Multiparty Asynchronous Session Types

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone
Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

Abstract

Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing 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-centred 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 a friendly type syntax of binary session types while 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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Process models

General Terms Theory, Types, Design

Keywords communications, multiparty, structured programming, session types, mobile processes, causality, choreography

1. Introduction

Backgrounds Communication is becoming one of the central elements in software development, ranging from web services to business protocols to parallel scientific computing to multi-core programming. As a potential typed foundation for structured communication-centred programming, session types have been studied in many contexts over the last decade, including calculi of mobile processes (Takeuchi et al. 1994; Gay and Hole 2005; Honda et al. 1998; Bonelli and Compagnoni 2008), higher-order processes (Mostrous and Yoshida 2007), Ambients (Garralda et al. 2006), multi-threaded ML (Vasconcelos et al. 2006), Haskell (Neubauer and Thiemann 2004b), F# (Corin et al. 2007), operating systems (Fähndrich et al. 2006), Java (Dezani-Ciancaglini et al. 2006; Coppo et al. 2007; Hu et al. 2007), and Web Services (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or *session*. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and 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, then sends his address (a string) and Seller sends back the delivery date (a date); otherwise it quits the conversation.

$$!\mathsf{string}; ?\mathsf{int}; \oplus\{\mathsf{ok} : !\mathsf{string}; ?\mathsf{date}; \mathsf{end}, \quad \mathsf{quit} : \mathsf{end}\} \qquad (1)$$

Above $!\mathsf{t}$ denotes an output of a value of type t, dually for $?\mathsf{t}$; $\oplus$ denotes a choice of the options; and end represents the termination of the conversation.

Such explicit representation of conversation structures helps us deal with one of the most common bugs in programming with communication, the synchronisation bugs. A programmer expects that communicating programs should together realise a consistent conversation, but they 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 (1) guides principled programming of communication behaviour and enables automatic protocol validation (WS-CDL; UNIFI; Hu et al. 2007). 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. 2007). 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).

Multiparty Asynchronous Sessions The foregoing studies on session types have focussed on binary (two-party) sessions. While 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 which involve more than two parties.

As an example, let us consider a simple refinement of the above Buyer-Seller protocol: consider two buyers, Buyer1 and Buyer2, 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'08, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001…$5.00

much she can pay, and Buyer2 either accepts the quote or receives 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 usecases are most naturally abstracted as a single session. 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 synchronisation. The widely used network transport, such as TCP, provides this mechanism through familiar APIs to alleviate the latency problem. Thus we ask: can we generalise the foregoing binary session types to multiparty asynchronous sessions preserving clarity and their key formal properties? This question was repeatedly posed by not only researchers but also the members of a W3C working group (WS-CDL) through our collaboration as invited experts (Honda et al. 2007a; Carbone et al. 2006, 2007), because of urgent need for a theoretical basis to validate a wide range of business protocols.

Challenges of Multiparty Asynchronous Sessions To answer this open question, we face two major technical difficulties. First, simplicity and tractability of the theory of binary sessions come from a notion of *duality* in interactions (Girard 1987). Consider the binary session type given in (1) for Buyer. Not only Buyer's behaviour can 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 they have mutually dual types. This framework based on duality is no longer effective in multiparty sessions where the whole conversation cannot be constructed from only single behaviour. We need an effective means to abstract as a type a global scenario which a programmer wishes to realise 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 to ensure safety and progress, becomes highly involved under a combination of asynchrony and multiparty since a conflict of actions can arise more easily. This demands a precise causal analysis for correct sequencing of interactions distributed among multi-peers.

This Work. This paper presents a generalisation of binary session types to multiparty sessions for the π-calculus. We overcome the aforementioned challenges with the following three technical apparatus:

1. A new notion of types which can directly abstract intended conversation structure among n-parties as *global scenarios*, retaining intuitive type syntax.
2. Consistency criteria for a conversation structure given as a causality analysis of actions in global types, modularly articulating different kinds of dependency.
3. A type discipline for individual processes (programs) which uses a global type through its *projection* onto individual local participants: the resulting local types are directly associated with individual processes for efficient 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). 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 can stipulate a new effective method for designing and type-checking multiparty sessions (Point 3). First, we design a global type G as an intended scenario. A team of programmers then develop code, one for each participant, incrementally validating its conformance to (the projection of) G. When programs are executed, their interactions automatically follow the stipulated scenario. For materialising this design framework, we propose a type discipline which can validate whether a program is typable or not, given G (as shared agreement) and an individual program (as its local realiser). The resulting type discipline guarantees all the original key properties, such as communication error freedom, progress and fidelity in a session among multiparty.

In the remainder, Section 2 gives the syntax and semantics of the calculus, and motivates the key ideas through business and streaming protocol examples. Section 3 explains the global types. Section 4 describes the typing system. Section 5 establishes the main results. Section 6 gives extensions and related works. Section 7 concludes with future issues. The omitted definitions, proofs and large size of examples are left in (Honda et al. 2007b).

2. Multiparty Asynchronous Sessions

2.1 Syntax for Multiparty Sessions

Several versions of the π-calculi with session types are proposed in the literature; the paper (Yoshida and Vasconcelos 2007) offers detailed discussions and analysis of their typing systems. We use a simple extension of the original language in (Honda et al. 1998; Takeuchi et al. 1994) to multiparty sessions.

Informally, a *session* is a series of interactions which 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 through which a series of communication actions are performed.

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 $s,t,\ldots$; *labels*, ranged over by $l,l',\ldots$; and *process variables*, ranged over by $X,Y,\ldots$. In the syntax for hiding, we use n for either a single shared name or a vector of session channels. 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 (Honda et al. 1998). The prefix $\overline{a}[\mathtt{2..n}](\tilde{s}).P$ 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 $2 \le \mathtt{p} \le \mathtt{n}$. All receive $\tilde{s}$, over which the actual session communications can now take place among the $\mathtt{n}$ parties. $\mathtt{p}$, $\mathtt{q}$,... range over natural numbers called *participants* of a session.

Session communications are performed using the next three pairs of primitives: the sending and receiving, the session delegation and reception (the former delegates to the latter the capability to participate in a session by passing the whole channels associated with the session), and the selection and branching (the former chooses one of the branches offered by the latter). The next three (the conditional, parallel and inaction) are standard. $(\nu a)P$ makes a local to P while $(\nu \tilde{s})P$ makes $\tilde{s}$ local to P. The recursion and process call realise recursive behaviour. $s:\tilde{h}$ is a *message queue* representing ordered messages in transit $\tilde{h}$ with destination s (which may be considered as a network pipe in a TCP-like transport). $(\nu \tilde{s})P$ and

Figure 1 Syntax

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

$s:\tilde{h}$ only appear at runtime. We often omit trailing $\mathbf{0}$ and write $s!$ and $s?.P$, omitting the arguments if unnecessary.

Binders are $\tilde{s}$ in $\overline{a}[2..n](\tilde{s}).P$, $a[p](\tilde{s}).P$ 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 $(\nu n)P$ and process variables in def D in P. The notions of bound and free identifiers, channels, alpha equivalence $\equiv_\alpha$ and substitution are standard. $\text{fpv}(P)$ and $\text{fn}(P)$, respectively denote the sets of *free process variables* and *free identifiers* in P. $\text{dpv}(\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I})$ denotes the set of *process variables* $\{X_i\}_{i\in I}$ introduced in $\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$. A sequence of parallel composition is written $\Pi_i P_i$.

2.2 Operational Semantics

Structural congruence is the smallest congruence relation on processes that includes the equations in Figure 2. 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.

[Link] describes a session initiation among n-parties through synchronisation, generating m fresh session channels and the associated m empty queues ($\emptyset$ denotes the empty string). As a result n participants now share the newly generated m channels, hence their queues. Note the number of threads (n) can be different from that of session channels (m), giving flexibility in channel usage.

[Send], [Deleg] and [Label] respectively enqueue values, channels and a label at the tail of the queue for s. [Recv], [SRec] [1] and [Branch] dequeue, at the head of the queue, values, channels and a label. [Branch] further selects the corresponding branch. Since [Link] provides a queue for each channel, these rules say that a sending action is never blocked (asynchrony) and that two messages from the same sender to the same channel arrive in the sending order (order preservation). Other rules are standard.

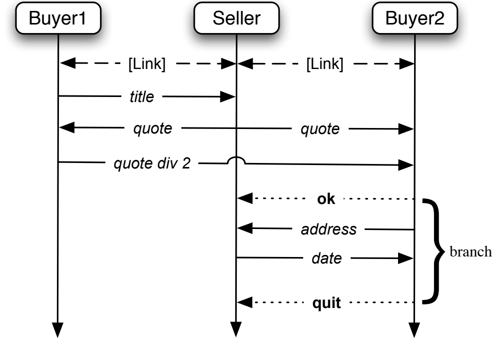
[1] This delegation rule (which is from (Honda et al. 1998)) is chosen over the more liberal one in (Gay and Vasconcelos 2007; Yoshida and Vasconcelos 2007) (which uses substitution as in [Recv]) for simpler presentation. The technical development does not depend on this choice, see §6.2.

Figure 2 Structural congruence.

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \text{fn}(Q) \qquad (\nu nn')P \equiv (\nu n'n)P$$
$$(\nu n)\mathbf{0} \equiv \mathbf{0} \qquad \text{def } D \text{ in } \mathbf{0} \equiv \mathbf{0} \qquad (\nu s_1..s_n)\Pi_i s_i : \emptyset \equiv \mathbf{0}$$
$$\text{def } D \text{ in } (\nu n)P \equiv (\nu n)\text{def } D \text{ in } P \quad \text{if } n \notin \text{fn}(D)$$
$$(\text{def } D \text{ in } P) \mid Q \equiv \text{def } D \text{ in } (P \mid Q) \quad \text{if } \text{dpv}(D) \cap \text{fpv}(Q) = \emptyset$$
$$\text{def } D \text{ in } (\text{def } D' \text{ in } P) \equiv \text{def } D \text{ and } D' \text{ in } P \quad \text{if } \text{dpv}(D) \cap \text{dpv}(D') = \emptyset$$

2.3 Examples

Two Buyer Protocol We describe the two-buyers-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 Buyer1/2; Buyer1 now tells Buyer2 how much she can contribute, and Buyer2 notifies Seller if it accepts the quote or not. We now describe the behaviour of Buyer1 as a process:

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

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 above 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.

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

$$\begin{aligned}\text{Seller} \stackrel{\text{def}}{=} \; & a[3](b_1,b_2,b_2',s).\;\; s?(title);\;\; b_1,b_2!\langle quote\rangle; \\ & s \rhd \{\text{ok}:\; s?(x); b_2!\langle date\rangle;Q, \quad \text{quit}:\; \mathbf{0}\}\end{aligned}$$

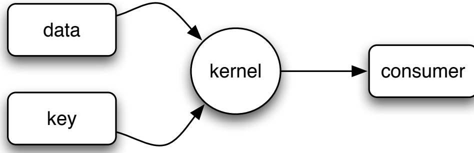
Above $s_1..s_m!\langle v\rangle;P$ stands for $s_1!\langle v\rangle;..s_m!\langle v\rangle;P$, assuming $s_1..s_m$ are pairwise distinct.[2] We can now explain why Buyer2 needs to use two input channels, b_2 and b_2'. The first input (for *quote*) is from Seller, while the second one (for *contrib*) is from Buyer1. Hence there is no guarantee that they arrive in a fixed order, as can be easily seen by analysing 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. Later we shall show our type discipline can detect such an error.

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

Figure 3 Reduction

$$\overline{a}_{[2..n]}(\tilde{s}).P_1 \mid a_{[2]}(\tilde{s}).P_2 \mid \cdots \mid a_{[n]}(\tilde{s}).P_n \qquad \text{[Link]}$$
$$\to (\nu\, \tilde{s})(P_1 \mid P_2 \mid ... \mid P_n \mid s_1 : \emptyset \mid \cdots \mid s_m : \emptyset)$$
$$s!\langle \tilde{e} \rangle; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot \tilde{v} \quad (\tilde{e} \downarrow \tilde{v}) \qquad \text{[Send]}$$
$$s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot \tilde{t} \qquad \text{[Deleg]}$$
$$s \lhd l; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot l \qquad \text{[Label]}$$
$$s?(\tilde{x}); P \mid s : \tilde{v} \cdot \tilde{h} \to P[\tilde{v}/\tilde{x}] \mid s : \tilde{h} \qquad \text{[Recv]}$$
$$s?(\!(\tilde{t})\!); P \mid s : \tilde{t} \cdot \tilde{h} \to P \mid s : \tilde{h} \qquad \text{[SRec]}$$
$$s \rhd \{l_i : P_i\}_{i \in I} \mid s : l_j \cdot \tilde{h} \to P_j \mid s : \tilde{h} \quad (j \in I) \qquad \text{[Branch]}$$
$$\text{if } e \text{ then } P \text{ else } Q \to P \quad (e \downarrow \text{true}) \qquad \text{[IfT]}$$
$$\text{if } e \text{ then } P \text{ else } Q \to Q \quad (e \downarrow \text{false}) \qquad \text{[IfF]}$$
$$\text{def } D \text{ in } (X\langle \tilde{e}\tilde{s} \rangle \mid Q) \to \text{def } D \text{ in } (P[\tilde{v}/\tilde{x}] \mid Q) \qquad \text{[Def]}$$
$$(\tilde{e} \downarrow \tilde{v}, X(\tilde{x}\tilde{s}) = P \in D)$$
$$P \to P' \Rightarrow (\nu n)P \to (\nu n)P' \qquad \text{[Scop]}$$
$$P \to P' \Rightarrow P \mid Q \to P' \mid Q \qquad \text{[Par]}$$
$$P \to P' \Rightarrow \text{def } D \text{ in } P \to \text{def } D \text{ in } P' \qquad \text{[Defin]}$$
$$P \equiv P' \text{ and } P' \to Q' \text{ and } Q' \equiv Q \Rightarrow P \to Q \qquad \text{[Str]}$$

A 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 realise this protocol as communicating processes. We only focus on communication behaviour. The kernel initiates a session:

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

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:

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

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

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

Figure 4 Syntax of Global Types

Global	G	::=	$\mathtt{p} \to \mathtt{p}' : k\langle U \rangle.G'$	values
		|	$\mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j \in J}$	branching
		|	G, G'	parallel
		|	$\mu \mathbf{t}.G$	recursive
		|	$\mathbf{t}$	variable
		|	end	end
Value	U	::=	$\tilde{S} \mid T@\mathtt{p}$	
Sort	S	::=	bool | nat | ... | $\langle G \rangle$	

3. Global Types and Causal Analysis

Developing programs for multiparty sessions demands a clear formal design as to how multiple participants communicate and synchronise with each other. To program individual participants without such a design and hope they somehow realise a meaningful and error-free conversation is hardly practical, especially for team programming. In binary session types the type for an endpoint also served as the description of the whole conversation, but this is no longer possible for multiparty sessions. This is why we need the type abstraction which describes global conversation scenarios of multiparty sessions, introduced in this section.

3.1 Session Types from a Global Viewpoint

The grammar of *global session type*, or *global type*, denoted $G, G', \ldots$, is given in Figure 4. Type $\mathtt{p} \to \mathtt{p}' : k\langle U \rangle.G'$ says that participant $\mathtt{p}$ sends a message of type U to channel k (represented as a finite natural number) received by participant $\mathtt{p}'$: and interactions described in G' takes place. $U, ...$ range over *value types*, denoting types for message values. Each value type is a vector of types for shared names called *sorts*, written $S, S', \ldots$, or of those for session channels. Both of these types are discussed in detail in § 4.2. For understanding this section, it suffices to consider U as a single base type. Type $\mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j \in J}$ says participant $\mathtt{p}$ sends one of the labels to channel k which is then received by participant $\mathtt{p}'$. If l_j is sent, interactions described in G_j take place.

Type G, G' represents concurrent run of interactions specified by 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. We identify "G, end" and "end, G" with G.

Definition 3.1 (prefix). We say the initial "$\mathtt{p} \to \mathtt{p}' : k$" in $\mathtt{p} \to \mathtt{p}' : k\langle U \rangle.G'$ and $\mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j \in J}$ is a *prefix from* $\mathtt{p}$ *to* $\mathtt{p}'$ *at* k *over* G' where in the former U is a *carried type*. If U is a carried type in a prefix in G then U is also a carried type in G.

Conventions 3.2. We assume that in each prefix from $\mathtt{p}$ to $\mathtt{p}'$ we have $\mathtt{p} \neq \mathtt{p}'$, i.e. we prohibit reflexive interaction.

Henceforth we often regard a global type G as the acyclic directed graph given by its standard regular tree presentation (Pierce 2002). A basic ordering on its nodes is induced by prefixes.

Definition 3.3 (prefix ordering). Write $\mathtt{n}, \mathtt{n}', ..$ for prefixes occurring in a global type, say G (but not in its carried types), seen as nodes of G as a graph. We write $\mathtt{n} \in G$ when $\mathtt{n}$ occurs in G. Then

[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 $\langle G \rangle$.

Figure 5 Causality Analysis

(II) Good	(II) Bad	(IO) Good	(IO) Bad	(OO, II) Good	(OI) Bad
$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{A} \to \mathtt{B} : s$
$\mathtt{C} \to \mathtt{B} : t$	$\mathtt{C} \to \mathtt{B} : s$	$\mathtt{B} \to \mathtt{C} : t$	$\mathtt{B} \to \mathtt{C} : s$	$\mathtt{A} \to \mathtt{B} : s$	$\mathtt{C} \to \mathtt{A} : s$
$s! \mid s?;t? \mid t!$	$s! \mid s?;s? \mid s!$	$s! \mid s?;t! \mid t?$	$s! \mid s?;s! \mid s?$	$s!;s! \mid s?;s?$	$s!;s? \mid s? \mid s!$

we write $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ when n_1 directly or indirectly prefixes n_2 in G. Formally $\prec$ is the least partial order including:

$$\begin{array}{ll} \mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k\langle U\rangle.G' & \text{if } \mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k, \mathsf{n}_2 \in G' \\ \mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j\in J} & \text{if } \mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k, \exists i \in J. \mathsf{n}_2 \in G_i \end{array}$$

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

The prefix ordering allows us to express intended sequencing in global types. To clarify its meaning is essential for its proper usage. Consider a global type:

$$\mathtt{A} \to \mathtt{B} : s\langle U\rangle.\ \mathtt{A} \to \mathtt{C} : t\langle U'\rangle.\ \mathsf{end} \qquad (2)$$

The two prefixes are ordered by $\prec$. In a "synchronous" interpretation, this 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 calculi, but *not* with asynchronous communication, where it is hard to impose this ordering on (2), since messages to distinct channels may not arrive in order.

Thus the present theory takes the more liberal interpretation of $\prec$, imposing sequencing *only on the actions of the same participant in ordered prefixes*. For example, in (2), A's two sending actions are ordered, but B's and C's receiving actions are not. The remaining causal ordering comes from communication *à la* Lamport (Lamport 1978). Let us further illustrate this idea with examples.

3.2 Examples of Global Types

The following is a global type of the two-buyer-protocol in §2.3. We write principals and channels with legible symbols though they are actually numbers: $\mathsf{Bi} = i$, $\mathsf{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$ and $s = 4$.

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

The type gives a vantage view of the whole conversation scenario. We show several salient points in the 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 can have a conflict at s.* Note this analysis echoes our operational argument in §2.3.
- Next we consider the following causal chain of actions from Line 1 to Line 3 to Line 5:

  $$\mathtt{B1} \to \mathtt{S} \prec \mathtt{S} \to \mathtt{B2} \prec \mathtt{B2} \to \mathtt{S}$$

  Above $\to$ denotes the ordering given by message delivery, while $\prec$ is the prefix ordering. Note in particular two sending actions by B1 (Line 1) and by B2 (Line 5), both done at s, are causally ordered. By focussing on $\prec$ from the first S (of Line 1) to the last S (of Line 5), the receiving actions in Lines 1 and 5 are also ordered. Since both sending and receiving take place in strict temporal order, no conflict occurs between these two communications in spite of their use of a common channel s.

Next we present the global type of the simple streaming protocol in §2.3. Below we unfold its recursion once, and set: $d = 1$, $k = 2$, $c = 3$, $\mathtt{K} = 1$, $\mathtt{DP} = 2$, $\mathtt{C} = 3$ and $\mathtt{KP} = 4$.

$$\begin{array}{llll} 1 & \mu\mathbf{t}.\ \ \mathtt{DP} \to \mathtt{K} : d\langle \mathsf{bool}\rangle. & 4 & \mathtt{DP} \to \mathtt{K} : d\langle \mathsf{bool}\rangle. \\ 2 & \quad\ \mathtt{KP} \to \mathtt{K} : k\langle \mathsf{bool}\rangle. & 5 & \mathtt{KP} \to \mathtt{K} : k\langle \mathsf{bool}\rangle. \\ 3 & \quad\ \mathtt{K} \to \mathtt{C} : c\langle \mathsf{bool}\rangle. & 6 & \mathtt{K} \to \mathtt{C} : c\langle \mathsf{bool}\rangle.\mathbf{t} \end{array}$$

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.

3.3 Safety Principle for Global Types

For a conversation in a session to proceed properly, it is desirable that there is no conflict (racing) at session channels. To ensure this, when a *common* channel is used in two communications, their sending actions and their receiving actions should respectively be ordered temporally, so that no confusion arises at neither sending nor receiving. If a global type satisfies this principle, then it specifies a safe protocol, and can be used as a basis of guaranteeing safe process behaviours through type checking.

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. IO denotes the causal ordering by $\prec$ is 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 shows that II alone is safe only when two channels differ. Similarly for IO.
- In OO,II (the fifth case), 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 "effective" causal relations on global types.

Definition 3.4. (dependency relations) Fix G. The relation $\prec_\phi$, with $\phi \in \{\mathsf{II}, \mathsf{IO}, \mathsf{OO}\}$, over its prefixes is generated from:

$$\begin{array}{ll} \mathsf{n}_1 \prec_{\mathtt{II}} \mathsf{n}_2 & \text{if } \mathsf{n}_1 \prec \mathsf{n}_2 \text{ and } \mathsf{n}_i = \mathtt{p_i} \to \mathtt{p} : k_i \ (i = 1, 2) \\ \mathsf{n}_1 \prec_{\mathtt{IO}} \mathsf{n}_2 & \text{if } \mathsf{n}_1 \prec \mathsf{n}_2, \mathsf{n}_1 = \mathtt{p_1} \to \mathtt{p} : k_1 \text{ and } \mathsf{n}_2 = \mathtt{p} \to \mathtt{p_2} : k_2. \\ \mathsf{n}_1 \prec_{\mathtt{OO}} \mathsf{n}_2 & \text{if } \mathsf{n}_1 \prec \mathsf{n}_2, \mathsf{n}_i = \mathtt{p} \to \mathtt{p_i} : k \ (i = 1, 2) \end{array}$$

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 $\phi_i \in \{\mathsf{II}, \mathsf{IO}\}$ 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 $\phi_i \in \{\mathsf{OO}, \mathsf{IO}\}$.

In the input dependency, the last II-ordering is needed since if it ends with an IO-edge an input at n_2 may not be suppressed.

Definition 3.5. (linearity) G is *linear* if, whenever $\mathsf{n}_i = \mathtt{p_i} \to \mathtt{p_i'} : k$ $(i = 1, 2)$ are in G for some k and do not occur in different branches of a branching, then both input and output dependencies exist from

Figure 6 Syntax of Local Types

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

n_1 to n_2, or, if not, both exist from n_2 to n_1. If G carries other global types, we inductively demand the same.

We illustrate the condition on branching by an example:

	(a) branching	(b) parallel
1.	$\mathtt{A} \to \mathtt{B} : t\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end}$	$\mathtt{A} \to \mathtt{B} : t.(\mathtt{C} \to \mathtt{D} : s.\mathsf{end},$
2.	$\mathsf{quit} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end}\ \}$	$\mathtt{C} \to \mathtt{D} : s.\mathsf{end})$

The type (a) represents branching: since only one of two branches is selected, there is no conflict between the two prefixes $\mathtt{C} \to \mathtt{D} : s$ in Lines 1 and 2. On the other hand, (b) means 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 unfoldings. First we observe we do need to unfold once.

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

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

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

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

Proposition 3.6. *(1) A global type is linear iff its one-time unfolding is linear. (2) The linearity of a global type is decidable.*

(2) is an immediate corollary of (1). For (1), we show if one-time unfolding is linear then each n-th folding is linear by induction on n, using tail recursiveness of the present global types.

4. Type Discipline for Multiparty Sessions

4.1 Programming Methodology for Multiparty Interactions

Once given global types as our tool, 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 each participant, incrementally validating its conformance to the projection of G onto each participant by efficient type-checking.

When programs are executed, their interactions are guaranteed to follow the stipulated scenario. The type specification also serves as a basis for maintenance and upgrade. This section introduces the type discipline which materialises this framework.

4.2 Local Types

Syntax *Local session types* or *local types*, ranged over by $T, T', ...$, are types for local behaviour of processes, acting as a link between global types and processes. The grammar is given in Figure 6 (the grammars for U and S are repeated from Figure 4). All constructs come from the binary session types (Honda et al. 1998) except for the following major changes for multiparty interactions.

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

Type $k?\langle U \rangle;T$ represents the behaviour of inputting values of type U at s_k (assume $s_1...s_n$ is shared at initialisation), 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: it waits with n options at k, and behave as type T_i if i-th label is selected; type $k \oplus \{l_i : T_i\}_{i \in I}$ represents the behaviour which selects one of the labels say l_i at k then behaves as T_i. The rest is the same as the global types, demanding type variables occur guarded by a prefix and taking an equi-recursive approach for recursive types. We often omit end. Note local type T does not contain parallel composition.

In addition to the folding/unfolding of recursive types, local types are considered up to the following isomorphism (closed under all type constructors). We assume $k \neq k'$, $m \in I$ and $n \in J$.

$$k!\langle U \rangle;k'!\langle U' \rangle;T \quad \approx \quad k'!\langle U' \rangle;k!\langle U \rangle;T \tag{3}$$

$$k \oplus \{l_i : k' \oplus \{l'_j : T_{ij}\}_{j \in J}\}_{i \in I} \quad \approx \quad k' \oplus \{l'_j : k \oplus \{l_i : T_{ij}\}_{i \in I}\}_{j \in J} \tag{4}$$

The equations permute two consecutive outputs with different subjects, capturing asynchrony in communication. The equation (4) specialises to permutation between selection and output by setting I or J a singleton: and to (3) when both are singletons.

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

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

- $(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G') \upharpoonright \mathsf{p} = \begin{cases} k!\langle U \rangle.(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{p}_1 \neq \mathsf{p}_2 \\ k?\langle U \rangle.(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{p}_2 \neq \mathsf{p}_1 \\ (G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \neq \mathsf{p}_2 \text{ and } \mathsf{p} \neq \mathsf{p}_1 \end{cases}$
- $(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j \in J}) \upharpoonright \mathsf{p} = \begin{cases} \oplus\{l_j : (G_j \upharpoonright \mathsf{p})\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \neq \mathsf{p}_2 \\ \&\{l_j : (G_j \upharpoonright \mathsf{p})\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \neq \mathsf{p}_1 \\ (G_1 \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \neq \mathsf{p}_2 \text{ and } \mathsf{p} \neq \mathsf{p}_1 \\ & \text{and } \forall i, j \in I.G_i \upharpoonright \mathsf{p} = G_j \upharpoonright \mathsf{p} \end{cases}$
- $(G_1, G_2) \upharpoonright \mathsf{p} = \begin{cases} G_i \upharpoonright \mathsf{p} & \text{if } \mathsf{p} \in G_i \text{ and } \mathsf{p} \notin G_j,\ i \neq j \in \{1,2\} \\ \mathsf{end} & \text{if } \mathsf{p} \notin G_1 \text{ and } \mathsf{p} \notin G_2 \end{cases}$
- $(\mu \mathbf{t}.G) \upharpoonright \mathsf{p} = \mu \mathbf{t}.(G \upharpoonright \mathsf{p})$, $\mathbf{t} \upharpoonright \mathsf{p} = \mathbf{t}$, and $\mathsf{end} \upharpoonright \mathsf{p} = \mathsf{end}$.

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

The mapping is intuitive. We regard the map to act on the syntax of global types. In the branching, all projections should generate an identical local type (otherwise undefined); and in parallel composition, p should be contained in at most a single type, ensuring each type is single-threaded. Below $\mathsf{pid}(G)$ denotes the set of participant numbers occurring in G (but not in carried types). In (2) $T_{\mathsf{p}}@\mathsf{p}$ appeared at the beginning of §4.2.

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

THEOREM 4.3. *Coherence of G is decidable.*

We also believe the coherence of $\{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}}$ is decidable but this is not necessary for the present technical development. Without projectability, a global type is not consistent. Linearity guarantees linear channel usage including message-order preservation. The next examples demonstrate the need of these conditions.

Examples of Coherence The following global type is linear but *not* coherent because the projection is undefined.

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

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 above nat to bool as: $\mathtt{A} \to \mathtt{B} : k\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool}\rangle, \mathsf{quit} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool}\rangle\}$, we can define the coherent projection as follows:

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

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

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

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.

4.3 Typing System

The purpose of the typing system introduced below is to efficiently type behaviours *which are built by programmers* hence which do not include runtime elements such as queues.

DEFINITION 4.4 (program phrase and program). A process P is a *program phrase* if P has no queues and no ν-bound session channels. P is a *program* if P is a program phrase in which no free session channels and process variables occur.

All of Buyer1, Buyer2, Seller, Data Producer, etc. in §2.3 are programs, hence are also program phrases.

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:

$$\begin{array}{rcl} \Gamma & ::= & \emptyset \mid \Gamma, u : S \mid \Gamma, X : \tilde{S}\tilde{T} \\ \Delta & ::= & \emptyset \mid \Delta, \tilde{s} : \{T@\mathtt{p}\}_{\mathtt{p}\in I} \end{array}$$

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to sorts and from process variables to sequences of sorts and types. *Typing* $(\Delta, \Delta', ..)$ records linear usage of session channels. In the binary sessions, it assigned a type to a single channel; now it assigns a family of located types to a vector of session channels. $\mathsf{sid}(G)$ stands for the set of session channel numbers in G. We write $\tilde{s} : T@\mathtt{p}$ for a singleton typing $\tilde{s} : \{T@\mathtt{p}\}$. Below we define a simple algebra of types for typing program phrases.

DEFINITION 4.5. A partial operator $\circ$ is defined as:

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

if $I \cap J = \emptyset$. Then we say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if for all $\tilde{s}_i \in \mathsf{dom}(\Delta_i)$ 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:

$$\begin{array}{rcl} \Delta_1 \circ \Delta_2 & = & \{\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) \mid \tilde{s} \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \\ & & \cup \Delta_1 \setminus \mathsf{dom}(\Delta_2) \cup \Delta_2 \setminus \mathsf{dom}(\Delta_1) \end{array}$$

In brief this algebra gives the disjoint union of located types which is undefined when disjointness is not satisfied.

Figure 7 Typing System for Expressions and Processes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Typing System The type assignment system for processes is given in Figure 7. We use the judgement $\Gamma \vdash P \triangleright \Delta$ which reads: "under the environment Γ, process P has typing Δ". If we set $|\tilde{s}| = 1$ and $\mathtt{n} = 2$, and delete $\mathtt{p}$ from located type, the shape of rules is essentially identical with the original binary session typing (Yoshida and Vasconcelos 2007). Below we only illustrate the rules.

[NAME],[BOOL],[OR] are the rules for the expressions and identical with (Yoshida and Vasconcelos 2007).

[MCAST] is the rule for the session request. The type for $\tilde{s}$ is the *first* projection of the declared global type for a in Γ. [MACC] is for the session accept, taking the $\mathtt{p}$-th projection. The local 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 local type. The condition $|\tilde{s}| = \max(\mathsf{sid}(G))$ ensures the number of session channels meets those in G. The typing $\tilde{s} : T@\mathtt{p}$ (which stands for $\tilde{s} : \{T@\mathtt{p}\}$) ensures 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 the number k. In both rules, "$\mathtt{p}$" in $T@\mathtt{p}$ ensures that P is (being inferred as) the behaviour for participant $\mathtt{p}$, and its domain should be

$\tilde{s}$. Then the relevant type prefixes ($k!\langle\tilde{S}\rangle$ for the output and $k?\langle\tilde{S}\rangle$ for the input) are composed in the conclusion's session environment.

[Deleg] and [SRec] are the rules for delegation of a session and its dual. Delegation of a multiparty session passes the whole 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 behaviour 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 $\tilde{t}\colon T'@\mathtt{p}'$ does not appear in P symmetrically to [SRec] which uses the channels in P.

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

[Conc] uses $\asymp$ to ensure well-formedness of the session typing, taking a the disjoint union of each local type.

[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.

An *annotated* P is the result of annotating P's bound names as e.g. $(\nu a : \langle G\rangle)P$ and $s?(x : \langle G\rangle)P$. Assuming these annotations is natural from our design framework. For typing annotated processes we assume the obvious updates for [Rcv] and [NRes] in Fig. 7.

Theorem 4.6. *Assume given an annotated program phrase P and Γ. Then it is decidable if there exists Δ such that $\Gamma \vdash P \triangleright \Delta$ or not. If such Δ exists there is an algorithm to construct one.*

Corollary 4.7 (typing for programs). *Given an annotated program P and Γ, it is decidable if $\Gamma \vdash P \triangleright \emptyset$.*

4.4 Typing Examples

Two Buyer Protocol Write Buyer1 as $\overline{a}_{[2,3]}(b_1, b_2, b_2', s).Q_1$ and Buyer2 as $a_{[2]}(b_1, b_2, b_2', s).Q_2$, both from §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 § 3.2, 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]. Note these typings are calculable by Corollary 4.7.

A Streaming Protocol We let $\Gamma = \{a : \langle G'\rangle\}$ where G' is given in the corresponding example in § 3.2. 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 which are under the initial prefix (at the shared name) of Kernel, DataProducer, Consumer and KeyProducer, respectively. Then we can type each agent 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 these types correspond to the projection of G' onto respective participants: thus Kernel, DataProducer, Consumer and KeyProducer are typable programs, 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 will type and see the relationship with global and local 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 multicasting, there are two global specifications, one for a and another for b as follows:

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

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

Let (Alice | Bob | Carol) $\to\to$ $(\nu\tilde{t}\tilde{s})(A \mid B \mid C \mid R)$ where A, B, C are the processes of Alice, Bob and Carol after initial multicasting and R are the generated queues. Let $s_1 = 1, t_1 = 1, \mathtt{A} = 1, \mathtt{B} = 2, \mathtt{C} = 3$. These processes have the following typings under Γ with $P \equiv \mathbf{0}$:

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

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

5. Safety and Progress

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

1. We extend the typing rules to include those for runtime processes which involve message queues.
2. We define reduction over session typings which eliminates a pair of minimal complementary actions from local types.
3. We then relate the reduction of processes and that of typings: showing the latter follows the former gives us *subject reduction* (Theorem 5.4), *safety* (Theorem 5.5) and *session fidelity* (Corollary 5.6), while showing the former follows the latter under a certain condition gives us *progress* (Theorem 5.12).

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

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

$$s!\langle 3\rangle; s!\langle\mathsf{true}\rangle; \mathbf{0} \mid s : \emptyset \mid s?(x); s?(x); \mathbf{0} \qquad (5)$$

We type the two processes with $s : 1!\langle\mathsf{nat}\rangle; 1!\langle\mathsf{bool}\rangle; \mathsf{end}@\mathtt{p}$ and $s : 1?\langle\mathsf{nat}\rangle; 1?\langle\mathsf{bool}\rangle; \mathsf{end}@\mathtt{q}$. After a reduction, (5) changes into:

$$s!\langle\mathsf{true}\rangle; \mathbf{0} \mid s : 3 \mid s?(x); s?(x); \mathbf{0} \qquad (6)$$

Note that (6) is identical with (5) except that an output prefix in (5) changes its place to the queue. Thus we can go back from (6) to (5) 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 a local type with a hole (a type context) for typing a queue. For example we type the queue in (6) as:

$$s : \{\ 1!\langle\mathsf{nat}\rangle; [\ ]@\mathtt{p},\ [\ ]@\mathtt{q}\ \} \qquad (7)$$

where [ ] indicates a hole. Now we cover the type $1!\langle\mathsf{bool}\rangle; \mathsf{end}$ with the type context for p given above, $1!\langle\mathsf{nat}\rangle; [\ ]$, 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 $k : l_1 l_2$ can be typed with $k\oplus l_1 : k\oplus l_2 : [\ ]$, omitting braces for a singleton selection. To do a "rollback" for selection types, we use the standard session subtyping (Gay and Hole 2005; Carbone et al. 2007), denoted $\leq_{\mathsf{sub}}$, which is the maximal fixed point of

Figure 8 Selected Typing Rules for Runtime Processes

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'} \qquad \frac{\Delta\ \mathsf{end\ only}}{\Gamma \vdash s_k : \emptyset \triangleright_{s_k} \tilde{s} : \{[\ ]@\mathrm{p}\}_{\mathrm{p}} \circ \Delta} \quad [\textsc{Subs}],[\textsc{Qnil}]$$

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

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

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

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

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

the function S that maps each binary relation $\mathcal{R}$ on local types as regular trees to $S(\mathcal{R})$ given as:

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

For example we have $k \oplus \{\mathsf{ok} : T_1\} \leq_{\mathsf{sub}} k \oplus \{\mathsf{ok} : T_1, \mathsf{quit} : T_2\}$.

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

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

The *isomorphism* $\approx$ *on type contexts* is generated from permutations given in §4.2 (3, 4). Each assignment in Δ may contain both local types and type contexts. We extend $\circ$ in Definition 4.5 as follows ($\mathsf{sid}(\mathcal{T})$ denotes 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 iff $\mathcal{T}' \circ \mathcal{T}$ is defined and in which case we have $\mathcal{T}[\mathcal{T}'] \approx \mathcal{T}'[\mathcal{T}]$. Note also $T \circ T'$ is never defined. Composition of $\{H_{\mathrm{p}}@\mathrm{p}\}_{\mathrm{p} \in I}$ by $\circ$ is given point-wise. The definition of $\Delta_1 \circ \Delta_2$ and $\Delta_1 \asymp \Delta_2$ stay the same as Definition 4.5.

Typing Rules for Runtime To guarantee that there is at most one queue for each channel, we use the typing judgement 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 judgement. [Qval] says 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. [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 all message queues are composed and the given configuration at $\tilde{s}$ is coherent.

The original typing rules in Figure 7 not appearing in Figure 8 are refined as follows: [Mcast], [Macc], [Rcv], [SRec], [Branch], [Def] replace $\Gamma \vdash P \triangleright \Delta$ with $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ and both [Def] and [NRes] 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, (5) and (6), are given an identical typing by "rolling back" the type of the message in the queues.

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

Proposition 5.1. *Let P be a program phrase and Δ be without a type context. Then $\Gamma \vdash P \triangleright \Delta$ iff $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ without using [*Subs*].*

Proposition 5.2. *If $\Gamma \vdash P \triangleright_{s_1..s_m} \Delta$ 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.*

Type Reduction Next we introduce reduction over session typings and global types, which abstractly represents interaction at session channels. Below we assume well-formedness of typing.

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

$$k \oplus \{l : H, ...\}@\mathrm{p}, k\&\{l : T, ...\}@\mathrm{q} \xrightarrow{k} H@\mathrm{p}, T@\mathrm{q} \qquad [\textsc{TR-Bra}]$$

$$\frac{H_1@\mathrm{p}_1, H_2@\mathrm{p}_2 \xrightarrow{k} H'_1@\mathrm{p}_1, H'_2@\mathrm{p}_2 \quad \mathrm{p}_1, \mathrm{p}_2 \in I}{\tilde{s} : \{H_1@\mathrm{p}_1, H_2@\mathrm{p}_2, ...\}_{i \in I}, \Delta \xrightarrow{s_k} \tilde{s} : \{H'_1@\mathrm{p}_1, H'_2@\mathrm{p}_2, ...\}_{i \in I}, \Delta} \qquad [\textsc{TR-Context}]$$

Then we write $G \xrightarrow{k} G'$ if $[[G]] \xrightarrow{k} [[G']]$ where we set $[[G]]$ to be the family $\{(G \upharpoonright \mathrm{p})@\mathrm{p} \mid \mathrm{p} \in \mathsf{pid}(G)\}$. In $G \xrightarrow{k} G'$, we take off a prefix at k in G not suppressed by $\prec_{\mathsf{II}}$, $\prec_{\mathsf{IO}}$ or $\prec_{\mathsf{OO}}$, to obtain G'.

Proposition 5.3. *Below* Δ is coherent *if $\Delta(\tilde{s})$ is coherent for each $\tilde{s} \in \mathsf{dom}(\Delta)$.*

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

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

Theorem 5.4 (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 $P \rightarrow P'$ imply $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ where $\Delta = \Delta'$ or $\Delta \xrightarrow{s'} \Delta'$ for some s'.*
3. *$\Gamma \vdash P \triangleright_{\emptyset} \emptyset$ and $P \rightarrow P'$ imply $\Gamma \vdash P' \triangleright_{\emptyset} \emptyset$.*

Theorem 5.4 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:

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

We also say and write:

- A prefix is *at s* (resp. *at a*) if its subject (i.e. its initial channel) is s (resp. a). Further 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 $P\langle\!\langle s!\rangle\!\rangle$ (resp. $P\langle\!\langle s?\rangle\!\rangle$) if P contains an emitting (resp. receiving) active prefix at s.
- P has a *redex at s* if it has an active prefix at s among its redexes.

The following result decomposes the standard property for synchronous session types (Takeuchi et al. 1994; Honda et al. 1998; Yoshida and Vasconcelos 2007) into the sending side and the receiving side, due to the existence of queues. We assume the standard bound name convention.

Theorem 5.5 (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* $R \equiv s \rhd \{l_i\colon Q_i\}_{i\in I}$ *then* $P \equiv \mathcal{E}'[s:l_j\cdot\tilde{h}]$ *for some* $\mathcal{E}'$ *and* $j \in I$.

By Theorems 5.4 and 5.5, 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. As the corollary of Theorem 5.4(2) and Proposition 5.3(3), we obtain *session fidelity*: the interactions of a typable process exactly follow the specification described by its global type.

Corollary 5.6 (session fidelity). *Assume* $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ *such that* Δ *is coherent and* $\Delta(\tilde{s}) = [\![G]\!]$. *If* $P \to P'$ *at the redex of* s_k, *then* $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ *with* $G \xrightarrow{k} G'$ *and* $[\![G']\!] = \Delta'(\tilde{s})$.

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 initialisation and communication of *different* sessions, then the converse holds: the reduction predicted by the typing surely takes place, the standard progress property in binary session types (Dezani-Ciancaglini et al. 2006; Honda et al. 1998). First we define:

Definition 5.7. 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 initialisations and communications) which can introduce deadlock. For example, two session initialisations $a[2](s).b[2](t).s?;t!$ and $\overline{a}[2](s).\overline{b}[2](t).t?;s!$ cause deadlock. Observe, because we have multiparty sessions, there is less need to use interleaved sessions.

Definition 5.8 (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.

Thus each prefixed subterm in a simple process has only a unique session. Another element which can hinder progress is when interactions at shared names cannot proceed.

Definition 5.9 (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. Below we safely confuse a channel in a typing and the corresponding free session channel of a process.

Lemma 5.10. *Let* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *and P is simple. If there is an active receiving (resp. active emitting) 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* $P\langle\!\langle s?\rangle\!\rangle$ *(resp. either* $P\langle\!\langle s!\rangle\!\rangle$ *or the queue at s is not empty).*

Proposition 5.11. *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* $G \xrightarrow{k} G'$, *then* $P \to^+ P'$ *at the redex at* t_k *s. t.* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ *with* $\Delta'(\tilde{t}) = [\![G']\!]$.

(2) above gives the converse of Corollary 5.6: if the global type has a reduction, then the process can always realise it.

Theorem 5.12 (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''.

A simple application of Theorems 5.4 (3), 5.5 and 5.12 for processes from §2.3 follow. Below *communication mismatch* stands for the violation of the conditions given in Theorem 5.5 (2).

Proposition 5.13 (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.*

6. Extensions and Related Work

6.1 Extensions

Graph-Based Global Types The syntax of global types uses the standard abstract syntax tree. We can further generalise this tree-based syntax to graph structures to obtain a strictly more expressive type language, enlarging typability. Consider the two end-point 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 § 3.3, based on input/output dependencies, can directly capture the safety of this configuration. All we need to do is to take the graphs of prefixes and II, IO and OO-edges (cf. Figure 5) under the linearity condition (precisely following §3.3) as global types, augmented with an acyclicity condition on chains of these causal edges. All other definitions and results stay the same.

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 § 3.2.

A different direction is to consider asynchronous message passing without order-preservation (Honda and Tokoro 1991) which are also used in some computing environments (though in many environments we have efficient order-preserving transport 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 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 $\mathtt{p} \to \mathtt{p}_1,..,\mathtt{p}_n : \{k_1,...,k_n\}$ (with a practical adaptation such as group addressing), representing the multicast of a message to $\mathtt{p}_1,..,\mathtt{p}_n$ via channels $k_1,...,k_n$ by participant $\mathtt{p}$, similarly we extend local 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.

6.2 Related Work

Asynchronous Session Types Our 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 (2004a). Recently Gay and Vasconcelos (2007) study the asynchronous version of binary sessions for an ML-like language based on (Vasconcelos et al. 2006). In (Gay and Vasconcelos 2007), a message queue is given two endpoint channels and a direction.

Coppo, Dezani-Ciancaglini, and Yoshida (2007) study the asynchronous binary session types for Java, extending the previous work in (Dezani-Ciancaglini et al. 2006), and prove the progress by introducing an effect system. The resulting system does not allow interleaving sessions so that interactions involving more than two parties such as examples in § 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, de' Liguoro, and Yoshida (2008) propose a typing system for progress in binary synchronous interleaving sessions. Typable processes there obey the partial orders of shared and session channels inferred during type-checking. Because of a use of the global types, processes typed by our multiparty session typing do not have to follow such ordering. The system in (Dezani-Ciancaglini et al. 2008) does not include recursive agents or types, but does not require the simpleness condition (Definition 5.8). A combination of these two typing systems will enlarge typability, guaranteeing the progress in many situations.

The concurrent work done by Bonelli and Compagnoni (2008), which is independently conceived and developed, studies a multiparty session typing for asynchronous communication. While 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) local types. In type checking, local 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 the guarantee of stronger behavioural properties such as progress, and (arguably) more intelligible description of multiparty interaction structures. Given the complementary nature of these two works, a further study of their relationship is an interesting topic of further study.

Global Description of Session Types There are two recent works which studied global descriptions of sessions in the context of web services and business protocols, by the present authors (Carbone et al. 2007) and by Corin et al. (2007). Carbone, Honda, and Yoshida (2007) 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 local 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 local) calculi studied in (Carbone et al. 2007) are based on binary synchronous session types, hence safety and progress for multiparty interactions are not considered.

Corin et al. (2007) 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 (Corin et al. 2007) 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. Secondly 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 towards type abstractions: for example their choice structures are based on transitions of flow graphs than additive structures realisable by branching and selection.

Semantics of Delegation The present work uses, for a simpler presentation, 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 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; 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. With this change the whole theories remain intact with exactly the same operational semantics and typing for programs.

Linear and Behavioural Types for Mobile Processes The session type disciplines are related with linear/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) where ordering among multiple linear names (which correspond to session channels) guarantees deadlock-freedom of typed processes. Kobayashi and his colleagues (Kobayashi 2006; Igarashi and Kobayashi 2004; Kobayashi et al. 2000) study generalised forms of linear typing for guaranteeing different kinds of deadlock-freedom, incorporating synchronisations and locking, with a detailed type inference system.

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. 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.

One of the novelties of the present work is the introduction of global description 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 while 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.

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 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, the static participants are enough even for delegations. It is a valuable further study to investigate a dynamic change of participant numbers when session initialisation (without explicitly declaring p in the syntax) by using channel dependent types (Mostrous and Yoshida 2007) or polymorphism.

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, what is their additional expressiveness and benefits. This paper 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. The central technical underpinning of the present work is the introduction of global types, which offer an intuitive syntax for describing multiparty conversation structures from a global viewpoint; and the use of their projection for efficient type-checking, proposing a new effective methodology for programming multiparty interactions in distributed environments. Global types also offer a basis of a clean modular causal analysis systematically applicable to both synchronous and asynchronous communications, ensuring the 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 generalised session type structure as one of the formal foundations of the next version of a web service description language, WS-CDL from W3C (WS-CDL) and a message scheme for financial protocols, UNIFI from ISO (UNIFI). Another topic is the use of this theory as a basis of communication-centred extensions of general purpose programming languages (Hu et al. 2007). Others include tools assistance for the design and elaboration of global types; incorporation of typed exceptions to sessions; and integration of the type discipline with diverse specification concerns including security and assertional methods.

Acknowledgements. We thank the reviewers for their useful comments and suggestions and our academic and industry colleagues for their stimulating conversations. The work is partially supported by EPSRC GR/T04236, GR/T04724, GR/T03208, GR/T03215, EP/F002114, EP/F003757 and IST2005-015905 MOBIUS.

References

Eduardo Bonelli and Adriana Compagnoni. Multipoint session types for a distributed calculus. In *TGC07*, LNCS. Springer, 2008. To appear.

Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. To be published by W3C. Available at www.dcs.qmul.ac.uk/~carbonem/cdlpaper, 2006.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.

Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, Karthikeyan Bhargavan, and James Leifer. Secure Implementations for Typed Session Abstractions. In *CFS'07*. IEEE-CS Press, 2007.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In *TGC07*, LNCS. Springer, 2008. To appear.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In *PPDP'06*, pages 61–72. ACM Press, 2006.

Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

Simon Gay and Vasco T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, may 2007.

Jean-Yves Girard. Linear logic. *TCS*, 50:1–102, 1987.

Matthew Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.

Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, 1991.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007a.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Full version of this paper. Technical Report 5, Imperial College London, 2007b.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. Type-safe Communication in Java with Session Types. http://www.doc.ic.ac.uk/~rh105/sessiondj.html, March 2007.

Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.

Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR'00*, volume 1877 of *LNCS*, pages 489–503, 2000.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

Matthias Neubauer and Peter Thiemann. Session Types for Asynchronous Communication. Universität Freiburg, 2004a.

Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004b.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.

Stephen Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), March 2006.

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. http://www.iso20022.org.

Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2): 64–87, 2006.

WS-CDL. Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/.

Nobuko Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCS*, pages 371–386. Springer, 1996.

Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisit. In *SecRet'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.