quad \Delta \asymp \Delta'}{\Gamma \vdash_{\tilde{t}_1 \cdot \tilde{t}_2} P \mid Q \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \Delta \circ \Delta'}$$

We first observe:

Claim A1. If the result of the operation $\circ$ on typings (when defined) has a minimal input prefix, then one of the original typings also has the same.

This is because, direct from the definition of $\circ$, if $\circ$ results in an input minimal input prefix, then it cannot come from a type context (which contains only an output prefix); hence, it can come only from the same in the premise. Furthermore:

Claim A2. If the result of the operation $\circ$ on typings (when defined) has a minimal output prefix, then one of the premises also has the same in the form of either the corresponding non-empty type context or the corresponding type ("corresponding" means that the minimal prefix coincides).

The details of the shape of a typing are in fact unnecessary.

Claim B. The composition $\mid$ preserves activeness of each prefix.

This is immediate from the definition.

Now we reason by induction. In the case of an input prefix in the typing, by Claim A1, we know that one of the premises also contains an input prefix in the typing. Hence, the corresponding process has an active input prefix by induction hypothesis. By Claim B, we are done.

On the other hand, in the case of an output prefix in the typing, by Claim A2 we know one of the premises also contains the same (either as the corresponding type context or the corresponding output prefix) in the typing. Hence, by induction hypothesis, the corresponding process has an active output prefix or a non-empty queue. Hence, by induction hypothesis, we are done. By Claim B, we are done.

Case [INACT], [VAR]: Vacuous since, in this case, the typing does not contain any active channel, hence violating Condition (C3).

Case [SUBS]: The subsumption does not add any new active prefix in the typing; hence, by induction hypothesis, we are done.

Case [DEF]: As for [SUBS].

Case [QVAL], [QSESS], [QSEL]: In these cases, we have a minimal emitting prefix in the typing and we have a corresponding non-empty queue, as required.

Case [QNIL]: Vacuous since Condition (C3) is violated.

Case [NRES]: This reads:

$$\frac{\Gamma, a : \langle G \rangle \vdash_{\tilde{t}} P \triangleright \Delta}{\Gamma \vdash_{\tilde{t}} (\boldsymbol{\nu}\, a)P \triangleright \Delta}$$

which shows there is no change in the typing and in the process with respect to (free) active/minimal prefixes; hence, immediate by induction hypothesis.

Case [CRES]: This reads:

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : \{T_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p}\in I} \quad \tilde{s} \in \tilde{t} \quad \{T_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p}\in I}\ \text{coherent}}{\Gamma \vdash_{\tilde{t}\setminus\tilde{s}} (\boldsymbol{\nu}\, \tilde{s})P \triangleright \Delta}$$

Suppose in the conclusion there is a minimal prefix at s in Δ. Then, it is also minimal in the premise; hence, by induction hypothesis, we are done.

This exhausts all cases. □

ACKNOWLEDGMENT

We would like to thank Andi Bejleri for an early collaboration on this work.

REFERENCES

AMQP. 2015. Advanced Message Queuing Protocol. http://www.iona.com/opensource/amqp/.

Lucia Acciai and Michele Boreale. 2008. A type system for client progress in a service-oriented calculus. In *Concurrency, Graphs and Models (LNCS)*, Vol. 5065. Springer, Pisa, Italy, 642–658.

Apims 2014. Apims. (2014). http://thelas.dk/index.php?title=Apims.

Samik Basu, Tevfik Bultan, and Meriem Ouederni. 2012. Deciding choreography realizability. In *Symposium on Principles of Programming Languages (POPL'12)*. ACM, Philadelphia, USA, 191–202.

Andi Bejleri and Nobuko Yoshida. 2009. Synchronous multiparty session types. In *Proceedings of Programming Languages Approaches to Concurrency and Communication-Centric Software (PLACES'08) (ENTCS)*, Vol. 241. Elsevier, Oslo, Norway, 3–33.

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global progress in dynamically interleaved multiparty sessions. In *International Conference on Concurrency Theory (CONCUR'08) (LNCS)*, Vol. 5201. Springer, Toronto, Canada, 418–433.

Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James Leifer. 2009. Cryptographic protocol synthesis and verification for multiparty sessions. In *Computer Security Foundations Symposium (CSF'09)*. IEEE, New York, USA, 124–140.

Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2013. Monitoring networks through multiparty session types. In *IFIP Joint International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'13) (LNCS)*, Dirk Beyer and Michele Boreale (Eds.), Vol. 7892. Springer, Florence, Italy, 50–65.

Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. 2012. A multiparty multi-session logic. In *7th International Symposium on Trustworthy Global Computing (TGC'12) (LNCS)*, Catuscia Palamidessi and Mark Dermot Ryan (Eds.), Vol. 8191. Springer, Newcastle upon Tyne, UK, 111–97.

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A theory of design-by-contract for distributed multiparty interactions. In *International Conference on Concurrency Theory (CONCUR'10) (LNCS)*, Vol. 6269. Springer, Paris, France, 162–176.

Laura Bocchi, Julien Lange, and Nobuko Yoshida. 2015. Meeting deadlines together. In *International Conference on Concurrency Theory (CONCUR 2015) (LIPIcs)*, Vol. 42. Schloss Dagstuhl, Madrid, Spain, 283–296.

Laura Bocchi, Hernán C. Melgratti, and Emilio Tuosto. 2014a. Resolving non-determinism in choreographies. In *European Symposium on Programming (ESOP'14) (LNCS)*, Zhong Shao (Ed.), Vol. 8410. Springer, Grenoble, France, 493–512.

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014b. Timed multiparty session types. In *International Conference on Concurrency Theory (CONCUR'14) (LNCS)*, Paolo Baldan and Daniele Gorla (Eds.), Vol. 8704. Springer, Rome, Italy, 419–434.

Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. 2005. Correspondence assertions for process synchronization in concurrent communications. *Journal of Functional Programming* 15, 2 (2005), 219–248.

Eduardo Bonelli and Adriana B. Compagnoni. 2007. Multipoint session types for a distributed calculus. In *Trustworthy Global Computing (TGC'07) (LNCS)*, Vol. 4912. Springer, Sophia-Antipolis, France, 240–256.

BPMNC 2012. Business Process Model and Notation 2.0 Choreography. Retrieved from http://en.bpmn-community.org/tutorials/34/.

Daniel Brand and Pitro Zafiropulo. 1983. On communicating finite-state machines. *Journal of the ACM* 30 (April 1983), 323–342. Issue 2.

Mario Bravetti and Gianluigi Zavattaro. 2007. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition (LNCS)*, Vol. 4829. Springer, Braga, Portugal, 34–50.

Roberto Bruni, Ivan Lanese, Hernan Melgratti, and Emilio Tuosto. 2008. Multiparty sessions in SOC. In *Coordination Models and Languages (COORDINATION'08) (LNCS)*, Vol. 5052. Springer, Oslo, Norway, 67–82.

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory (CONCUR'10) (LNCS)*, Vol. 6269. Springer, Paris, France, 222–236.

Luís Caires and Hugo Torres Vieira. 2010. Conversation types. *Theoretical Computer Science* 411, 51–52 (2010), 4399–4440.

Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2014. Typing access control and secure information flow in sessions. *Information and Computation* 238 (2014), 68–105.

Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2015. Information flow safety in multiparty sessions. To appear.

Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. 2010. Session types for access and information flow control. In *International Conference on Concurrency Theory (CONCUR'10) (LNCS)*, Vol. 6269. Springer, Paris, France, 237–252.

Sara Capecchi, Elena Giachino, and Nobuko Yoshida. 2016. Global escape in multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 156–205.

Marco Carbone and Joshua Guttman. 2009a. Choreographies with Secure Boxes and Compromised Principals. In *Proceedings of the 2nd Interaction and Concurrency Experience - Structured Interactions (ICE'09) (EPTCS)*, Vol. 12. Bologna, Italy, 1–16.

Marco Carbone and Joshua Guttman. 2009b. Execution models for choreographies and cryptoprotocols. In *Proceedings of the 2nd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'09) (EPTCS)*, Vol. 17. York, UK, 31–42.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. Structured communication-centred programming for web services. In *European Symposium on Programming (ESOP'07) (LNCS)*, Vol. 4421. Springer, Braga, Portugal, 2–17.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured interactional exceptions in session types. In *International Conference on Concurrency Theory (CONCUR'08) (LNCS)*, Franck van Breugel and Marsha Chechik (Eds.), Vol. 5201. Springer, Toronto, Canada, 402–417.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems* 34, 2 (2012), 8.

Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. 2006. A Theoretical Basis of Communication-Centred Concurrent Programming. Retrieved from http://www.w3.org/2002/ws/chor/.

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Symposium on Principles of Programming Languages (POPL'13)*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, Rome, Italy, 263–274.

Marco Carbone, Fabrizio Montesi, Carsten Schrmann, and Nobuko Yoshida. 2015. Multiparty session types as coherence proofs. In *International Conference on Concurrency Theory (CONCUR'15) (LIPIcs)*, Vol. 42. Schloss Dagstuhl, Madrid, Spain, 412–426.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On global types and multiparty sessions. In *International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS/FORTE) (LNCS)*, Vol. 6722. Springer, Reykjavik, Iceland, 1–28.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. On global types and multiparty session. *Logical Methods in Computer Science* 8, 1 (2012), 24.

Giuseppe Castagna and Luca Padovani. 2009. Contracts for mobile processes. In *International Conference on Concurrency Theory (CONCUR'09) (LNCS)*. Springer, Bologna, Italy, 211–228.

Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. 2012. Asynchronous distributed monitoring for multiparty session enforcement. In *Trustworthy Global Computing (TGC'11) (LNCS)*, Roberto Bruni and Vladimiro Sassone (Eds.), Vol. 7173. Springer, Newcastle upon Tyne, UK, 25–45.

Tzu-Chun Chen and Kohei Honda. 2012. Specifying stateful asynchronous properties for distributed programs. In *International Conference on Concurrency Theory (CONCUR'12) (LNCS)*, Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer, Newcastle upon Tyne, UK, 209–224.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of global progress properties for dynamically interleaved multiparty sessions. In *Coordination Models and Languages (COORDINATION'13) (LNCS)*, Rocco De Nicola and Christine Julien (Eds.), Vol. 7890. Springer, Florence, Italy, 45–59.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015a. A gentle introduction to multiparty asynchronous session types. In *SFM-15:MP (LNCS)*, Vol. 9104. Springer, Bertinoro, Italy, 146–178.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 2014. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications* 9, 3–4 (2014), 249–268.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2007. Asynchronous session types and progress for object-oriented languages. In *IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'07) (LNCS)*, Vol. 4468. Springer, Paphos, Cyprus, 1–31.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2015b. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2015), 238–302.

Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. 2014. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *International Conference on Software Language Engineering (SLE'14) (LNCS)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.), Vol. 8706. Springer, Västerås, Sweden, 161–170.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'12)*, Danny De Schreye, Gerda Janssens, and Andy King (Eds.). ACM Press, Leuven, Belgium, 139–150.

Romain Demangeon and Kohei Honda. 2012. Nested protocols in session types. In *International Conference on Concurrency Theory (CONCUR'12) (LNCS)*, Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer, Newcastle upon Tyne, UK, 272–286.

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design* 46, 3 (2015), 197–225.

Pierre-Malo Deniélou and Nobuko Yoshida. 2010. Buffered communication analysis in distributed multiparty sessions. In *International Conference on Concurrency Theory (CONCUR'10) (LNCS)*, Vol. 6269. Springer, Paris, France, 343–357.

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *Symposium on Principles of Programming Languages (POPL'11)*. ACM, Austin, USA, 435–446.

Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty session types meet communicating automata. In *European Symposium on Programming (ESOP'12) (LNCS)*, Helmut Seidl (Ed.), Vol. 7211. Springer, Tallin, Estonia, 194–213.

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *International Colloquium on Automata, Languages and Programming (ICALP'13) (LNCS)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.), Vol. 7966. Springer, Riga, Latvia, 174–186.

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised multiparty session types. *Logical Methods in Computer Science* 8, 4 (2012).

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. 2010. Sessions and session types: An overview. In *International Workshop on Web Services and Formal Methods (WS-FM'09) (LNCS)*, Cosimo Laneve and Jianwen Su (Eds.), Vol. 6194. Springer, Bologna, Italy, 1–28.

Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2007. On progress for structured communications. In *Trustworthy Global Computing (TGC'07) (LNCS)*, Vol. 4912. Springer, Sophia-Antipolis, France, 257–275.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. 2009. Objects and session types. *Information and Computation* 207, 5 (2009), 595–641.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP'06) (LNCS)*, Vol. 4067. Springer, Nantes, France, 328–352.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys2006 (ACM SIGOPS)*. ACM Press, Leuven, Belgium, 177–190.

Luca Fossati, Raymond Hu, and Nobuko Yoshida. 2014. Multiparty session nets. In *Trustworthy Global Computing (TGC'14) (LNCS)*, Matteo Maffei and Emilio Tuosto (Eds.), Vol. 8902. Springer, Rome, Italy, 112–127.

Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. 2006. BASS: Boxed ambients with safe sessions. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*. ACM Press, Venice, Italy, 61–72.

Simon Gay. 2008. Bounded polymorphism in session types. *MSCS* 18 (2008), 895–930.

Simon Gay and Malcolm Hole. 2005. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica* 42, 2/3 (2005), 191–225.

Simon Gay and Vasco T. Vasconcelos. 2009. Linear type theory for asynchronous session types. *Journal of Functional Programming* (2009).

Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *Symposium on Principles of Programming Languages (POPL'10)*. ACM, Madrid, Spain, 299–312.

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50 (1987), 1–102.

Matthew Hennessy. 2007. *A Distributed Pi-Calculus*. Cambridge University Press.

Anders Henriksen, Lasse Nielsen, Thomas Hildebrandt, Nobuko Yoshida, and Fritz Henglein. 2013. Trustworthy pervasive healthcare services via multi-party session type. In *Foundations of Health Information Engineering and Systems (FHIES'12) (LNCS)*, Jens Weber and Isabelle Perseil (Eds.), Vol. 7789. Paris, France, 124–141.

Kohei Honda. 1993. Types for dyadic interaction. In *International Conference on Concurrency Theory (CONCUR'93) (LNCS)*, Eike Best (Ed.), Vol. 715. Springer-Verlag, Hildesheim, Germany, 509–523.

Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. 2014. Structuring communication with session types. In *Concurrent Objects and Beyond (COB'14) (LNCS)*, Gul A. Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura (Eds.), Vol. 8665. Springer, 105–127.

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling interactions with a formal foundation. In *International Conference on Distributed Computing and Internet Technology (ICDCIT'11) (LNCS)*, Raja Natarajan and Adegboyega K. Ojo (Eds.), Vol. 6536. Springer, Bhubaneswar, India, 55–75.

Kohei Honda and Mario Tokoro. 1991. An object calculus for asynchronous communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, Vol. 512. Geneva, Switzerland, 133–147.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP'98) (LNCS)*, Vol. 1381. Springer-Verlag, Lisbon, Portugal, 22–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2007. Web services, mobile processes and types. *The Bulletin of the European Association for Theoretical Computer Science*. February, 91 (2007), 165–185.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008a. Multiparty asynchronous session types. In *Symposium on Principles of Programming Languages (POPL'08)*. ACM, San Francisco, USA, 273–284.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008b. Multiparty Asynchronous Session Types. (2008). Web page. http://www.doc.ic.ac.uk/∼yoshida/multiparty.

Raymond Hu, Dimitrious Kouzapas, Oliver Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-safe eventful sessions in Java. In *European Conference on Object-Oriented Programming (ECOOP'10) (LNCS)*, Vol. 6183. Springer, Maribor, Slovenia, 329–353.

Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. 2013. Practical interruptible conversations: Distributed dynamic verification with session types and python. In *Runtime Verification (RV'13) (LNCS)*, Axel Legay and Saddek Bensalem (Eds.), Vol. 8174. Springer, Rennes, France, 148–130.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming (ECOOP'08)*, Jan Vitek (Ed.), Vol. 5142. Springer, Paphos, Cyprus, 516–541.

Atsushi Igarashi and Naoki Kobayashi. 2004. A generic type system for the Pi-calculus. *Theoretical Computer Science* 311, 1–3 (2004), 121–163.

International Telecommunication Union. 1996. Recommendation Z.120: Message Sequence Chart. (1996).

Naoki Kobayashi. 2006. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory (CONCUR'06) (LNCS)*, Vol. 4137. Bonn, Germany, 233–247.

Dimitrios Kouzapas, Jorge A. Perez, and Nobuko Yoshida. 2015. Characteristic bisimulations for higher-order session processes. In *International Conference on Concurrency Theory (CONCUR'15) (LIPIcs)*, Vol. 42. Schloss Dagstuhl, Madrid, Spain, 398–411.

Dimitrios Kouzapas and Nobuko Yoshida. 2014. Globally governed session semantics. *Logical Methods in Computer Science* 10, 4 (2014).

Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. 2016. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science* 26, 2 (2016), 303–364.

Pavel Krcál and Wang Yi. 2006. Communicating timed automata: The more synchronous, the more difficult to verify. In *Computer Aided Verification (CAV'06) (LNCS)*, Vol. 4144. Springer, Seattle, USA, 249–262.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (July 1978), 558–564.

Julien Lange and Emilio Tuosto. 2012. Synthesising choreographies from local session types. In *International Conference on Concurrency Theory (CONCUR'12) (LNCS)*, Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer, Newcastle upon Tyne, UK, 225–239.

Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From communicating machines to graphical choreographies. In *Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM Press, Mumbai, India, 221–232.

Massimo Merro. 2007. An observational theory for mobile Ad hoc networks. In *Electronic Notes in Theoretical Computer Science*, Vol. 172. Elsevier, 275–293.

Nicola Mezzetti and Davide Sangiorgi. 2006. Towards a calculus for wireless systems. In *Electronic Notes in Theoretical Computer Science*, Vol. 158. Elsevier, 331–353.

Leonardo Gaetano Mezzina. 2008. How to infer finite session types in a calculus of services and sessions. In *Coordination Models and Languages (COORDINATION'08) (LNCS)*, Vol. 5052. Springer, Oslo, Norway, 216–231.

Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional choreographies. In *International Conference on Concurrency Theory (CONCUR'13) (LNCS)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.), Vol. 8052. Springer, Buenos Aires, Argentina, 439–425.

Dimitris Mostrous and Nobuko Yoshida. 2007. Two session typing systems for higher-order mobile processes. In *Typed Lambda Calculi and Applications (TLCA'07) (LNCS)*, Vol. 4583. Springer, Paris, France, 321–335.

Dimitris Mostrous and Nobuko Yoshida. 2009. Session-based communication optimisation for higher-order mobile processes. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009 (LNCS)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, Brasilia, Brazil, 203–218.

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global principal typing in partially commutative asynchronous sessions. In *European Symposium on Programming (ESOP'09) (LNCS)*, Vol. 5502. Springer, York, UK, 316–332.

Sebastian Nanz, Flemming Nielson, and Hanne Riis Nielson. 2007. Topology-dependent abstractions of broadcast networks. In *International Conference on Concurrency Theory (CONCUR'07)*. Lisbon, Portugal, 226–240.

Matthias Neubauer and Peter Thiemann. 2004a. An implementation of session types. In *Practical Aspects of Declarative Languages (PADL'04) (LNCS)*, Vol. 3057. Springer, Dallas, USA, 56–70.

Matthias Neubauer and Peter Thiemann. 2004b. *Session Types for Asynchronous Communication*. (2004). Universität Freiburg.

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2014. Timed runtime monitoring for multiparty conversations. In *Workshop on Behavioural Types (BEAT'14) (EPTCS)*, Marco Carbone (Ed.), Vol. 162. Rome, Italy, 19–26.

Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty session actors. In *Coordination Models and Languages (COORDINATION'14) (LNCS)*, Eva Kühn and Rosario Pugliese (Eds.), Vol. 8459. Springer, Berlin, Germany, 131–146.

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local verification of global protocols. In *Runtime Verification (RV'13) (LNCS)*, Axel Legay and Saddek Bensalem (Eds.), Vol. 8174. Springer, Rennes, France, 363–358.

Nicholas Ng, Jose G. F. Coutinho, and Nobuko Yoshida. 2015. Protocols by default: Safe MPI code generation based on session types. In *Compiler Construction (CC'15) (LNCS)*. Björn Franke (Ed.). Springer, London, UK, 212–232.

Nicholas Ng and Nobuko Yoshida. 2014. Pabble: Parameterised scribble. *Service Oriented Computing and Applications* 9, 3–4 (2014), 1–16.

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty session C: Safe parallel programming with message optimisation. In *TOOLS'12 (LNCS)*, Carlo A. Furia and Sebastian Nanz (Eds.), Vol. 7304. Springer, Prague, Czech Republic, 202–218.

Nicholas Ng, Nobuko Yoshida, and Wayne Luk. 2013. Scalable session programming for heterogeneous high-performance systems. In *International Conference on Software Engineering and Formal Methods (SEFM'13) (LNCS)*, Steve Counsell and Manuel Núñez (Eds.), Vol. 8368. Springer, Madrid, Spain, 82–98.

Nicholas Ng, Nobuko Yoshida, Xin Yu Niu, Kuen Hung Tsoi, and Wayne Luk. 2012. Session types: Towards safe and fast reconfigurable programming. *SIGARCH CAN* 40, 5 (2012), 22–27.

Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. 2011. Safe parallel programming with session Java. In *Coordination Models and Language (COORDINATION'11) (LNCS)*, Vol. 6721. Springer, Reykjavik, Iceland, 110–126.

Lasse Nielsen, Nobuko Yoshida, and Kohei Honda. 2010. Multiparty symmetric sum types. In *Expressiveness in Concurrency (EXPRESS'10) (EPTCS)*, Vol. 41. Paris, France, 121–135.

OOI. 2015. Ocean Observatories Initiative. Retrieved from http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/.

Luca Padovani. 2014a. Deadlock and lock freedom in the linear π-calculus. In *Computer Science Logic and Logic in Computer Science (CSL-LICS'14)*, Thomas A. Henzinger and Dale Miller (Eds.). ACM Press, Vienna, Austria, 72:1–72:10.

Luca Padovani. 2014b. Fair subtyping for multi-party session types. *Mathematical Structures in Computer Science* (2014), 1–41.

B. Pierce and D. Sangiorgi. 1996. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science* 6, 5 (1996), 409–454.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Jérémy Planul, Ricardo Corin, and Cédric Fournet. 2009. Secure enforcement for global process specifications. In *International Conference on Concurrency Theory (CONCUR'09) (LNCS)*, Mario Bravetti and Gianluigi Zavattaro (Eds.), Vol. 5710. Springer, Bologna, Italy, 511–526.

K. V. S. Prasad. 2001. Broadcast calculus interpreted in CCS upto bisimulation. In *Electronic Notes in Theoretical Computer Science* 52, 1, 83–100.

K. V. S. Prasad. 2006. A prospectus for mobile broadcasting systems. In *Electronic Notes in Theoretical Computer Science* 162, 1, 295–300.

Riccardo Pucella and Jesse Tov. 2008. Haskell session types with (almost) no class. In *Haskell Symposium (Haskell'08)*. ACM SIGPLAN, Victoria, Canada.

Matthew Sackman and Susan Eisenbach. 2008. Session Types in Haskell. draft.

SAVARA. 2010. SAVARA JBoss Project. Retrieved from http://www.jboss.org/savara.

Bruce Schneier. 1993. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc.

Scribble. 2008. Scribble Project. Retrieved from www.scribble.org.

K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. 2010. Efficient session type guided distributed interaction. In *Coordination Models and Languages (COORDINATION'10) (LNCS)*, Vol. 6116. Springer, Amsterdam, Holland, 152–167.

Stephen Sparkes. 2006. Conversation with steve ross-talbot. *ACM Queue* 4, 2 (March 2006).

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP'11)*. IEEE, Tokyo, Japan, 266–278.

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe (PARLE'94) (LNCS)*, Vol. 817. Springer-Verlag, Athens, Greece, 398–413.

F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. 1999. Strand spaces: Proving security protocols correct. *Journal of Computer Security* 7, 2/3 (1999), 191–230.

Vasco T. Vasconcelos, Simon Gay, and António Ravara. 2006. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science* 368, 1–2 (2006), 64–87.

Hugo Torres Vieira, Luís Caires, and João Costa Seco. 2008. The conversation calculus: A model of service-oriented computation. In *European Symposium on Programming (ESOP'08) (LNCS)*, Vol. 4960. Springer, Budapest, Hungary, 269–283.

Jules Villard. 2011. *Heaps and Hops*. Ph.D. Dissertation. ENS Cachan.

Phil Wadler. 2012. Proposition as sessions. In *International Conference on Functional Programming (ICFP'12)*. IEEE, Copenhagen, Denmark, 273–286.

WS-CDL. 2003. Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/. (2003).

Nobuko Yoshida. 1996. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96) (LNCS)*, Vol. 1180. Springer, Hyderabad, India, 371–386.

Nobuko Yoshida, Martin Berger, and Kohei Honda. 2001. Strong normalisation in the π-Calculus. In *Proc. LICS'01*. IEEE, 311–322. The full version in *Journal of Information and Computation,* 191 (2004), 145–202, Elsevier.

Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The scribble protocol language. In *Trustworthy Global Computing (TGC'13) (LNCS)*, Martín Abadi and Alberto Lluch-Lafuente (Eds.), Vol. 8358. Springer, Buenos Aires, Argentina, 22–41.

Nobuko Yoshida and Vasco Thudichum Vasconcelos. 2007. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes on Theoretical Computer Science* 171, 4 (2007), 73–93.

Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. 2008. Session-based compilation framework for multicore programming. In *International Symposium on Formal Methods for Components and Objects (FMCO'08) (LNCS)*, Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain (Eds.), Vol. 5751. Springer, Sophia Antipolis, France, 226–246.

ZDLC. 2015. Zero Deviation Lifecycle. Retrieved from http://www.zdlc.co.

Received January 2009; revised February 2013 and August 2015; accepted September 2015