Multiparty Asynchronous Session Types

Kohei Honda[1], Nobuko Yoshida[2], Andi Bejleri[2], and Marco Carbone[1]

[1] Department of Computer Science, Queen Mary, University of London
[2] Department of Computing, Imperial College London

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 specifying dependencies and capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual peers. The fundamental properties of the session type discipline such as communication safety, progress and session fidelity are established for general n-party asynchronous interactions.

Contents

1 Introduction . . . 1
2 Multiparty Asynchronous Sessions . . . 4
  2.1 Syntax for Multiparty Sessions . . . 4
  2.2 Operational Semantics . . . 5
  2.3 Examples . . . 7
3 Global Types and Causal Analysis . . . 10
  3.1 Session Types from a Global Viewpoint . . . 11
  3.2 Examples of Global Types . . . 12
  3.3 Safety Principle for Global Types: Linearity of Channels . . . 13
4 Type Discipline for Multiparty Sessions . . . 15
  4.1 Programming Methodology for Multiparty Interactions . . . 15
  4.2 End-point Types . . . 16
  4.3 Typing System . . . 18
  4.4 Typing Examples . . . 20
5 Safety and Progress . . . 21
  5.1 Typing Runtime . . . 22
  5.2 Typing Rules for Runtime . . . 24
  5.3 Type Reduction . . . 25

5.4 Subject Reduction and Communication Safety 27
5.5 Progress 31
6 Extensions and Related Work 33
6.1 Extensions 33
6.2 Related Work 34
7 Conclusion 37
A Proof of Proposition 3.6 40
B Full Typing Rules for Runtime Processes 42
B.1 Proof of Proposition 5.3 43
B.2 Proof of Proposition 5.4 43
B.3 Proof of Proposition 5.13 44
B.4 Proof of Lemma 5.16 44
B.5 Proof of Subject Reduction Theorem (Theorem 5.22) 46
B.6 Remaining Cases of Theorem 5.22 51
B.7 Proof of Lemma 5.23 53
B.8 Proof of Proposition 5.28 54
B.9 Proof of Lemma 5.30 56

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 [5, 9, 14, 18, 19, 24, 33, 49, 56], higher-order processes [34], Ambients [17], multi-threaded ML [20, 51], Haskell [38, 44, 45], F# [13], operating systems [16], Java [12, 15, 27] and Web Services [8, 10, 25, 48, 53]. 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 protocols or 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.

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

Above $!\mathsf{t}$ denotes an output of a value of type t, $?\mathsf{t}$ denotes an input of a value of type 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 bug. 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 [27, 50, 53]. In addition, a clean separation between abstraction and implementation given by type-based abstraction and associated primitives leads to intelligible programs and flexible implementations [27]. 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).

Thus at each step in a session, a single input and a single output or a single selection and a single branching take place via a session channel, moving to the next step.

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 much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller. It is extremely awkward (if logically possible) to decompose this scenario into three binary sessions, between Buyer1 and Seller, between Buyer2 and Seller, and between Buyer1 and Buyer2. Abstracting this protocol as three separate session types also means that our type abstraction loses essential sequencing information in this interaction scenario. For validating this conversation scenario as a whole, therefore, the conversation structure should be represented as a *single session*.

Many existing business protocols including financial protocols are written as a collaboration of several peers. Typical message-passing parallel algorithms also frequently demand distribution of a request to, and collection of the results from, many peers. All these 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 [53] through our collaboration as invited experts [8, 10, 25], 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 [21]. 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. A linearity property holds if a communication

via the same channel of a global type does not break the order of messages as it is specified in the global description. This demands a precise causal analysis for correct sequencing of interactions distributed among 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 with respect to the protocol specification given as a causality analysis of actions in global types, modularly articulating different kinds of dependency.
3. A type discipline for individual processes (programs) which uses a global type through its *projection* onto individual end-point participants: the resulting end-point 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 [10, 53]. 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. The projection can also be used as a hint for modelling and designing local behaviours of participants. After the development, a global type will serve as a basis of documentation for maintenance and upgrade. 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 end-point realiser). The resulting type discipline guarantees all the original key properties, such as communication error freedom, progress and fidelity in a session among multiparty.

This paper is a full version of [26], with detailed definitions and full proofs. It is also expanded with more examples and comparisons with recent related work. 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. Appendix gives the proofs of the propositions, lemmas and theorems stated in the main sections.

2 Multiparty Asynchronous Sessions

2.1 Syntax for Multiparty Sessions

Several versions of the π-calculi with session types are proposed in the literature; the paper [56] offers detailed discussions and analysis of their typing systems. We use a simple extension of the original language in [24, 49] 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 $s, t, ...$; *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.

Fig. 1 Syntax

P	$::= \overline{a}_{[2..\mathrm{n}]}\,(\tilde{s}).P$	multicast session request
	$\mid a_{[\mathrm{p}]}\,(\tilde{s}).P$	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
	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional branch
	$\mid P \mid Q$	parallel composition
	$\mid \mathbf{0}$	inaction
	$\mid (\nu n)P$	hiding
	$\mid \mathsf{def}\ D\ \mathsf{in}\ P$	recursion
	$\mid X\langle\tilde{e}\tilde{s}\rangle$	process call
	$\mid s : \tilde{h}$	message queue
e	$::= v \mid e\ \mathsf{and}\ e' \mid \mathsf{not}\ e \quad \ldots$	expressions
v	$::= a \mid \mathsf{true} \mid \mathsf{false}$	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

Except for the first two primitives for session initiation and the final message queue, all constructs are from the binary session calculi [24]. For the primitives for session initiation, the prefix process $\overline{a}_{[2..n]}(\tilde{s}).P$ 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 $2 \leq \mathtt{p} \leq \mathtt{n}$. All receive $\tilde{s}$, over which the actual session communications can now take place among the $\mathtt{n}$ parties. $\mathtt{p}$, $\mathtt{q}$,... 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 $s:\tilde{h}$ 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 $\overline{a}_{[2..n]}(\tilde{s}).P$, $a_{[\mathtt{p}]}(\tilde{s}).P$ and $s?(\tilde{s});P$, $\tilde{x}$ in $s?(\tilde{x});P$, $\tilde{x}\tilde{s}$ in $X(\tilde{x}\tilde{s}) = P$, n in $(\nu n)P$ and process variables in $\mathsf{def}\ D\ \mathsf{in}\ P$. The notions of bound and free identifiers, channels, alpha equivalence $\equiv_\alpha$ and substitution are standard. $\mathsf{fpv}(P)$ and $\mathsf{fn}(P)$, respectively denote the sets of *free process variables* and *free identifiers* in P. $\mathsf{dpv}(\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I})$ denotes the set of *process variables* $\{X_i\}_{i\in I}$ introduced in $\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$. A sequence of parallel composition is written $\Pi_i P_i$.

2.2 Operational Semantics

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

Fig. 2 Structural congruence.

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

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

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

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

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

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

Using $\equiv$, 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. We illustrate each rule one by one.

Rule [L] describes a session initiation among $\mathtt{n}$-parties through n-party synchronisation, generating m fresh session channels and the associated m empty queues ($\emptyset$ denotes the empty string). Each fresh channel is given a new empty queue. As a result $\mathtt{n}$ participants now share the newly generated m channels, hence their queues. Note the number of participants ($\mathtt{n}$) can be different from that of session channels (m), giving flexibility in channel usage. The use of the n-party synchronisation in this rule captures, albeit abstractly, an n-party handshake which would be necessary for establishing an n-party link in real-world protocols.

Fig. 3 Reduction

Rule	Label
$\overline{a}[2..\mathtt{n}]\,(\tilde{s}).P_1 \mid a[2]\,(\tilde{s}).P_2 \mid \cdots \mid a[\mathtt{n}]\,(\tilde{s}).P_n \;\to\; (\nu\,\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_{\mathtt{n}} \mid s_1 : \emptyset \mid \cdots \mid s_m : \emptyset)$	[L]
$s!\langle\tilde{e}\rangle; P \mid s : \tilde{h} \;\to\; P \mid s : \tilde{h} \cdot \tilde{v} \quad (\tilde{e} \downarrow \tilde{v})$	[S]
$s!\langle\tilde{t}\rangle; P \mid s : \tilde{h} \;\to\; P \mid s : \tilde{h} \cdot \tilde{t}$	[D]
$s \lhd l; P \mid s : \tilde{h} \;\to\; P \mid s : \tilde{h} \cdot l$	[L]
$s?(\tilde{x}); P \mid s : \tilde{v} \cdot \tilde{h} \;\to\; P[\tilde{v}/\tilde{x}] \mid s : \tilde{h}$	[R]
$s?((\tilde{t})); P \mid s : \tilde{t} \cdot \tilde{h} \;\to\; P \mid s : \tilde{h}$	[SR]
$s \rhd \{l_i : P_i\}_{i \in I} \mid s : l_j \cdot \tilde{h} \;\to\; P_j \mid s : \tilde{h} \quad (j \in I)$	[B]
$\text{if } e \text{ then } P \text{ else } Q \;\to\; P \quad (e \downarrow \text{true})$	[I T]
$\text{if } e \text{ then } P \text{ else } Q \;\to\; Q \quad (e \downarrow \text{false})$	[I F]
$\text{def } D \text{ in } (X\langle\tilde{e}\tilde{s}\rangle \mid Q) \;\to\; \text{def } D \text{ in } (P[\tilde{v}/\tilde{x}] \mid Q) \quad (\tilde{e} \downarrow \tilde{v}, X(\tilde{x}\tilde{s}) = P \in D)$	[D]
$P \to P' \;\Rightarrow\; (\nu n)P \to (\nu n)P'$	[S]
$P \to P' \;\Rightarrow\; P \mid Q \to P' \mid Q$	[P]
$P \to P' \;\Rightarrow\; \text{def } D \text{ in } P \to \text{def } D \text{ in } P'$	[D]
$P \equiv P' \text{ and } P' \to Q' \text{ and } Q' \equiv Q \;\Rightarrow\; P \to Q$	[S]

Rules [S], [D] and [L] respectively enqueue values, channels and a label at the tail of the queue for s. Symmetrically rules [R], [SR] and [B] dequeue, at the head of the queue, values, channels and a label. Rules [R] and [B] respectively further instantiates the value in the body and selects the corresponding branch.

In these communication rules, sending and receiving are mediated by a queue: only when a message sent by (say) Alice is received by (say) Bob by going through a queue,

we can say that an interaction between Alice and Bob has taken place. Since [L] generates a queue for each channel, these rules entail that:

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

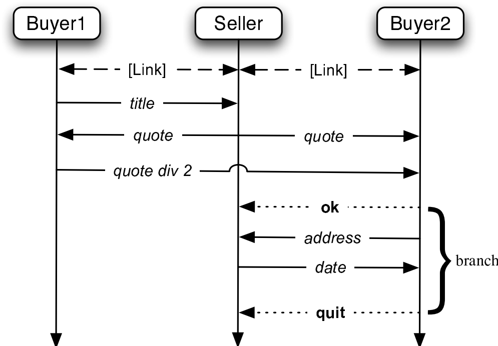
As we discussed in Introduction, these are among the characteristics of well-known transport mechanisms such as TCP.

All other rules are standard: for reference we briefly illustrate them. The two rules for conditional, [I T] and [I F], reduce to one of the branches depending on the result of evaluating the conditional guard. Rule [D] performs unfolding of recursion. Rules [S], [P] and [D] close the reduction under hiding, parallel composition and definition. Finally Rule [S] says that the reduction is defined over processes up to ≡.

Remark 2.1 This delegation rule (from [24]) is chosen over the more liberal one in [19, 20, 56] (which uses substitution as in [R]) for simpler presentation. The technical development does not depend on this choice, see §6.2.

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:

$$\text{Buyer1} \stackrel{\text{def}}{=} \overline{a}_{[2,3]}\,(b_1, b_2, b'_2, s).\ \ s!\langle\text{“War and Peace”}\rangle;\\ b_1?(\mathit{quote});\ \ b'_2!\langle \mathit{quote}\ \mathsf{div}\ 2\rangle; P_1$$

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.

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

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.[3] 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 [30]). Thus if we were to use b_2 for both actions, the two messages can be confused, losing linear usage of a channel. The problem becomes visible after the fifth step of the following reduction. If b_2 and b_2' were the same then the contribution of the Buyer1 could be queued before the price of the book and therefore received before at Buyer2. Later we shall show our type discipline can detect such an error.

We show an example of the reduction. Let us define:

$$
\begin{aligned}
P &\triangleq \text{if } (\mathit{quote} - \mathit{contrib} \;\leq\; 99)\\
&\qquad \text{then} \quad s \lhd \mathsf{ok}; s!\,\langle \mathit{address}\rangle; b_2?(x); P_2\\
&\qquad \text{else} \quad s \lhd \mathsf{quit}; \mathbf{0}\\[1ex]
S &\triangleq s \rhd \{\mathsf{ok}\colon\; s?(x); b_2!\,\langle \mathit{date}\rangle; Q\,,\quad \mathsf{quit}\colon\; \mathbf{0}\}.
\end{aligned}
$$

Below a tag denotes a name of the applied rule defined in Figure 3. After the second reduction, we omit [P] and [S] for simplicity.

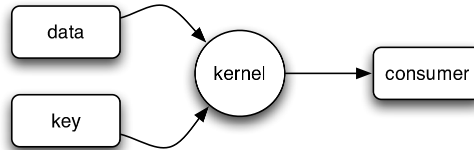
[3] Due to asynchrony there is in effect no order among the sending actions at $s_1..s_m$.

Buyer1 | Buyer2 | Seller

$$
\begin{array}{lll}
\to^{[\mathrm{L}\quad]} & (\nu\, b_1, b_2, b_2', s)(& s!\langle\text{“War and Peace”}\rangle; b_1?(quote); b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1 \\
& & \mid b_2?(quote); b_2'?(contrib); P \\
& & \mid s?(title); b_1, b_2!\langle quote\rangle; S \\
& & \mid b_1 : \emptyset \mid b_2 : \emptyset \mid b_2' : \emptyset \mid s : \emptyset) \\
\to^{[\mathrm{S}\quad],[\mathrm{P}\quad],[\mathrm{S}\quad]} & (\nu\, b_1, b_2, b_2', s)(& b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1 \\
& & \mid b_2?(quote); b_2'?(contrib); P \\
& & \mid s?(title); b_1, b_2!\langle quote\rangle; S \\
& & \mid b_1 : \emptyset \mid b_2 : \emptyset \mid b_2' : \emptyset \mid s : \text{“War and Peace”}) \\
\to^{[\mathrm{R}\quad]} & (\nu\, b_1, b_2, b_2', s)(& b_1?(quote); b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1 \\
& & \mid b_2?(quote); b_2'?(contrib); P \\
& & \mid b_1, b_2!\langle quote\rangle; S \\
& & \mid b_1 : \emptyset \mid b_2 : \emptyset \mid b_2' : \emptyset \mid s : \emptyset) \\
\to^{[\mathrm{S}\quad]} & (\nu\, b_1, b_2, b_2', s)(& b_1?(quote); b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1 \\
& & \mid b_2?(quote); b_2'?(contrib); P \\
& & \mid b_2!\langle quote\rangle; S \\
& & \mid b_1 : quote \mid b_2 : \emptyset \mid b_2' : \emptyset \mid s : \emptyset) \\
\to^{[\mathrm{R}\quad]} & (\nu\, b_1, b_2, b_2', s)(& b_2'!\langle quote\ \mathsf{div}\ 2\rangle; P_1 \\
& & \mid b_2?(quote); b_2'?(contrib); P \\
& & \mid b_2!\langle quote\rangle; S \\
& & \mid b_1 : \emptyset \mid b_2 : \emptyset \mid b_2' : \emptyset \mid s : \emptyset)
\end{array}
$$

...

A Streaming Protocol We next consider a simple protocol for the standard stream cipher [46].



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:

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

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:[4]

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

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

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.

3 Global Types and Causal Analysis

Fig. 4 Syntax of Global Types

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

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: global types introduced this section extend binary session types in the sense that they not only can ensure communication-error freedom but also they express dependencies between communications between multiple peers.

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

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 through channel k (represented as a finite natural number) to participant $\mathtt{p}'$: and interactions described in G' takes place. $U, \ldots$ 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. In the value types, $T@\mathtt{p}$ means communication delegating a behavior typed by T of participant $\mathtt{p}$. In the sorts, $\langle G \rangle$ means a delivery of a shared name typed by $\langle G \rangle$. We often write $\mathtt{p} \to \mathtt{p}' : k.G'$ for $\mathtt{p} \to \mathtt{p}' : k \langle \rangle.G'$ (i.e. U is empty).

Type $\mathtt{p} \to \mathtt{p}' : k \{l_j : G_j\}_{j \in J}$ says participant $\mathtt{p}$ sends one of the labels on channel k to participant $\mathtt{p}'$. If l_j is sent from $\mathtt{p}$, interactions described in G_j take place at $\mathtt{p}'$.

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 $G[\mu\mathbf{t}.G/\mathbf{t}]$ [41]. We assume that $\langle G \rangle$ in the grammar of sorts is closed, i.e. without type variables.[5] Type end represents the termination of the session and is often omitted. 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 [41]. A basic ordering on its nodes is induced by prefixes.

Definition 3.3 (prefix ordering) Write $\mathsf{n}, \mathsf{n}', ..$ for prefixes occurring in a global type, say G (but not in its carried types), seen as nodes of G as a graph. We write $\mathsf{n} \in G$ when n occurs in G. Then we write $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ when n_1 directly or indirectly prefixes n_2 in G. Formally $\prec$ is the least partial order 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.

[5] In the presence of the standard recursive sorts [24], which we omit for simpler presentation, we allow sort variables to occur in $\langle G \rangle$.

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:

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

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 [30]. 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 participants 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 $s = 4$.

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

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 have a conflict at* s (i.e. loose the ordering). Note that 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 $\mathtt{KP} = 4$.

1	$\mu\mathbf{t}.\ \mathtt{DP} \to \mathtt{K}\colon d\,\langle\mathsf{bool}\rangle.$	4	$\mathtt{DP} \to \mathtt{K}\colon d\,\langle\mathsf{bool}\rangle.$
2	$\mathtt{KP} \to \mathtt{K}\colon k\,\langle\mathsf{bool}\rangle.$	5	$\mathtt{KP} \to \mathtt{K}\colon k\,\langle\mathsf{bool}\rangle.$
3	$\mathtt{K} \to \mathtt{C}\colon c\,\langle\mathsf{bool}\rangle.$	6	$\mathtt{K} \to \mathtt{C}\colon c\,\langle\mathsf{bool}\rangle.\mathbf{t}$

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: Linearity of Channels

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 an ordering of interactions, and can be used as a basis of guaranteeing process behaviours through type checking.

Fig. 5 Causality Analysis

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

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 $\phi_i \in \{\mathsf{IO}\}$ 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 and, both input and output dependencies exist from n_i to n_j ($i \neq j$, $i, j \in \{1, 2\}$). If G carries other global types, we inductively demand the same.

We illustrate the condition on branching by an example:

$$
\begin{array}{lll}
1.\ \mathtt{A} \to \mathtt{B} : t\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end} & & \mathtt{A} \to \mathtt{B} : t.(\mathtt{C} \to \mathtt{D} : s.\mathsf{end}, \\
2.\ \qquad\qquad\quad \mathsf{quit} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end}\ \} & & \qquad\quad \mathtt{C} \to \mathtt{D} : s.\mathsf{end})
\end{array}
$$

(a) branching (b) parallel

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.

Note that we do not require the ordering between $\mathsf{n}_i \in |G_k|$ and $\mathsf{n}_j \in |G_h|$ in $\mathtt{p} \to \mathtt{p}' : k\, \{l_j : G_j\}_{j \in J}$ such that $h, k \in J$, $h \neq k$ since only one of branches is performed.

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

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

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:

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

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

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

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

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

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

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

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

But in fact unfolding once turns out to be enough. Taking G as a syntax, let us call the *one-time unfolding of* G the result of unfolding once for each recursion in G (but never in carried types), and replacing the remaining variable with end.

Proposition 3.6 *1. The one-time unfolding of a global type is linear if and only if its n-th unfolding is linear.*
2. The linearity of a global type is decidable.

Proof. For (1), the if-direction is obvious. The only if-direction is proved by induction on n. See Appendix A for the full proofs. (2) is an immediate corollary of (1).[6]

4 Type Discipline for Multiparty Sessions

4.1 Programming Methodology for Multiparty Interactions

Once given global types as a description of global interactions, 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.

Fig. 6 Syntax of End-point Session Types

Value	U	$::=$	$\tilde{S} \mid T@\mathtt{p}$	
Sort	S	$::=$	$\mathsf{bool} \mid \ldots \mid \langle G \rangle$	
End-point	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
		$\mid$	$\mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end}$	

[6] We also examine the compuational complexity in [36].

4.2 End-point Types

Syntax *End-point session types* or *end-point types*, ranged over by $T, T', ..$, are types for end-point behaviour of processes, acting as a link between global types and processes in Section 3, which give intended conversation structures of multiparty sessions, and processes in Section 2. 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 [24] 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 used for type-checking each participant, we use a notation $T@\mathtt{p}$ (called *located type*) representing an end-point type T assigned to participant $\mathtt{p}$. A located type is also used for delegation.

The rest stays identical with the original session types. 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 $k\&\{l_i\colon T_i\}_{i\in I}$ describes a branching: it waits with n options at k, and behave as type T_i if i-th label is selected; type $k\oplus\{l_i\colon T_i\}_{i\in I}$ represents the behaviour which selects one of the labels say l_i at k then behaves as T_i. These four are *action prefixes* in end-point types. 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 that end-point type T does not contain parallel composition as the original session types, retaining simplicity.

In addition to the folding/unfolding of recursive types, end-point types are considered up to the following isomorphism (closed under all type constructors).

$$k!\langle U\rangle;k'!\langle U'\rangle;T \;\approx\; k'!\langle U'\rangle;k!\langle U\rangle;T \qquad (k\neq k') \tag{3}$$

$$k\oplus\{l_i : k'\oplus\{l'_j : T_{ij}\}_{j\in J}\}_{i\in I} \;\approx\; k'\oplus\{l'_j : k\oplus\{l_i : T_{ij}\}_{i\in I}\}_{j\in J} \qquad (k\neq k') \tag{4}$$

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 end-point types at each participant.

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

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

– $(\mathtt{p}_1 \to \mathtt{p}_2 : k\,\{l_j : G_j\}_{j\in J}) \upharpoonright \mathtt{p} =$

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

– $(G_1, G_2) \upharpoonright \mathtt{p} =$

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

– $(\mu\mathbf{t}.G) \upharpoonright \mathtt{p} = \mu\mathbf{t}.(G \upharpoonright \mathtt{p})$, $\mathbf{t} \upharpoonright \mathtt{p} = \mathbf{t}$, and $\mathsf{end} \upharpoonright \mathtt{p} = \mathsf{end}$.

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 of participants that behavior does not depend on the branching should generate an identical end-point type (otherwise undefined); and in parallel composition, $\mathtt{p}$ 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) $T_{\mathtt{p}}@\mathtt{p}$ appeared at the beginning of §4.2.

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

Theorem 4.3 *Coherence of G is decidable.*

Proof. By Proposition 3.6 (2) (note the projection is only applied to a given global type without unholding).[7] □

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 $\mathtt{C}$ or $\mathtt{D}$, regardless of the choice made by $\mathtt{A}$, they should behave in the same way: participants $\mathtt{C}$ and $\mathtt{D}$ 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:

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

[7] [36] gives a complexity analysis.

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

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

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

4.3 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:

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

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to sorts 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.

Notations 4.5
- $\mathsf{sid}(G)$ stands for the set of session channel numbers in G.
- We write Δ, Δ' to denote a typing made from the disjoint union of Δ and Δ' always assuming their domains contain disjoint sets of session channels.
- We write $\tilde{s} : T@\mathtt{p}$ for a singleton typing $\tilde{s} : \{T@\mathtt{p}\}$.

Typing System The type assignment system for processes is given in Figure 7. We use the judgement for the process and expression as:

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

which read: "under the environment Γ, process P has typing Δ" and "under the environment Γ, expression e has type S". If we set $|\tilde{s}| = 1$ and $\mathtt{n} = 2$, and delete $\mathtt{p}$ from located type, the shape of rules is essentially identical with the original binary session typing [56]. Below we only illustrate the rules.

[N],[B],[O] are the rules for the expressions and identical with [56].

Fig. 7 Typing System for Expressions and Processes

$$\Gamma, a : S \vdash a : S \qquad \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{bool} \qquad \frac{\Gamma \vdash e_i \triangleright \mathsf{bool}}{\Gamma \vdash e_1 \mathsf{or}\ e_2 : \mathsf{bool}}$$ [N], [B], [O]

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

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

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

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

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

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

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

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

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

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

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

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

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

[M] 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. [M] is for the session accept, taking the $\mathtt{p}$-th projection. The end-point 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 end-point type. The condition $|\tilde{s}| = |\mathsf{sid}(G)|$ ensures the number of session channels meets those in G. The typing $\tilde{s} : T@\mathtt{p}$ (stands for $\tilde{s} : \{T@\mathtt{p}\}$) means that each prefix does not contain parallel threads which share $\tilde{s}$.

[S] and [R] 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, "p" in T@p ensures that P is (being inferred as) the behaviour for participant 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.

[D] and [SR] are the rules for delegation of a session and its dual. Delegation of a multiparty session passes the whole remaining capability to participate in a multiparty session: thus operationally we send the whole vector of session channels. The carried type T' is located, making sure that the behaviour by the receiver at the passed channels takes the role of a specific participant (here p′) in the delegated multiparty session. The rest follows the standard delegation rule [56], observing [D] says that $\tilde{t}: T'$@p′ does not appear in P symmetrically to [SR] which uses the channels in P.

[S] and [B] are the rules for selection and branching, and identical with [56].

[C] composes two processes if their end-point types are disjoint.

[I], [I], [V], and [D] are standard. [NR] is the restriction rule for shared name a. In [I] and [V], "end only" means Δ only contains end as session types.

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 [C].

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 under Γ, which can be composed to make the initial configuration.

Delegation One source of the expressiveness of the session types comes from a facility of *delegation* (often called *higher-order session passing*). We will type and see the

relationship with global and end-point types. Consider the following three participants:

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

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:

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

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

Let (Alice | Bob | Carol) $\to (\nu\tilde{t}\tilde{s})(A \mid B \mid C \mid R)$ where A, B, C are the processes of Alice, Bob and Carol after 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 $P \equiv \mathbf{0}$:

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

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

5 Safety and Progress

This section establishes the fundamental 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 end-point types.
3. We then relate the reduction of processes and that of typings: showing the latter follows the former gives us *subject reduction* (Theorem 5.22), *safety* (Theorem 5.24) and *session fidelity* (Corollary 5.25), while showing the former follows the latter under a certain condition gives us *progress* (Theorem 5.32).

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

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

5.1 Typing Runtime

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

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

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

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

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 an end-point type with a hole (a type context) for typing a queue. For example we type the queue in (6) as:

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

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 $k : l_1 \cdot \mathsf{true} \cdot l_2$ can be typed with

$$k \oplus l_1 : \; k!\langle \mathsf{bool}\rangle; \; k \oplus l_2 : [\;],\tag{8}$$

omitting braces for a singleton selection. Now consider reduction

$$s_i \lhd \mathsf{ok}; P \mid s_i : \emptyset \quad \rightarrow \quad P \mid s_i : \mathsf{ok}. \tag{9}$$

Assume we type the left-hand side as

$$\tilde{s} : k \oplus \{\mathsf{ok} : T, \mathsf{quit} : T'\}@\mathsf{p}. \tag{10}$$

After the reduction, we obtain the type for P as

$$\tilde{s} : T@\mathsf{p}. \tag{11}$$

and the type for the queue as:

$$\tilde{s} : k \oplus \{\mathsf{ok} : [\;\;]\}@\mathsf{p}. \tag{12}$$

By combining (11) and (12) as before, we obtain

$$\tilde{s} : k \oplus \{\mathsf{ok} : T\}@\mathsf{p}. \tag{13}$$

We now observe that the located type in (13) is a *subtype* of the located type in (10) in the standard session subtyping [8, 19], which is formally defined as:

Definition 5.1 *The* subtyping *over end-point types, denoted* $\leq_{\mathsf{sub}}$*, is the maximal fixed point of the function* S *that maps each binary relation* $\mathcal{R}$ *on end-point 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* $\&\{l_j : T_j\}_{i\in J}\ S(\mathcal{R})\ \&\{l_i : T'_i\}_{i\in I}$.

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

Since $k \oplus \{\mathsf{ok} : T\} \leq_{\mathsf{sub}} k \oplus \{\mathsf{ok} : T, \mathsf{quit} : T'\}$, we can type the reductum in (9) using the located type given in (10), which is a supertype of the located type in (13), through the standard subsumption, achieving the required rollback.

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

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

Thus a type context represents a sequence of outputs and singleton selections which ends with a hole. As before, the notation "Δ, Δ'" denotes the union assuming the domains should not include a common channel name. The *isomorphism* $\approx$ *on type contexts* is generated from permutations given in § 4.2 (3, 4). Each assignment in Δ may contain both end-point types and type contexts. We define the partial commutative algebra $\circ$ as follows ($\mathsf{sid}(\mathcal{T})$ denotes the channel numbers in $\mathcal{T}$).

$$\begin{array}{c} T \circ \mathcal{T} = \mathcal{T} \circ T = \mathcal{T}[T] \\ \mathcal{T} \circ \mathcal{T}' = \mathcal{T}[\mathcal{T}'] \quad (\mathsf{sid}(\mathcal{T}) \cap \mathsf{sid}(\mathcal{T}') = \emptyset) \end{array}$$

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 $\{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I}$ by $\circ$ is given point-wise.

Below we define a simple algebra of environments for runtime processes.

Definition 5.2 (type algebra) A partial operator $\circ$ is defined as:

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

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

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

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

Fig. 8 Selected Typing Rules for Runtime Processes

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

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

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

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

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

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

5.2 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. [S] allows subsumption ($\leq_{\mathsf{sub}}$ is extended pointwise from types). [Q] starts from the empty hole for each participant, recording the session channel in the judgement. [Q] says when we enqueue $\tilde{v}$, the type for $\tilde{v}$ is added at the tail. [Q] and [Q] are the corresponding rules for delegated channels and a label.

[I] allows weakening for empty queue types, while [C] 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 [CR], 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.

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

- For [M],[M],[R],[SR],[B] and [D], we replace $\Gamma \vdash P \triangleright \Delta$ with $\Gamma \vdash P \triangleright_{\emptyset} \Delta$.
- [V] is similar to [I] (so that a queue can never occur in processes realising participants).
- For both [D] and [NR], we replace $\Gamma \vdash P \triangleright \Delta$ by $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$.

Using these typing rules, we can check that the configurations at the beginning of this section, (5) and (6), are given an identical typing by "rolling back" the type of the message in the queues; similarly for the next redex and reductum pair in the same page, (10) and (11).

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

Proposition 5.3 *Let P be a program phrase and Δ be without a type context. Then $\Gamma \vdash P \triangleright \Delta$ in the typing system in § 4 iff $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ is derived without using [S   ] in the typing system in this section.*

Proof. See Appendix B.

Proposition 5.4 *If $\Gamma \vdash P \triangleright_{s[1..m]} \Delta$ then P has a unique queue at $s[i]$ $(1 \leq i \leq m)$, no other queue at a free channel occurs in P, and no queue in P is under any prefix.*

Proof. By mechanical rule induction, see Appendix B.2. □

5.3 Type Reduction

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

Definition 5.5 (type reduction) We generate $\Delta \rightarrow \Delta'$ by the following rule:

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

$$k \oplus \{l : H, ...\}@\mathtt{p}, k\&\{l : T, ...\}@\mathtt{q} \xrightarrow{k} H@\mathtt{p}, T@\mathtt{q} \qquad \text{[TR-B]}$$

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

$$\frac{\Delta \approx \Delta_0 \quad \Delta_0 \xrightarrow{s[k]} \Delta_0' \quad \Delta_0' \approx \Delta'}{\Delta \xrightarrow{s[k]} \Delta'} \qquad \text{[TR-I]}$$

For analysing properties of type reduction, in particular with respect to causality in coherent global types, we introduce several ideas.

Definition 5.6 1. (full projection) Assume G is coherent. Then *full projection of G*, denoted by $[\![G]\!]$ is defined as the family $\{(G \upharpoonright \mathtt{p})@\mathtt{p} \mid \mathtt{p} \in \mathsf{pid}(G)\}$.
2. (causal edges on $[\![G]\!]$) For $[\![G]\!]$ given above, regarding each type in $[\![G]\!]$ as the corresponding regular tree, we define the causal edges $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$ and $\prec_{\mathtt{OO}}$ among its prefixes precisely we have done in G.

Proposition 5.7 *Each causal edge in G is preserved through the projection onto $[\![G]\!]$.*

Proof. A casual edge between two nodes of a global type is either $\mathtt{II}$, $\mathtt{IO}$ or $\mathtt{OO}$. For example, the shortest path that contains an $\mathtt{II}$-dependency takes a form of: $\mathtt{p}' \to \mathtt{p} : k.\mathtt{p}'' \to \mathtt{p} : k'.G$. and the projection of this path on participants is $\{k!\langle\rangle; G \upharpoonright \mathtt{p}'@\mathtt{p}', k?\langle\rangle; k'?\langle\rangle; G \upharpoonright \mathtt{p}@\mathtt{p}, k'!\langle\rangle; G \upharpoonright \mathtt{p}''@\mathtt{p}''\}$. The order of receiving two messages by $\mathtt{p}$ in the end-point types is the same as in the path. We can also prove that the projection preserves the order of messages received by each participant for other dependencies. □

Definition 5.8 We define $G \xrightarrow{k} G'$ if $[\![G]\!] \xrightarrow{k} [\![G']\!]$. 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'.

Definition 5.9 (merge set) *Assume G is coherent. Then we say two prefixes in G in different branches of a branching prefix are* mergeable *with each other when they are collapsed in its projection. A prefix is always mergeable with itself. Given a prefix* n*, its* merge set *is the set of prefixes mergeable with* n*.*

Proposition 5.10 *Two prefixes in G are mergeable iff they are related to one common input prefix and one common output prefix in* $[\![G]\!]$ *through projection.*

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

Corollary 5.11 *There is a bijection between the merge sets in G and the set of input prefixes in* $[\![G]\!]$*. Similarly for resp. the set of output prefixes in* $[\![G]\!]$*.*

Proof. This is immediate from Proposition 5.10, taking the map induced by the projection as the required bijection. □

Definition 5.12 *If a merge set of G is related to an input prefix and an output prefix in* $[\![G]\!]$ *by the bijection noted in Corollary 5.11 above, we say the former is the* projection preimage *or simply* preimage *of the latter, or the latter is the* projection image *or* image *of the former.*

The following result links the linearity in global types to the causal properties in end-point types.

Proposition 5.13

(1) *If a pair of prefixes in* $[\![G]\!]$ *(taken up to the type isomorphism* $\approx$*) form a redex with respect to* $\to$ *then they are not suppressed by any of* $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$ *and* $\prec_{\mathtt{OO}}$*.*
(2) *Given coherent G, let* G' *be the result of taking off the merge set of a prefix from G which is not suppressed by any of* $\prec_{\mathtt{II}}$, $\prec_{\mathtt{IO}}$ *and* $\prec_{\mathtt{OO}}$*. Then* G' *is again coherent.*
(3) *Let G be coherent. Then the causal edges are preserved and reflected between the two merge sets in G and their images in* $[\![G]\!]$*. Further each redex pair in* $[\![G]\!]$ *is the image of some prefix in G.*

Proof. See Appendix B.3. □

Proposition 5.13 (1), (2) and (3) together say that the "redexes" in a global type (in the sense that they are not under any non-trivial IO, OO or II-dependency) is exactly reflected onto those in its projection.

We can now clarify the relationship between reduction of typings and coherence, after a definition.

Definition 5.14 (coherence and partial coherence of typings) (1) We say Δ *is coherent* if $\Delta(\tilde{s})$ is coherent for each $\tilde{s} \in \mathsf{dom}(\Delta)$. (2) Δ is *partially coherent* if for some Δ' we have $\Delta \asymp \Delta'$ and $\Delta \circ \Delta'$ is coherent.

Proposition 5.15

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

Proof. For (1) suppose $\Delta_1 \stackrel{s}{\to} \Delta'_1$ and $\Delta_1 \asymp \Delta_2$. Note $\Delta_1 \asymp \Delta_2$ means that each pair of vectors of channels from $\Delta_{1,2}$ either coincide or are disjoint, and that, if they coincide, their image are participant-wise composable by $\circ$. Since no typed reduction rule invalidate either condition we conclude $\Delta'_1 \asymp \Delta_2$.

For (2), suppose Δ is coherent and $\Delta \stackrel{s}{\to} \Delta'$. Suppose the associated redex is in $\Delta(\tilde{s})$. By coherence we can write $\Delta(\tilde{s})$ as $[\![G]\!]$ for some coherent G. Now consider the preimage of the associated redex in $[\![G]\!]$, whose existence is guaranteed by Proposition 5.13 (3). This preimage is not suppressed by causal edges by Proposition 5.13 (1,3). Reducing $[\![G]\!]$ corresponds to eliminating its preimage from G, say G', whose projection $[\![G']\!]$ precisely gives the result of reducing $[\![G]\!]$. Since G' is coherent by Proposition 5.13 (2) we are done.

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

Finally the only if-direction of (4) follows directly from Definition 5.8 and that $\Delta \stackrel{s[k]}{\to} \Delta'$ implies $\Delta(\tilde{s}) \stackrel{s[k]}{\to} \Delta'(\tilde{s})$. For the if-direction, by Proposition 5.13(3), the casual edges of G are preserved in the images of $[\![G]\!]$ and by Corollary 5.11, there is a bijection between prefixes in G and the set of input-output prefixes in $[\![G]\!]$. Hence $[\![G]\!] \stackrel{s[k]}{\to} [\![G']\!]$ where $\Delta(\tilde{s}) = [\![G]\!]$ and $\Delta'(\tilde{s}) = [\![G']\!]$. Then we apply [TR-C]. □

5.4 Subject Reduction and Communication Safety

For subject reduction we use the following lemmas. In the first lemma below, we say that two typings, Δ_1 and Δ_2, *share a common target channel in their type contexts* when, for some $\tilde{s}$ and k, we have: (1) $\mathcal{T}_1@\mathtt{p} \in \Delta_1(\tilde{s})$ and $\mathcal{T}_2@\mathtt{p} \in \Delta_2(\tilde{s})$; and (2) $k!\langle U\rangle$ or $k \oplus l$ occurs in $\mathcal{T}_1$ and $k!\langle U'\rangle$ or $k \oplus l'$ occurs in $\mathcal{T}_2$ (i.e. they have an output/selection type at a shared channel).[8]

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

Lemma 5.16 (partial commutativity and associativity of $\circ$) *$\circ$ on typings is partially commutative and associative with identity $\emptyset$ under the condition that, whenever we compose two typings, they never share a target channel in their type contexts (in the above sense).*

Proof. See Appendix B.4.

Lemma 5.17 *Assume $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. Then all free names and free variables in P occur in Γ and all free channels in P occur in Δ.*

Proof. Mechanical. □

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

Lemma 5.18 (permutation) (1) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [S] at its last two steps. Then $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ has a derivation identical with the original one except its last two steps are replaced by a single application of [S].* (2) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [S] as its last rule and another rule which is not one of [S], [S] and [B]. Then $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ has a derivation which is identical with the original one except that the last two rules used are permuted.*

Proof. (1) is immediate from the transitivity of [S]. (2) is mechanical. □

Lemma 5.19 (queue) *The following rules are admissible in the typing system for runtime processes. Below let $\tilde{s} = s[1..k..n]$ and assume in each rule $\circ$ in the premise is well-defined.*

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

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

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

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

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

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

Proof. See Appendix B.2. □

Below we do not require the substitution lemmas for session channels and process variables, cf. [56].

Lemma 5.20 (substitution and weakening) (1) *(substitution)* $\Gamma, x : S \vdash P \triangleright_{\tilde{s}} \Delta$ *and* $\Gamma \vdash v : S$ *imply* $\Gamma \vdash P[v/x] \triangleright_{\tilde{s}} \Delta$. (2) *(weakening) Whenever* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *is derivable then its weakening,* $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta, \Delta'$ *for disjoint* Δ' *where* Δ' *contains only empty type contexts and for types* end*, is also derivable.*

Proof. Standard, see [56]. □

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

Remark 5.21 Theorem 5.22 (3) and the subsequent results (in particular Theorems 5.24 and 5.32 below) tell us, through Proposition 5.3, that the typing system in § 4, which is for programs and program phrases, guarantees type safety and other significant behavioural properties for typable programs.

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

Theorem 5.22 (subject congruence and reduction)

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

Proof. See Appendix B.5.

Theorem 5.22 immediately entails the lack of the standard type errors in expressions (such as true + 3). The type discipline also satisfies, as in the preceding session type disciplines [24], communication error freedom, including linear usage of channels. We first introduce the reduction context $\mathcal{E}$ as follows:

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

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 [D]. We write $P\langle s \rangle$ if P contains an active subject at s after applying [D], and $P\langle s! \rangle$ (resp. $P\langle s? \rangle$) 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.

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

Lemma 5.23 *Assume* $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ *s.t.* $\Delta \circ \Delta_0$ *is coherent for some* Δ_0.

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

Proof. By easy rule induction, see Appendix B.7.

The following result decomposes the standard property for synchronous session types [24, 49, 56] into the sending side and the receiving side, due to the existence of queues. We assume the standard bound name convention.

Theorem 5.24 (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) $P\langle s?\rangle$, s *occurs exactly once in* $\mathcal{E}$ *and* $\tilde{h} \neq \emptyset$; *or*
   (b) $P\langle s!\rangle$ *and* s *occurs exactly once in* $\mathcal{E}$; *or*
   (c) $P\langle s?\rangle$, $P\langle s!\rangle$, *and* s *occurs exactly twice in* $\mathcal{E}$.
2. *(error-freedom) if* $P \equiv \mathcal{E}[R]$ *with* $R\langle s?\rangle$ *being a redex:*
   (a) *If* $R \equiv s?(\tilde{y}); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{v} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{v}| = |\tilde{y}|$.
   (b) *If* $R \equiv s?(\tilde{s}); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{t} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{s}| = |\tilde{t}|$.
   (c) *If* $R \equiv s \triangleright \{l_i : Q_i\}_{i \in I}$ *then* $P \equiv \mathcal{E}'[s : l_j \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $j \in I$.

Proof. For (1), let $P \equiv (\nu \tilde{n})(P_0 | s : \tilde{h} | Q)$ where P_0 does not contain a queue and Q only contains queues (by Proposition 5.4). By Lemma 5.23 we know P_0 has either a single active prefix or a pair of a receiving active prefix and an emitting active prefix. So we have three cases:

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

Hence as required.

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

By Theorems 5.22 and 5.24, a typed process "never goes wrong" in the sense that its interaction at a multiparty session channel is always one-to-one and that each delivered value matches the receiving prefix. By Lemma 5.23 (2) and by the typing of the associated queue, this delivery precisely corresponds to a redex in the session typing.

As the corollary of Theorem 5.22(2) and Proposition 5.15(4), we obtain *session fidelity*: the interactions of a typable process exactly follow the specification described by its global type.

Corollary 5.25 (session fidelity) *Assume $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ such that Δ is coherent and $\Delta(\tilde{s}) = [\![G]\!]$. If*

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

Proof. In (1), the conclusion $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ where $\Delta = \Delta'$ or $\Delta \stackrel{s[k]}{\to} \Delta'$ follows directly from the subject reduction theorem 5.22(2). Then second conclusions $G \stackrel{k}{\to} G'$ and $[\![G']\!] = \Delta'(\tilde{s})$ follow directly from Proposition 5.15 (4). If not, a sender puts some value in the queue. Hence (2) obviously holds. □

5.5 Progress

Communication safety says that if a process ever does a reduction, it conforms to the typing and it is linear. If interactions within a session are not hindered by 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 [15, 24]. First we define:

Definition 5.26 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 5.27 (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.

Proposition 5.28 *Let P_0 be simple and $P_0 \to^* P$. Then no delegation prefix (input or output) occurs in P and for each prefix with a shared name in P, say $a[\mathtt{i}](\tilde{s}).P'$ or $\overline{a}[2..\mathtt{n}](\tilde{s}).P'$, there is no free session channels in P' except $\tilde{s}$.*

Proof. See Appendix B.8.

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

Definition 5.29 (well-linked) We say P is *well-linked* when for each $P \rightarrow^* Q$, 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 5.30 *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 $P\langle s? \rangle$ (resp. either $P\langle s! \rangle$ or the queue at s is not empty).*

Proof. By rule induction using Proposition 5.28, see Appendix B.9. □

Proposition 5.31 *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 $P \rightarrow P'$ for some P'.*
2. *If $\Delta(\tilde{t}) = [\![G]\!]$ and $G \stackrel{k}{\rightarrow} G'$, then $P \rightarrow^+ P'$ at the redex at t_k s. t. $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ with $\Delta'(\tilde{t}) = [\![G']\!]$.*

Proof. Let P be simple, queue-full and well-linked, and $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ such that Δ is coherent. Without loss of generality we can assume P does not have hidings (we can just take off and the result is again simple, queue-full, well-linked and coherent). Since Δ is coherent, if Δ contains any prefix then, by Proposition 5.28, it should form a redex (together with another prefix to form the image of a merge set). By Lemma 5.30 and Theorem 5.24 (1,2) and by the well-linkedness, either there is an if-branch above the prefix or P has an active prefix (or prefixes) at s in P. For the former, this if-branch itself cannot be under any prefix since that violates the activeness at s in Δ. So this if-branch can reduce hence done.

If not then by Lemma 5.30 there are the following cases:

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

In each case there is a reduction hence done. □

(2) above gives the converse of Corollary 5.25: if the global type has a reduction, then the process can always realise it.

Theorem 5.32 (progress) Let P be a simple and well-linked program. Then P has the *progress property* in the sense that $P \rightarrow^* P'$ implies either $P' \equiv \mathbf{0}$ or $P' \rightarrow P''$ for some P''.

Proof. Immediate from Proposition 5.28, Lemma 5.30 and Theorem 5.31. □

A simple application of Theorems 5.22 (3), 5.24 and 5.32 for processes from §2.3 follow. Below *communication mismatch* stands for the violation of the conditions given in Theorem 5.24 (2).

Proposition 5.33 (properties of two protocols)

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

Proof. Immediate from Theorem 5.32 because these two configurations are typable programs each of which loses its shared name in the initial reduction (at which point all the occurrences of the shared name are used). □

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

6 Extensions and Related Work

We discuss several extensions and directly related works.

6.1 Extensions

Existing Extensions of Session Types In the literature, several extensions of binary session type disciplines have been proposed, including subtyping [19], bounded polymorphism [18], integration with security annotations to guarantee authentication properties [4], and integration with higher-order π-calculus [34]. We believe that integrations with these extensions should be possible and will enrich expressive power and applicability of the theory.

Multithreaded Participant Another straightforward extension is to allow a multithreaded participant, so that a single participant can perform parallel conversations with others during a session. For this extension we add the parallel composition (T_1, T_2) to the grammar of end-point types, equipped with the following isomorphism (using type contexts in §5): $\mathcal{T}[T_1], T_2 \approx \mathcal{T}[T_1, T_2]$ if for no k there is an output at k in both $\mathcal{T}$ and T_2 (such a prefix adds false OO-dependency), as well as commutativity and associativity. Linearity between T_1 and T_2 in T_1, T_2 is given by coherence via projection.

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. Our recent work [35] studies a generation of graph-based global types from end-point types.

Synchrony and Asynchrony Most of the session types currently studied are binary and synchronous [24]. 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. Our subsequent work [2] studies a synchronous multiparty session type.

A different direction is to consider asynchronous message passing without order-preservation [23] 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 $\mathtt{p} \to \mathtt{p}_1, .., \mathtt{p}_\mathtt{n} : \{k_1, ..., k_n\}$ (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 end-point session types to $\tilde{k}!\langle U \rangle$ from $k!\langle U \rangle$. Causality analysis remains the same by decomposing each multicasting prefix into its unicasting elements and considering causality for each of them. Our subsequent work [2, 3] use multicasting and prove the subject reduction theorems in asynchronous and synchronous multiparty sessions.

6.2 Related Work

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

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 [39]. Recently [20] studies the asynchronous version of binary sessions for an ML-like language based on [51]. In [20], a message queue is given two endpoint channels and a direction.

The work [12] studies the asynchronous binary session types for Java, extending the previous work in [15], 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 [12]. Recently [14] 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; on the other hand, the system in [14] does not require the simpleness condition (Definition 5.27). A combination of these two typing systems will enlarge typability, guaranteeing the progress in many situations: recent work [3] published after the extended abstract of this paper [26] develops a progress typing system for multiparty sessions based on the technique in [14].

The concurrent work done by [5], 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) end-point types. In type checking, end-point types are projected to each binary session, so that type safety can be ensured using duality. Since we lose sequencing information in this way, the progress property is not guaranteed. The use of global types in the present work leads to transparent treatment of type structures such as recursion, the guarantee of stronger behavioural properties such as progress, and (arguably) more intelligible description of multiparty interaction structures.

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 [8] and by [13]. [8] 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 end-point processes. The use of global description for *types* and its associated theories have not been developed in [8]. The type disciplines for the two (global and end-point) calculi studied in [8] are based on binary synchronous session types, hence safety and progress for multiparty interactions are not considered.

[13] 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 [13] 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.

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

Semantics of Delegation The present work uses, for a simpler presentation, the operational semantics of delegation from [24] 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 [19, 56] allows substitution of session channels as in [R   ], which also satisfies type safety and progress through annotations on channels and types. This annotation extends the method in [19, 56]: instead of polarities we use indices by participants to annotate each usage of channels. With this change the whole theories remain intact with exactly the same operational semantics and typing for programs. We study this delegation in [2, 3].

Linear and Behavioural Types for Mobile Processes Among many works on types for mobile processes, session type disciplines in general and the present work in particular are most closely related with linear/IO-typed π-calculi with causality information. The session type disciplines are related with linear and IO-typed π-calculi with causality information. The causality analysis in global types is partly inspired by the graph-based linear types developed in [54, 55] where ordering among multiple linear names (which correspond to session channels) guarantees deadlock-freedom of typed processes. Several work [28, 29] study generalised forms of linear typing for guaranteeing different kinds of deadlock-freedom, incorporating synchronisations and locking.

A main difference of session type disciplines from these and other preceding works in this field is a notion of rigorously structured conversations and their direct type abstraction. See [1, 14] for detailed discussions, including comparisons between the session-based and the behavoural-based ones [29, 54, 55]; in [1, 3, 14], structured session primitives help to give simpler typing systems for progress for binary sessions.

By raising the level of abstraction through the use of structured primitives such as separate session initiation, branching and recursion, session types can describe complex interaction structures more intelligibly and enable efficient type checking. These features would have direct applicability for the design of programming languages with communication [8, 25, 27, 44, 45, 47].

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.

Advanced Process Calculi and Types Several process calculi for broadcasting have been investigated to model and analyse broadcasting networks including (recently) mobile ad-hoc networks, starting from Prasad's thesis [42]. Recent works focus on behavioural equivalences with lts [31, 32, 43] and static analysis [37] to investigate a number of different broadcasting. None of them studied the typing system and provided a strong progress guarantee as ensured by our session types. Our session types use a static participant information in the syntax and types. Recent advanced typing systems for location-based distributed processes [22] 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 $\mathtt{p}$ in the syntax) by using channel dependent types [34] or polymorphism.

Other Recent Service-Oriented Calculi Different approaches to the description of service-oriented multiparty communications are taken in [6, 7]. In [6], the global and local views of protocols are described using a synchronous CCS-based calculus as a contract language, and testing-preorders to check subcontract compliance; [7] proposes a distributed calculus which provides communications either inside sessions or inside locations, modelling merging running sessions. Another recent work [52] presents a calculus for service orientations by extending the π-calculus with context-sensitive interactions, equipped with service and request primitives and local exceptions. The study of formal theories of contracts are studied in [11, 40] using CCS-like processes as a type representation. These recent works do not treat a framework in the present work – a prescription of protocols by the global types, their end-point projections, backed-up by the efficient projection and type-checking, ensuring strong safety properties based on the session type discipline. Our subsequent work on synchronous multiparty session types [2], advanced progress [3] and asynchronous commutative multiparty session types for a refinament [35] demonstrate generality and applicability of the framework developed in this paper.

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. We proposed a new efficient

type checking system and proved type safety and progress, extended to multiparty interactions. The central technical underpinning of the present work is the introduction of global types, 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 [53] and a message scheme for financial protocols, UNIFI from ISO [50]. We are currently designing and implementing a modelling and specification language with multiparty session types [47] for these standards with our industry collaborators. This consists of three layers: the first layer is a global type which corresponds to a signature of class models in UML; the second one is for conversation models where signatures and variables for multiple conversations are integrated; and the third layer includes extensions of the existing languages (such as Java [27]) which implement conversation models. Others future topics 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.

References

1. Lucia Acciai and Michele Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 642–658. Springer, 2008.
2. Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. In *PLACES'08*, ENTCS. Elsevier. To appear.
3. Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
4. Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *JFP*, 15(2):219–248, 2005.
5. Eduardo Bonelli and Adriana B. Compagnoni. Multipoint session types for a distributed calculus. In *TGC*, volume 4912 of *LNCS*, pages 240–256. Springer, 2007.
6. Mario Bravetti and Gianluigi Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *Software Composition*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.
7. Roberto Bruni, Ivan Lanese, Hernan Melgratti, and Emilio Tuosto. Multiparty Sessions in SOC. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.
8. 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.
9. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2008.

10. 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.
11. Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *POPL*, pages 261–272, 2008.
12. 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.
13. Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. Secure implementations for typed session abstractions. In *CSF*, pages 170–186, 2007.
14. Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC*, volume 4912 of *LNCS*, pages 257–275. Springer, 2007.
15. 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.
16. 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.
17. Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In *PPDP'06*, pages 61–72. ACM Press, 2006.
18. Simon Gay. Bounded polymorphism in session types. *MSCS*, 18:895–930, 2008.
19. Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
20. Simon Gay and Vasco T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, may 2007.
21. Jean-Yves Girard. Linear logic. *TCS*, 50:1–102, 1987.
22. Matthew Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
23. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP*, volume 512, pages 133–147, 1991.
24. 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.
25. 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, 2007.
26. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
27. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In Jan Vitek, editor, *ECOOP*, volume 5142, pages 516–541. Springer, 2008.
28. Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
29. Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.
30. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
31. Massimo Merro. An observational theory for mobile ad hoc networks. In *Electronic Notes in Theoretical Computer Science*, volume 172, pages 275–293. Elsevier, 2007.
32. Nicola Mezzetti and Davide Sangiorgi. Towards a calculus for wireless systems. In *Electronic Notes in Theoretical Computer Science*, volume 158, pages 331–353. Elsevier, 2006.

33. Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION*, volume 5052 of *LNCS*, pages 216–231. Springer, 2008.
34. 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.
35. Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, LNCS. Springer, 2009. To appear.
36. On-line appendix of this paper. `http://www.doc.ic.ac.uk/~yoshida/multiparty`.
37. Sebastian Nanz, Flemming Nielson, and Hanne Riis Nielson. Topology-dependent abstractions of broadcast networks. In *CONCUR*, pages 226–240, 2007.
38. Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
39. Matthias Neubauer and Peter Thiemann. Session Types for Asynchronous Communication. Universität Freiburg, 2004.
40. Luca Padovani. Contract-directed synthesis of simple orchestrators. In *CONCUR*, volume 5201 of *LNCS*, pages 131–146, 2008.
41. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
42. K.V.S. Prasad. Broadcast calculus interpreted in ccs upto bisimulation. In *Electronic Notes in Theoretical Computer Science*, volume 52, pages 83–100. Elsevier, 2001.
43. K.V.S. Prasad. A prospectus for mobile broadcasting systems. In *Electronic Notes in Theoretical Computer Science*, volume 162, pages 295–300. Elsevier, 2006.
44. Riccardo Pucella and Jesse Tov. Haskell session types with (almost) no class. In *Haskell'08*. ACM SIGPLAN, 2008.
45. Matthew Sackman and Susan Eisenbach. Session types in haskell, 2008. draft.
46. Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.
47. Scribble. Scribble Project, 2008. `www.scribble.org`.
48. Stephen Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), March 2006.
49. 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.
50. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`.
51. Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.
52. Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283, 2008.
53. WS-CDL. Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.
54. Nobuko Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCS*, pages 371–386. Springer, 1996.
55. Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong Normalisation in the π-Calculus. In *Proc. LICS'01*, pages 311–322. IEEE, 2001. The full version in *Journal of Inf. & Comp.*., 191 (2004) 145–202, Elsevier.
56. Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

A Proof of Proposition 3.6

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

Notations A.1 (i) In the following we write $G(0)$, $G(1)$, ..., $G(n)$, ... for the result of n-times unfolding of each recursion in G. For example if G is $\mu\mathbf{t}.G'$ and this is the only recursion in G, then $G(0)$ is given as $G'[\mathsf{end}/\mathbf{t}]$, $G(1)$ is given as $G'[G(0)/\mathbf{t}]$ and, for each n, $G(n+1)$ is given as $G'[G(n)/\mathbf{t}]$. If G contains more than one recursion we perform the unfolding of each of its recursions. For convenience we set $G(-1)$ to be the empty graph.

(ii) Observing each $G(n+1)$ is the result of adding zero or more unfoldings to $G(n)$, so that $G(n+1)$ contains the exact copy of $G(n)$, we write $G(n+1)\setminus G(n)$ to denote the newly added (unfolded) part of $G(n+1)$.

(iii) Given a node n in $G(m+1)\setminus G(m)$, we can jump back from n once to reach its "original" in $G(m)\setminus G(m-1)$ (which is $G(0)$ if $m = 0$). This exact copy of n which was created "one unfolding ago", is called the *one-time folding of* n, or simply the *folding of* n. In the same way we define *the i-th folding of* n which is in $G(m-i+1)\setminus G(m-i)$ (which is $G(0)$ if $i = m+1$). Note there are $m+1$ such "foldings" of n in $G(m+1)\setminus G(m)$.

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

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

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

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

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

Induction Step. Suppose $G(n)$ is linear. Then take two nodes n_1 and n_2 in $G(n+1)$ (but not in carried types) which happen to share a common channel. We show there are input/output dependencies from n_1 to n_2, or the same holds in the reverse direction. We say such $\mathsf{n}_{1,2}$ are *conflict-free* for brevity. We do case analysis depending on the position of these nodes in $G(m+1)$.

(i) If $\mathsf{n}_{1,2}$ are in $G(n)$ then they already have input/output dependencies by induction hypothesis.

(ii) If n_1 is in $G(n)\setminus G(n-1)$ and n_2 is in $G(n+1)\setminus G(n)$ then take their two foldings say n'_1 and n'_2 respectively. By induction hypothesis they are conflict-free by a pair of dependency chains. By Claim A we are done.

(iii) If n_1 is in $G(n-i)$ $(i \geq 1)$ and n_2 is in $G(n+1)\backslash G(n)$ then take the folding of n_2 say n_2' which is in $G(n)$. By induction we know n_1 and n_2' are conflict-free.

By Claim B, there are both input and output dependencies from n_2 to n_2'. Thus we have both input and output dependencies from n_1 to n_2' and n_2' to n_2 (hence n_1 to n_2). Now we connect these chains and we are done. □

B Full Typing Rules for Runtime Processes

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

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

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

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

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

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

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

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

$$\frac{\Delta\ \mathsf{end\ only} \quad \Delta'\ [\ ]\ \text{only}}{\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \Delta, \Delta'} \qquad \frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'} \quad [\mathrm{I} \quad],[\mathrm{S} \quad]$$

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

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

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

The typing rules for queues are from Figure 8.

B.1 Proof of Proposition 5.3

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

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

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

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

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

B.2 Proof of Proposition 5.4

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

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

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

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

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

Case [M], [M], [S], [R], [D], [SR], [S] and [B], [I], [V], [D]: Each of these process construction rules leaves the queue channels unchanged (empty).

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

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

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

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

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

B.3 Proof of Proposition 5.13

For (1), observe that redexes in the base rules in Definition 5.5, [TR-C] and [TR-B], are in the minimal positions: thus we have only to validate the statement when a redex syntactically occurs below another prefix but gets permuted up by the type isomorphism, reflected in reduction by [TR-I]. We consider each kind of the causal edges.

Case (a): If a prefix suppresses another prefix (which is a redex, similarly in the following) through $\prec_{\mathtt{II}}$, then the latter has no chance to get permuted up.

Case (b): If a prefix suppresses another in $\prec_{\mathtt{IO}}$, then this is an input suppressing an output, so there is again no hope of permutation.

Case (c): If a prefix suppresses another in $\prec_{\mathtt{OO}}$, then this is precisely the case which does not allow permutation in the isomorphism so again there is no way that the suppressed can be permuted to a minimal node. This exhausts all cases.

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

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

B.4 Proof of Lemma 5.16

By the definition of $\circ$ on Δ, it suffices to show the commutativity and associativity at the level of types and type contexts, assuming that combined type contexts never share a target channel (in the sense defined just before Lemma 5.16, page 27).

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

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

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

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

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{14}$$

$$\mathcal{T}_1 \circ \mathcal{T}_2 = \mathcal{T}_1[\mathcal{T}_2] \tag{15}$$

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

$$\mathcal{T}_2 \asymp \mathcal{T}_1 \tag{16}$$

$$\mathcal{T}_2 \circ \mathcal{T}_1 = \mathcal{T}_2[\mathcal{T}_1] \tag{17}$$

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

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

Case Context-Type: Symmetric to the case above.

Case Type-Type: Never defined hence vacuous.

This exhausts all cases.

Next we show associativity.

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

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{18}$$

$$\mathcal{T}_1[\mathcal{T}_2] \asymp \mathcal{T}_3 \tag{19}$$

$$\mathcal{T}_1[\mathcal{T}_2] \circ \mathcal{T}_3 = \mathcal{T}_1[\mathcal{T}_2][\mathcal{T}_3] \tag{20}$$

By (19) we have:

$$\mathcal{T}_2 \asymp \mathcal{T}_3 \tag{21}$$

$$\mathcal{T}_1 \asymp \mathcal{T}_2[\mathcal{T}_3] \tag{22}$$

$$\mathcal{T}_1 \circ \mathcal{T}_2[\mathcal{T}_3] = \mathcal{T}_1[\mathcal{T}_2[\mathcal{T}_3]] \tag{23}$$

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

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

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{24}$$

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

$$\mathcal{T}_1[\mathcal{T}_2] \circ T = \mathcal{T}_1[\mathcal{T}_2][T] \tag{26}$$

We also always have:

$$\mathcal{T}_2 \asymp T \tag{27}$$

$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{28}$$

$$\mathcal{T}_1 \circ \mathcal{T}_2[T] = \mathcal{T}_1[\mathcal{T}_2[T]] \tag{29}$$

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

$$\mathcal{T}_2 \asymp T \tag{30}$$

$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{31}$$

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

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

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{33}$$

$$\mathcal{T}_1[\mathcal{T}_2] \asymp T \tag{34}$$

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

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

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

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

B.5 Proof of Subject Reduction Theorem (Theorem 5.22)

(1) is by rule induction on $\equiv$ showing, in both ways, that if one side has a typing then the other side has the same typing. In the following we safely ignore uninteresting (permutable) final applications of [S] in derivations by way of Lemma 5.18.

Case $P \mid \mathbf{0} \equiv P$: First assume $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. By $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \emptyset$ and by applying [C] to these two sequents we immediately obtain $\Gamma \vdash P|\mathbf{0} \triangleright_{\tilde{s}} \Delta$, as required. For the converse direction assume $\Gamma \vdash P|\mathbf{0} \triangleright_{\tilde{s}} \Delta$. We can safely assume (via Lemma 5.18) that the last rule applied is [C]. Thus we can set $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta_1$ and $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. Note we can safely regard $\Gamma \vdash \mathbf{0} \triangleright \Delta_2$ as being inferred by the axiom [I] since applying [S] to empty types and empty type contexts again lead to the empty types and empty type contexts: thus Δ_2 consists of only empty types and empty type contexts. Thus, in the composition $\Delta_1 \circ \Delta_2$, the empty types and some of the empty type contexts from Δ_2 are added to Δ_1 to generate Δ. Let this added part be Δ_2'. Since we can weaken Δ_1 in the first sequent with Δ_2' using Lemma 5.20 (2) we are done.

Case $P \mid Q \equiv Q \mid P$: By symmetry of the rule we have only to show one direction. Suppose $\Gamma \vdash P|Q \triangleright_{\tilde{s}} \Delta$. We can safely assume the last rule applied is [C]. We can thus set $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$ and $\Gamma \vdash Q \triangleright_{\tilde{r}} \Delta_2$ such that $\Delta_1 \asymp \Delta_2$, $\Delta_1 \circ \Delta_2 = \Delta$ and $\tilde{t} \uplus \tilde{r} = \tilde{s}$. By Lemma 5.16 we know $\Delta_2 \asymp \Delta_1$ and $\Delta_2 \circ \Delta_1 = \Delta$ hence by applying [C] with the

premises reversed we are done.

Case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$: By the establishment of the previous case again we have only to show one direction. Suppose $\Gamma \vdash (P \mid Q) \mid R \triangleright_{\tilde{s}} \Delta$. We can safely assume: $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$, $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta_2$ and $\Gamma \vdash P \triangleright_{\tilde{q}} \Delta_3$ such that $\Delta_1 \asymp \Delta_2$, $(\Delta_1 \circ \Delta_2) \asymp \Delta_3$ and $(\Delta_1 \circ \Delta_2) \circ \Delta_3 = \Delta$, as well as $\tilde{t} \uplus \tilde{r} \uplus \tilde{q} = \tilde{s}$. By the last condition, no two of Δ_1, Δ_2 and Δ_3 share a common target channel in their type contexts (in the sense given just before Lemma 5.16, page 27) because if the queue for a certain channel does not exist in a sequent then it cannot be used as a target channel in a type context in its typing. Thus we can apply Lemma 5.16 to know $\Delta_2 \asymp \Delta_3$, $\Delta_1 \asymp (\Delta_2 \circ \Delta_3)$ and $\Delta_1 \circ (\Delta_2 \circ \Delta_3) = \Delta$. By applying [C   ] in an appropriate order we are done.

The remaining rules are reasoned exactly as in [56] (note the arguments for congruence rules are direct from the compositionality of the typing rules). This concludes the proof of (1).

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

Claim. Suppose $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ and Δ is partially coherent (cf. Def. 5.14). Then $P \to P'$ implies $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta'$ such that either $\Delta \to \Delta'$ or $\Delta = \Delta'$.

All results on reduction on coherent typing is immediately applicable to partially coherent typing by Proposition 5.15 (1). Further by Proposition 5.15 (2), Δ' above is again partially coherent. Below we again ignore irrelevant final application of [S   ] through Lemma 5.18. All rule names are those of the typing rules .

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

$$\Gamma \vdash R \triangleright \Delta \tag{36}$$

By Lemma 5.17 we know $a \in \mathsf{dom}(\Gamma)$. Let $\Gamma(a) = G$. Since (36) can only be inferred by the sequence of [C   ] (up to permutable [S   ], similarly in the following), we know $\Gamma \vdash R_i \triangleright \Delta_i$ $(1 \le i \le n)$ such that $\Delta_1 \circ ... \circ \Delta_n = \Delta$. By [M   ] and [M   ] this means:

$$\Gamma \vdash P_i \triangleright \Delta_i, \tilde{s} : \{(G \upharpoonright i)@i\} \tag{37}$$

for each $1 \le i \le n$. Hence by the successive applications of [C   ] we reach:

$$\Gamma \vdash (\Pi_i P_i) \mid (\Pi_i s_i : \emptyset) \triangleright_{\tilde{s}} \Delta, \tilde{s} : \{(G \upharpoonright i)@i\}_{1 \le i \le n} \tag{38}$$

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

$$\Gamma \vdash (\nu \tilde{s})(\,(\Pi_i P_i) \mid (\Pi_i s_i : \emptyset)\,) \triangleright \Delta \tag{39}$$

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

Case [S   ]: We use the first rule of Lemma 5.19 for "rolling back" a message. Suppose we have:

$$\Gamma \vdash s!\langle \tilde{e} \rangle; P \mid s : \tilde{h} \triangleright_s \Delta \tag{40}$$

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

$$\Gamma \vdash s!\langle \tilde{e} \rangle; P \triangleright_{\emptyset} \Delta_1 \tag{41}$$

and

$$\Gamma \ \vdash \ s:\tilde{h} \ \triangleright_s \ \Delta_2 \tag{42}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (41) can only be inferred from [S] we know, first:

$$\Gamma \vdash e_j : S_j \tag{43}$$

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

$$\Delta_1 \ = \ \Delta_1' \circ \tilde{s} : k!\langle \tilde{S} \rangle; T@\mathtt{p} \tag{44}$$

and moreover

$$\Gamma \ \vdash \ P \ \triangleright_\emptyset \ \Delta_1' \circ \tilde{s} : T@\mathtt{p}. \tag{45}$$

On the other hand by $\Delta_1 \asymp \Delta_2$ and (42) we know:

$$\Delta_2 \ = \ \Delta_2' \ \circ \ \tilde{s} : \mathcal{T}@\mathtt{p} \tag{46}$$

Now assume $\tilde{e} \downarrow \tilde{v}$. Notice by (43) we have $\Gamma \vdash v_j : S_j$ for each v_j in $\tilde{v}$. Thus by Lemma 5.19, [Q], we infer:

$$\Gamma \ \vdash \ s:\tilde{h} \cdot \tilde{v} \ \triangleright \ \Delta_2' \ \circ \ \tilde{s} : \mathcal{T}[k!\langle \tilde{S} \rangle; [\ ]]@\mathtt{p}. \tag{47}$$

By the algebra of located types and type contexts:

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

Thus by applying [C] to (41) and (42) we obtain:

$$\Gamma \ \vdash \ P \mid s:\tilde{h} \cdot \tilde{v} \ \triangleright \ \Delta \tag{48}$$

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

Case [D]: Similar to [S] using the second rule of Lemma 5.19, see Appendix B.6.

Case [L]: We use the third rule of Lemma 5.19 together with the subtyping $\leq_{\mathsf{sub}}$. Suppose we have:

$$\Gamma \ \vdash \ s \lhd l; P \mid s:\tilde{h} \ \triangleright_s \ \Delta \tag{49}$$

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

$$\Gamma \ \vdash \ s \lhd l; P \ \triangleright_\emptyset \ \Delta_1 \tag{50}$$

and

$$\Gamma \ \vdash \ s:\tilde{h} \ \triangleright_s \ \Delta_2 \tag{51}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (50) can only be inferred from [S] as the last rule (up to permutable applications of [S]), we know, for some $\mathtt{p}$ and for some $\tilde{s}$ which includes s and for some $\{l_i\}$ which includes l,

$$\Delta_1 \;=\; \Delta'_1 \circ \tilde{s} : k \oplus \{l_i : T_i\}_{i\in I} @\mathtt{p} \tag{52}$$

and moreover

$$\Gamma \;\vdash\; P \;\triangleright_\emptyset\; \Delta'_1 \circ \tilde{s} : T_i @\mathtt{p}, \quad \text{for } i \in I. \tag{53}$$

On the other hand we can write:

$$\Delta_2 \;=\; \Delta'_2 \;\circ\; \tilde{s} : \mathcal{T} @\mathtt{p} \tag{54}$$

By (51), (54) and Lemma 5.19, [Q], we infer:

$$\Gamma \;\vdash\; s : \tilde{h} \cdot l \;\triangleright\; \Delta'_2 \;\circ\; \tilde{s} : \mathcal{T}[k \oplus l : [\;]] @\mathtt{p}. \tag{55}$$

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

$$\begin{array}{lll}(\Delta'_1 \circ \tilde{s} : T_i @\mathtt{p}) \;\circ\; (\Delta'_2 \circ \tilde{s} : \mathcal{T}[k \oplus l : [\;]] @\mathtt{p}) & & \\ & = & \Delta'_1 \circ \Delta'_2 \circ \tilde{s} : \mathcal{T}[k \oplus l : T_i] @\mathtt{p} \\ & \leq_{\mathsf{sub}} & \Delta'_1 \circ \Delta'_2 \circ \tilde{s} : \mathcal{T}[k \oplus \{l_i : T_i\}_{i\in I}] @\mathtt{p} \\ & = & (\Delta'_1 \circ \tilde{s} : k \oplus \{l_i : T_i\}_{i\in I} @\mathtt{p}) \circ (\Delta'_2 \circ \tilde{s} : \mathcal{T} @\mathtt{p}) \\ & = & \Delta \end{array}$$

Thus we obtain, by applying [C] to (53), (55) then applying [S] (the subsumption rule):

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

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

Case [R]: By the first of the latter three rules of Lemma 5.19 together with Lemma 5.20. Suppose

$$\Gamma \;\vdash\; s?(\tilde{x}); P \mid s : \tilde{v} \cdot \tilde{h} \;\triangleright_s\; \Delta \tag{57}$$

Since [C] is the only possible last rule (up to permutable [S]) we can set

$$\Gamma \;\vdash\; s?(\tilde{x}); P \;\triangleright_\emptyset\; \Delta_1 \tag{58}$$

and

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

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (58) can only be inferred from [R] we know, for some $\mathtt{p}$ and for some $\tilde{s}$ which includes s,

$$\Delta_1 \;=\; \Delta'_1 \circ \tilde{s} : k?\langle \tilde{S} \rangle; T @\mathtt{p} \tag{60}$$

and moreover

$$\Gamma, \tilde{x} : \tilde{S} \;\vdash\; P \;\triangleright_\emptyset\; \Delta'_1 \circ \tilde{s} : T @\mathtt{p}. \tag{61}$$

By Lemma 5.20, we obtain:

$$\Gamma \ \vdash \ P[\tilde{v}/\tilde{x}] \ \triangleright_{\emptyset} \ \Delta'_1 \circ \tilde{s} : T@\mathtt{p}. \tag{62}$$

Further by $\Delta_1 \asymp \Delta_2$ and (59) we know:

$$\Delta_2 \ = \ \Delta'_2 \ \circ \ \tilde{s} : k!\langle\tilde{S}\rangle.\mathcal{T}@\mathtt{p} \tag{63}$$

By Lemma 5.19, [Q], we infer:

$$\Gamma \ \vdash \ s : \tilde{h} \ \triangleright \ \Delta'_2 \ \circ \ \tilde{s} : \mathcal{T}@\mathtt{p}. \tag{64}$$

Using Proposition 5.15 we obtain:

$$\begin{aligned} \Delta \ \stackrel{\text{def}}{=} \ (\Delta'_1 \ \circ \ \tilde{s} : k?\langle\tilde{S}\rangle.T@\mathtt{p}) \circ (\Delta'_2 \ \circ \ \tilde{s} : k!\langle\tilde{S}\rangle.\mathcal{T}@\mathtt{p}) \\ \to \quad (\Delta'_1, \ \tilde{s} : T@\mathtt{p}) \circ (\Delta'_2 \ \circ \ \tilde{s} : \mathcal{T}@\mathtt{p}) \quad (\stackrel{\text{def}}{=} \Delta') \end{aligned}$$

Thus by applying [C] to (58) and (59) we obtain:

$$\Gamma \ \vdash \ P[\tilde{v}/\tilde{x}] \mid s : \tilde{h} \ \triangleright \ \Delta' \tag{65}$$

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

Case [SR], [B]: Similar to [R], using the latter two rules of Lemma 5.19, see Appendix B.6.

Case [I T], [I F], [D], [D]: Standard, cf. [56]. No difference in the typing.

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

$$\Gamma \ \vdash \ (\nu a)P \ \triangleright_{\tilde{s}} \ \Delta \tag{66}$$

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

$$\Gamma, a : \langle G\rangle \ \vdash \ P \ \triangleright_{\tilde{s}} \ \Delta. \tag{67}$$

By induction hypothesis we know

$$\Gamma, a : \langle G\rangle \ \vdash \ P' \ \triangleright_{\tilde{s}} \ \Delta' \tag{68}$$

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

$$\Gamma \ \vdash \ (\nu a)P' \ \triangleright_{\tilde{s}} \ \Delta' \tag{69}$$

as required. When session channels are hidden, suppose

$$\Gamma \ \vdash \ (\nu \tilde{s})P \ \triangleright_{\tilde{t}\setminus\tilde{s}} \ \Delta \tag{70}$$

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

$$\Gamma \ \vdash \ P \ \triangleright_{\tilde{t}} \ \Delta, \{s\} : \{T_p@p\}_{p\in I} \tag{71}$$

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

$$\Gamma \ \vdash \ P' \ \triangleright_{\tilde{t}} \ \Delta', \{s\} : \{T'_p@p\}_{p\in I} \tag{72}$$

where either $\Delta \rightarrow^{0,1} \Delta'$ or $\{s\} : \{T_p@p\}_{p\in I} \rightarrow^{0,1} \{s\} : \{T'_p@p\}_{p\in I}$. By Lemma 5.15 (1) $\{T'_p@p\}_{p\in I}$ is again coherent. Hence by [CR] we obtain

$$\Gamma \ \vdash \ (\nu\tilde{s})P' \ \triangleright_{\tilde{t}\setminus\tilde{s}} \ \Delta' \tag{73}$$

as required.

Case [P]: Suppose we have $\Gamma \ \vdash \ P|Q \ \triangleright_{\tilde{t}_1\cdot\tilde{t}_2} \ \Delta$ and $P \rightarrow P'$. By [C] we have $\Gamma \ \vdash \ P \ \triangleright_{\tilde{t}_1} \ \Delta_1$ and $\Gamma \ \vdash \ Q \ \triangleright_{\tilde{t}_2} \ \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. By induction hypothesis we have $\Gamma \ \vdash \ P' \ \triangleright_{\tilde{t}_1} \ \Delta'_1$ such that $\Delta_1 \rightarrow^{0,1} \Delta'_1$. By Proposition 5.15 (1) we have $\Delta'_1 \asymp \Delta_2$ hence $\Gamma \ \vdash \ P'|Q \ \triangleright_{\tilde{t}_1\cdot\tilde{t}_2} \ \Delta'_1 \circ \Delta_2$. Noting Proposition 5.15 (1) also says that $(\Delta_1 \circ \Delta_2) \rightarrow^{0,1} (\Delta'_1 \circ \Delta_2)$ we are done.

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

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

B.6 Remaining Cases of Theorem 5.22

Case [D]: We use the second rule of Lemma 5.19. Suppose we have:

$$\Gamma \ \vdash \ s!\langle\tilde{t}\rangle; P \mid s : \tilde{h} \ \triangleright_s \ \Delta \tag{74}$$

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

$$\Gamma \ \vdash \ s!\langle\tilde{t}\rangle; P \ \triangleright_{\emptyset} \ \Delta_1 \tag{75}$$

and

$$\Gamma \ \vdash \ s : \tilde{h} \ \triangleright_s \ \Delta_2 \tag{76}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (75) can only be inferred from [D] we know, for some p and for some $\tilde{s}$ which includes s,

$$\Delta_1 \ = \ \Delta'_1 \circ (\tilde{s} : k!\langle T'@\mathtt{p}'\rangle.T@\mathtt{p}, \ \tilde{t} : T'@\mathtt{p}') \tag{77}$$

and moreover

$$\Gamma \ \vdash \ P \ \triangleright_{\emptyset} \ \Delta'_1, \ \tilde{s} : T@\mathtt{p}. \tag{78}$$

On the other hand by $\Delta_1 \asymp \Delta_2$ and (42) we know:

$$\Delta_2 \ = \ \Delta'_2 \ \circ \ \tilde{s} : \mathcal{T}[\ ]@\mathtt{p} \tag{79}$$

By Lemma 5.19, [Q], we infer:

$$\Gamma \ \vdash \ s : \tilde{h} \cdot \tilde{t} \ \triangleright_{\tilde{s}'} \ \Delta'_2 \circ \tilde{s} : \{\mathcal{T}[k!\langle T@\mathtt{p}'\rangle.[\ ]]@\}, \ \tilde{t} : \{T@\mathtt{p}'\}. \tag{80}$$

By the algebra of located types and type contexts:

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

Thus by applying [C] to (75) and (76) we obtain:

$$\Gamma\ \vdash\ P \mid s\colon\tilde{h}\cdot\tilde{t}\ \triangleright\ \Delta \tag{81}$$

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

Case [SR]: By the second to the last rule of Lemma 5.19. Suppose

$$\Gamma\ \vdash\ s?(\tilde{t}); P \mid s\colon\tilde{t}\cdot\tilde{h}\ \triangleright_s\ \Delta \tag{82}$$

Since [C] is the only possible last rule (up to permutable [S]) we can set

$$\Gamma\ \vdash\ s?(\tilde{t}); P\ \triangleright_\emptyset\ \Delta_1 \tag{83}$$

and

$$\Gamma\ \vdash\ s\colon\tilde{t}\cdot\tilde{h}\ \triangleright_s\ \Delta_2 \tag{84}$$

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

$$\Delta_1\ =\ \Delta_1' \circ \tilde{s} : k?\langle T'@\mathtt{p}'\rangle.T@\mathtt{p} \tag{85}$$

and moreover

$$\Gamma\ \vdash\ P\ \triangleright_\emptyset\ \Delta_1' \circ \tilde{s}\colon T@\mathtt{p},\ \tilde{t}\colon T'@\mathtt{p}' \tag{86}$$

By $\Delta_1 \asymp \Delta_2$ and (84) we know:

$$\Delta_2\ =\ \Delta_2'\ \circ\ \tilde{s} : k!\langle T'@\mathtt{p}'\rangle.\mathcal{T}@\mathtt{p},\ \tilde{t} : T'@\mathtt{p}' \tag{87}$$

By Lemma 5.19, [Q], we infer:

$$\Gamma\ \vdash\ s\colon\tilde{h}\ \triangleright\ \Delta_2'\ \circ\ \tilde{s} : \mathcal{T}@\mathtt{p}. \tag{88}$$

Using Proposition 5.15 we obtain:

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

Thus by applying [C] to (83) and (84) we obtain:

$$\Gamma\ \vdash\ P \mid s\colon\tilde{h}\ \triangleright\ \Delta' \tag{89}$$

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

Case [B]: By the last rule of Lemma 5.19. Suppose

$$\Gamma \ \vdash \ s \triangleright \{l_i\colon P_i\}_{i\in I} \mid s\colon l_j \cdot \tilde{h} \ \triangleright_s \ \Delta \tag{90}$$

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

$$\Gamma \ \vdash \ s \triangleright \{l_i\colon P_i\}_{i\in I} \ \triangleright_\emptyset \ \Delta_1 \tag{91}$$

and

$$\Gamma \ \vdash \ s\colon l_j \cdot \tilde{h} \ \triangleright_s \ \Delta_2 \tag{92}$$

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

$$\Delta_2 \ = \ \Delta_2' \ \circ \ \tilde{s} : k \oplus l_j : \mathcal{T}@\mathtt{p} \tag{93}$$

where by assumption we have $j \in I$. Since (91) can only be inferred from [B] and by $\Delta_1 \asymp \Delta_2$, we also know:

$$\Delta_1 \ = \ \Delta_1' \circ \tilde{s}\colon k \ \& l_j : T_j@\mathtt{p} \tag{94}$$

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

$$\Gamma \ \vdash \ P_i \triangleright_\emptyset \Delta_1' \circ \tilde{s}\colon T_i@\mathtt{p} \tag{95}$$

for each $i \in I$ (so (94) is inferred using [S]). By Lemma 5.19, [Q], we infer:

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

Using Proposition 5.15 we obtain:

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

Thus by applying [C] to (91) and (92) we obtain:

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

such that $\Delta \rightarrow \Delta'$, as required. Again we need a reduction of typings. □

B.7 Proof of Lemma 5.23

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

Claim. Assume $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ s.t. Δ is partially coherent and there is no queue at s. Assume $P\langle s \rangle$. Then one of the following conditions holds.

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

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

Case [S], [R], [D], [SR], [S] and [B]: Immediate since there can only be a unique active channel name which is by the given prefixing.

Case [I], [I], [V], [D], [Q], [Q], [Q], [Q]: Vacuous.

Case [C]: Suppose

$$\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta, \qquad \Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta' \tag{98}$$

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

1. If only one party has an active prefix at s there is nothing to prove.
2. If both are active at s, suppose both processes, hence Δ and Δ', have receiving active prefixes at s. Then this cannot be partially coherent since if so then the assumed completion of $\Delta \circ \Delta'$ to a coherent typing should also contain two minimal receiving prefixes which are impossible by Proposition 5.13 (2, 3). Similarly when two include active emitting prefixes at s, hence as required

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

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

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

Case [S]: Vacuous again. □

B.8 Proof of Proposition 5.28

We show the following logically equivalent result:

Claim. (1) If P is simple then

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

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

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

Case [M]: The rule reads:

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

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

Case [M]: The rule reads:

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

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

Case [S]: The rule reads:

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

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

Case [R]: The rule reads:

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

Precisely the same as in [S] above.

Case [D]: The rule reads:

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

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

Case [SR]: The rule reads:

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

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

Case [S],[B]: Similar with [S] and [R].

Case [I], [C], [CR], [NR], [S], [D]: By the shape of these rules, in each rule, there is no addition or removal of a prefix from the premise to the conclusion. Hence both (1-a/b) are immediate from the induction hypothesis.

Case [I], [V], [Q], [Q], [Q], [Q]: Vacuous since no prefixes are involved.

Hence as required.

For (2) suppose a derivation of P is simple. By the proof of Theorem 5.22, if $P \to P'$ then we have essentially the same derivation for both P and P' except:

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

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

B.9 Proof of Lemma 5.30

Suppose:

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

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

Case [M], [M]: By Proposition 5.28 there can be no free session channels hence vacuous (since (C3) is not satisfied).

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

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

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

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

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

Same as [S].

Case [SR], [D]: By Proposition 5.28 these rules are not used in derivation of a simple process, hence vacuous.

Case [S],[B]: Similar with [S],[R].

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

Case [C]: The rule reads:

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

We first observe:

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

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

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

Above the details of the shape of a typing is in fact unnecessary.
Claim B. The composition | preserves activeness of each prefix.

This is immediate from the definition.

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

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

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

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

Case [D]: As [S] above.

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

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

Case [NR]: This reads:

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

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

Case [CR]: This reads:

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

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

This exhausts all cases. □