Multiparty Asynchronous Session Types

Kohei Honda, Nobuko Yoshida, and Marco Carbone

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

Contents

1 Introduction

Backgrounds Communication is becoming one of the central elements in software development, ranging from web services to business protocols to corporate applications integration 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 [1, 11, 12, 17, 36], higher-order processes [27], Ambients [10], multi-threaded ML [38], Haskell [29], F# [6], operating systems [9], Java [5, 7, 8, 19], CORBA [37], C++ [4], and Web Services [2, 3, 35, 39]. 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 conversation structure is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

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

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

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

Such explicit representation of conversation structure helps us deal with one of the most common bugs in programming with communication, the synchronisation bugs. A programmer expects that communicating programs should together realise a consistent conversation structure, but they easily fail to handle a specific incoming message or to send a message at the correct timing, with no way for her to detect such errors before runtime. An explicit specification as in (1) guides principled programming for communications and enables automatic protocol validation [39]. Further, a clean separation between abstraction and implementation given by type-based abstraction and associated primitives leads to intelligible programs and flexible implementations [19]. 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 (predictability).

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

As an example, let us consider a simple refinement of the above Buyer-Seller protocol: consider two buyers, Buyer1 and Buyer2, wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much s/he can pay, and Buyer2 either accepts the quote or receives the quote by notifying Seller. It is extremely awkward (if logically possible) to decompose this scenario into three binary sessions, between Buyer1 and Seller, between Buyer2 and Seller, and between Buyer1 and Buyer2. Abstracting this protocol as three separate session types 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 abstracted 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 [26], alleviating 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 [35, 39] through our collaboration as invited experts [2, 3, 18, 39], 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 [14]. Consider the binary session type given in (1) for Buyer. Not only Buyer's be-

haviour 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 (which is given by exchanging input and output and branching and selection in the original type). Thus a designer can describe the intended conversation structure for sure by a session type; and, when composing two parties, we only have to check they have mutually dual types to check their compatibility.

This framework based on duality is no longer effective in multiparty sessions. Here the whole conversation cannot be constructed only from the behaviour of a single participant. It is however the structure of this whole conversation which we wish to type abstract and check our programs against: as we already discussed, even a simplest example such as Two Buyer Protocol is hard to decompose into binary conversations. Thus we need an effective means to abstract, as a type, a multiparty interaction scenario which a designer/programmer wishes to realise, to check programs' correctness against that type, and to effectively calculate compatibility of programs to be composed.

Secondly, linearity analysis of channels, which is the key to ensure safety and progress, becomes highly involved under a combination of asynchrony and multiparties since a conflict of actions can arise more easily. For example, Alice may *send* her messages to Bob and Carol in this order but they may not *arrive* in the same order. On the other hand, if Alice sends her messages to Bob consecutively many times, and if the underlying asynchronous transport guarantees message-order preservation, these messages will arrive at Bob's without confusion, ensuring linear(ized) usage of channels. Thus it is both worthwhile and subtle to make the best of available causality principles, for a precise analysis for correct sequencing of actions distributed among multiple peers.

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

1. *A new notion of types* which directly abstract intended conversation structure among n-parties as global scenarios, retaining intuitive type syntax.
2. *Consistency criteria* for an asynchronous conversation structure given as a causality analysis of asynchronous 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 local projection onto individual local participants: the resulting local types are directly associated with individual processes for efficient type checking.

The idea of type abstraction based on a global view (Point 1) comes from an abstract version of "choreography" developed in a W3C web services working group [2, 3, 18, 39]. The safety principle for multiparty asynchrony (Point 2) precisely and modularly captures causality structures in asynchronous interactions in the abstract setting of global types, offering a foundation for the type discipline. Through the use of global types, we obtain 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 introduce a type discipline which can validate whether a program is typable or not, given G (as shared agreement) and an individual program (as its local realiser). The resulting type discipline guarantees all the original key properties of session types, such as communication error freedom and progress property (deadlock freedom), for general multiparty asynchronous sessions.

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.

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 [41] offers detailed discussions and analysis of their typing systems. We use a simple extension of the original language in [17, 36] 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 $a, b, x, y, z, \ldots$ (representing known names of endpoints); *session channels* or *channels*, ranged over by $s, t, ...$ (representing fresh ports belonging to each instance of a multiparty session); *labels*, ranged over by $l, l', \ldots$ (functioning like method names or labels in labelled records); *process variables*, ranged over by $X, Y, \ldots$ (used for representing recursive behaviour). In the syntax for hiding, we use n for either a single shared name or a vector of session channels. Then *processes*, ranged over by $P, Q \ldots$, and *expressions*, ranged over by $e, e', \ldots$, are given by the grammar in Figure 1.

Except for the first two primitives for session initiation and the final primitive representing a message queue, all constructs come from [17].

The first two constructs are for session initiation. The prefix $\overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P$ initiates a new session through a shared interaction point a, by distributing a vector of freshly generated session channels $\tilde{s}$ to the remaining $\mathtt{n}-1$ participants, each of shape $a_{[\mathtt{p}]}(\tilde{s}).Q_{\mathtt{p}}$, the second construct in the grammar, 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. These prefixes are chosen for representing the abstract notion of establishment of a multiparty session in bare minimal syntax.

Session communications (i.e. communications that take place inside an established session) are performed using the next three pairs of primitives: the sending and receiving of a vector of values, $s!\langle\tilde{e}\rangle;P$ and $s?(\tilde{x});P$; the session delegation and reception, $s!\langle\!\langle\tilde{s}\rangle\!\rangle;P$ and $s?(\!(\tilde{s})\!);P$ (where 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, $s \lhd l;P$ and $s \rhd \{l_i : P_i\}_{i\in I}$ (where 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

Fig. 1 Syntax

P	$::= \overline{a}_{[2..n]}(\tilde{s}).P$	multicast session request
	$\mid a_{[1]}(\tilde{s}).P$	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

messages in transit 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 of input and output when unnecessary.

There are the following binders in processes:

- $\tilde{s}$ in $\overline{a}_{[2..n]}(\tilde{s}).P$, $a_{[p]}(\tilde{s}).P$ and $s?(\!(\tilde{s})\!); P$;
- $\tilde{x}$ in $s?(\tilde{x}); P$;
- $\tilde{x}\tilde{s}$ in $X(\tilde{x}\tilde{s}) = P$;
- n in $(\nu n)P$ and
- the 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. For P a process, $\mathsf{fpv}(P)$, $\mathsf{fn}(P)$, and $\mathsf{fs}(P)$ respectively denote the sets of *free process variables*, *free identifiers* and *free session channels* in P. $\mathsf{dpv}(\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i \in I})$ denotes the set of *process variables* introduced in $\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i \in I}$.

Fig. 2 Structural congruence.

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

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

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

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

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

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

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).

Using $\equiv$, the operational semantics is given by the *reduction relation*, denoted $P \rightarrow Q$, 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 [LINK] 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.

Rules [SEND], [DELEG] and [LABEL] respectively enqueue values, channels and a label at the tail of the queue for s. Rules Symmetrically [RECV], [SREC] [3] and [BRANCH] dequeue, at the head of the queue, values, channels and a label. [RECV] and [BRANCH] respectively further instantiates the value in the body and selects the corresponding branch.

[3] This delegation rule (from [17]) is chosen over the more liberal one in [12, 13, 41] (which uses substitution as in [RECV]) for simpler presentation. The technical development does not depend on this choice, see §6.2.

Fig. 3 Reduction

$$\overline{a}_{[2..n]}(\tilde{s}).P_1 \mid a_{[2]}(\tilde{s}).P_2 \mid \cdots \mid a_{[n]}(\tilde{s}).P_n \;\to\; (\nu\,\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_n \mid s_1:\emptyset \mid \cdots \mid s_m:\emptyset) \quad \text{[LINK]}$$

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

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

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

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

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

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

$$\text{if } e \text{ then } P \text{ else } Q \;\to\; P \qquad (e\downarrow \text{true}) \qquad \text{[IFT]}$$

$$\text{if } e \text{ then } P \text{ else } Q \;\to\; Q \qquad (e\downarrow \text{false}) \qquad \text{[IFF]}$$

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

$$P \to P' \;\Rightarrow\; (\nu n)P \to (\nu n)P' \qquad \text{[SCOP]}$$

$$P \to P' \;\Rightarrow\; P \mid Q \to P' \mid Q \qquad \text{[PAR]}$$

$$P \to P' \;\Rightarrow\; \text{def } D \text{ in } P \to \text{def } D \text{ in } P' \qquad \text{[DEFIN]}$$

$$P \equiv P' \text{ and } P' \to Q' \text{ and } Q' \equiv Q \;\Rightarrow\; P \to Q \qquad \text{[STR]}$$

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 [LINK] generates a queue for each channel, these rules entail that:

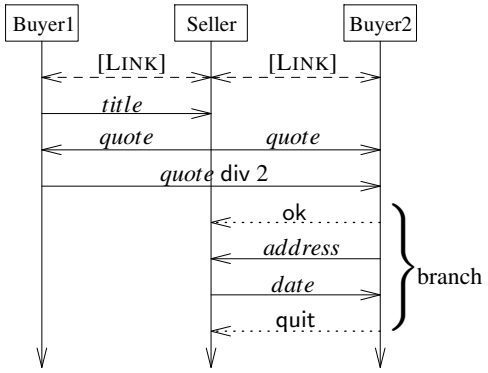
1. A sending action is never blocked (communication asynchrony); and that
2. two messages from the same sender to the same channel arrive in the sending order (message order preservation).

As we discussed in 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, [IFT] and [IFF], reduce to one of the branches depending on the result of evaluating the conditional guard. Rule [DEF] performs unfolding of recursion. Rules [SCOP], [PAR] and [DEFIN] close the reduction under hiding, parallel composition and definition. Finally Rule [STR] says that the reduction is defined over processes up to $\equiv$.

2.3 Examples

Two Buyer Protocol We describe the two-buyers-protocol from Introduction first by a sequence diagram below, 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 s/he can contribute, and Buyer2 notifies Seller if it accepts the quote or not. We now describe the behaviour of Buyer1 as a process:

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

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

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

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

Above we write $s_1..s_m!\langle v\rangle; P$ for $s_1!\langle v\rangle;..s_m!\langle v\rangle; P$, assuming $s_1..s_m$ are pairwise distinct.[4] We can now explain why Buyer2 needs to use two input channels,

[4] Due to asynchrony there is in effect no order among the sending actions at $s_1..s_m$, so that $s_1..s_m!\langle v\rangle; P$ act as multicasting of v to $s_1..s_n$.

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 [23]). Thus if we were to use b_2 for both actions, two messages can be confused, losing linear usage of a channel. Later we shall show such a session can be detected by our type discipline.

A Streaming Protocol We next consider a simple protocol for the standard stream cipher [34].

> 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. In the following description we only focus on communication behaviour. The kernel initiates a session:

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

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: [5]

$$\begin{array}{rl} \text{DataProducer} \stackrel{\text{def}}{=} & \mathsf{def}\ \mathrm{P}(d,k,c) = d!\langle data\rangle;\mathrm{P}\langle d,k,c\rangle\ \mathsf{in}\ a_{[2]}(d,k,c).\mathrm{P}\langle d,k,c\rangle \\ \text{Consumer} \stackrel{\text{def}}{=} & \mathsf{def}\ \mathrm{C}(d,k,c) = c?(data);\mathrm{C}\langle d,k,c\rangle\ \mathsf{in}\ a_{[3]}(d,k,c).\mathrm{C}\langle d,k,c\rangle \end{array}$$

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.

[5] 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 Global Types and Causal Analysis

Developing programs for multiparty sessions demands a clear formal design as to how multiple participants communicate and synchronise with each other. To program individual participants without such a design and to hope that 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 through duality, but this is no longer possible for multiparty sessions, since the whole conversation structure cannot be represented by that of a single endpoint. But it is this whole conversation structure which a designer wishes to realise and hence against which she wishes to check the correctness of her programs.

This is why we need the type abstraction which describes global conversation scenarios of multiparty sessions, introduced and studied in this section.

3.1 Session Types from a Global Viewpoint

Global session type, or *global type*, denoted $G, G', \ldots$, are intended to offer a concise syntax for describing the conversation scenario of a multiparty session. Its grammar is given in Figure 4. We illustrate its constructs in the following.

Type $\mathtt{p} \to \mathtt{p}' \colon k\,\langle U \rangle.G'$ says that participant $\mathtt{p}$ sends a message of type U to channel k (represented as a finite natural number) which is then received by participant $\mathtt{p}'$; and then interactions described in G' ensues. Symbols $U, \ldots$ range over *value types*, denoting types for message values. Each value type is either a vector of types for shared names called *sorts*, written $S, S', \ldots$, or a type for a vector of session channels written $T@\mathtt{p}$. The latter is discussed in detail in the next section. For the development in this section, it suffices to consider U as a single base type.

Type $\mathtt{p} \to \mathtt{p}' \colon k\,\{l_j \colon G_j\}_{j \in J}$ says participant $\mathtt{p}$ sends one of the labels to channel k which is then received by participant $\mathtt{p}'$. If l_j is sent, interactions described in G_j take place.

Type G, G' represents concurrent run of interactions specified by G and G'. Type $\mu \mathbf{t}.G$ is a recursive type for recurring conversation structures, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e. type variables only appear under the prefixes (hence contractive). We take an *equi-recursive* view of recursive types, not distinguishing between $\mu \mathbf{t}.G$ and its unfolding $G[\mu \mathbf{t}.G/\mathbf{t}]$ [31]. We assume that $\langle G \rangle$ in the grammar of sorts is closed, i.e. without type variables.[6] Type end represents the termination of the session. We identify "G, end" and "end, G" with G. The trailing end is sometimes omitted from examples.

[6] In the presence of the standard recursive sorts [17] which we omit for simpler presentation, we allow sort variables to occur in $\langle G \rangle$.

Fig. 4 Syntax of Global Types

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

Definition 3.1 (prefix) We say that 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. In the former U is a *carried type* of the prefix. 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.

Conventions 3.3 Henceforth we often regard a global type G as the acyclic directed graph given by the standard regular tree presentation [31] of G.

A basic ordering on the nodes of this graph is induced by prefixes.

Definition 3.4 (prefix ordering) Write $\mathtt{n}, \mathtt{n}', ..$ for prefixes occurring in a global type, say G (but not in its carried types), seen as nodes of G as a graph. We write $\mathtt{n} \in G$ when $\mathtt{n}$ occurs in G. Then we write $\mathtt{n}_1 \prec \mathtt{n}_2 \in G$ when $\mathtt{n}_1$ directly or indirectly prefixes $\mathtt{n}_2$ in G. Formally $\prec$ is the least partial order generated by:

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

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

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

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

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 prefixes ordered by* $\prec$. 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 [23]. The choice of this asynchronous interpretation of the prefix ordering is a natural outcome of asynchronous communication in our theory and enables us to capture causal ordering among asynchronous multiparty interactions generally and with precision. We now illustrate this idea with substantial examples.

3.2 Examples of Global Types

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

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

The type gives a vantage view of the whole conversation scenario. We show several salient points in the interpretation of this type.

- Consider Lines 3 and 4. Since they have different senders, the sending actions are unordered in spite of their $\prec$-ordering. *Hence if* $b_2 = b_2'$ *two messages can have a conflict at* s. Note this analysis carries out the preceding operational argument in §2.3 at the level of a global type.
- Consider the following causal chain of actions from Line 1 to 3 to 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. Further there is a direct dependency $\prec$ from S of Line 1 to S of Line 5, showing the receiving actions (or activeness in input channels) in Lines 1 and 5 are also ordered. Since actions take place in strict temporal order in both sending and receiving, no conflict occurs in spite of the use of a common channel s.

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

$$\begin{array}{l} 1\ \mu\mathbf{t}.\mathtt{DP} \to \mathtt{K}\colon d\,\langle \mathsf{bool}\rangle. \\ 2\ \mathtt{KP} \to \mathtt{K}\colon k\,\langle \mathsf{bool}\rangle. \\ 3\ \mathtt{K} \to \mathtt{C}\colon c\,\langle \mathsf{bool}\rangle.\mathbf{t} \end{array}$$

Note the repetition of interactions is described by the (tail) recursion at the end. For an analysis of its conversation structure, we unfold the recursion once, obtaining:

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

Since the unfolding results in having a precise copy of the original type, channels are repeated in Lines 4/5/6 from Lines 1/2/3. We note:

- Lines 1 and 2 are temporally unordered in sending (representing concurrent sending actions from two participants). However 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 they do not induce conflict. Similarly for other pairs of the original and unfolded actions.

We can easily infer that essentially The same arguments hold in any n-fold unfoldings.

3.3 Safety Principle for Global Types

For a conversation in a session to proceed properly, it is desirable that there is no conflict (racing) at session channels. To ensure this, the necessary and sufficient condition is that, when a *common* channel is used in two communications, their sending actions and their receiving actions should respectively be ordered temporally,[7] so that no confusion arises at neither sending nor receiving. If a global type satisfies this principle, then it specifies a safe protocol: and can be used as a basis of guaranteeing safe process behaviours through type checking.

[7] To be precise, we demand that the pair of the concerned output prefixes become active in a strict temporary order; and that the same hold for the pair of the concerned prefixes.

Fig. 5 Causality Analysis

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

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

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

Definition 3.5 (dependency relations) Fix G. The relation $\prec_\phi$ over its prefixes is generated from:

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

An *input dependency from* n_1 *to* n_2 is a chain of the form $\mathsf{n}_1 \prec_{\phi_1} \cdots \prec_{\phi_n} \mathsf{n}_2$ ($n \geq 0$) such that $\phi_i \in \{\mathsf{II}, \mathsf{IO}\}$ for $1 \leq i \leq n-1$ and $\phi_n = \mathsf{II}$. An *output dependency from* n_1 *to* n_2 is a chain $\mathsf{n}_1 \prec_{\phi_1} \cdots \prec_{\phi_n} \mathsf{n}_2$ ($n \geq 1$) such that $\phi_i \in \{\mathsf{OO}, \mathsf{IO}\}$.

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

Definition 3.6 (linearity) G is *linear* if, whenever $\mathsf{n}_i = \mathtt{p_i} \to \mathtt{p}'_\mathtt{i} : k$ ($i = 1,2$) are in G for some k and do not occur in different branches of a branching, then both input and output dependencies exist from n_1 to n_2, or, if not, both exist from n_2 to n_1. If G carries other global types, we inductively demand the same.

Remark 3.7 We illustrate the condition on branching in Definition 3.6 by the following example.

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. $\quad \mathsf{quit} : \mathtt{C} \to \mathtt{D} : s.\mathsf{end}\ \}$		$\quad \mathtt{C} \to \mathtt{D} : s.\mathsf{end})$
(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.

Remark 3.8 Let G be linear and suppose n_1 and n_2 in G share a common channel and there is an input dependency and an output dependency in the same direction, say from n_1 to n_2. This does *not* mean the causal chains witnessing these dependencies are identical. As a simple example, consider

$$\mathtt{A} \to \mathtt{B} : t\langle \mathsf{nat} \rangle ; \mathtt{B} \to \mathtt{C} : s\langle \mathsf{nat} \rangle ; \mathtt{C} \to \mathtt{A} : t\langle \mathsf{nat} \rangle .\mathsf{end}$$

Here the input dependency from the first node to the last has no intermediate nodes while their output dependency should go through the second node.

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

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

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

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

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 3.9 *(1) A global type is linear iff its one-time unfolding is linear. (2) The linearity of a global type is decidable by a quadratic algorithm.*

Proof. 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).

Proof of (1)

Notation. (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 $G(n+1)\backslash G(n)$ to denote the newly added (unfolded) part of $G(n+1)$.

(iii) Given a node n in $G(m+1)\backslash G(m)$, we can jump back from n once to reach its "original" in $G(m)\backslash G(m-1)$ (which is $G(0)$ if $m=0$). This exact copy of n 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 $G(m-i+1)\backslash G(m-i)$ (which is $G(0)$ if $i=m+1$). Note there are $m+1$ such "foldings" of n in $G(m+1)\backslash G(m)$.

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). For (B) we use induction. If they are in $G(1)$ we are done (they cannot be in $G(0)$). Suppose this holds up to $G(m)$ ($m \geq 1$). If n is in $G(m+1)$ (hence n' is in $G(m)$) then take their foldings which are in $G(m)$ and $G(m-1)$. By induction they have dependencies. Then use (A). □

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

(iii) If n_1 is in $G(n-i)$ $(i \geq 1)$ and n_2 is in $G(n+1)\backslash G(n)$ then take the folding of n_2 say n_2' which is in $G(n)$. By induction we know n_1 and n_2' are conflict-free. Since the present recursive types only allow tail-recursion [8] the witnessing pairs of dependencies can only come from n_1 to n_2'. By Claim B there are both input/output dependencies from n_2' to n_2'. Thus we have, writing $\stackrel{\text{IO}}{\Longrightarrow}$ for the existence of both input/output dependencies:

$$\mathsf{n}_1 \quad \stackrel{\text{IO}}{\Longrightarrow} \quad \mathsf{n}_1' \quad \stackrel{\text{IO}}{\Longrightarrow} \quad \mathsf{n}_2'$$

Now we connect these chains and we are done. □

Proof of (2)

By (1) above, it suffices to (in)validate the linearity of $G(1)$. We consider the following algorithm.

1. First it checks dependencies by scanning through the graph structure of $G(1)$ once, annotating the graph with their dependencies as well as those nodes with common names (with indexes on branching):
2. Second it again traverses the annotated graph. When it encounters a new channel, it records so: and it continues to record for the continuity of IO-dependencies until one or both of them are cut. Call this state "off". If it meets a node of some name second time when its entry is in the "off" node then the algorithm fails. If not it continues.

If the whole graph is checked without failing then the algorithm succeeds. Since this algorithm demands, at each node, actions on at most m-entries where m is the number of all channels in the global type, which is at most the size of the graph, we conclude that this is quadratic on the size of the graph. □

[8] This tail recursiveness (which is automatically satisfied by all recursive types in the present type discipline) may not be essential for the argument, though details need be checked.

Fig. 6 Syntax of Local Types

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

4 Type Discipline for Multiparty Sessions

4.1 Programming Methodology for Multiparty Interactions

Once given global types as our tool, we can consider the following development steps for programs with multiparty sessions.

Step 1 A programmer describes her 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. The type discipline uses local types which represent local behaviour of processes and which act as a link between global types and processes.

4.2 Local Types

Syntax of Local Types *Local session types* or *local types*, ranged over by $T, T', ..$, are types for representing local behaviour of processes. The grammar of local types is given in Figure 6 (where the grammars for carried types U and S are repeated from Figure 4).

Local types are close to the foregoing session types in their shape: Indeed all constructs of local types come from the binary session types [17] except for the following changes to represent multiparty interactions.

Fig. 7 Type Isomorphism

$$k!\langle U\rangle;k'!\langle U'\rangle;T \approx k'!\langle U'\rangle;k!\langle U\rangle;T \qquad (k \neq k')$$

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

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

Type $k?\langle U\rangle;T$ represents the behaviour of receiving values of type U at s_k as messages (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.

For the rest, we have recursive types and type variables. As in global types, we demand type variables occur guarded by a prefix and take an equi-recursive approach for recursive types. Note local session type T does not contain parallel composition as in the original session types, making each participant single-threaded (taking off this restriction is easy, see Section 6.1).

$U, ...$ range over *value types*, which already appeared in Section 3, 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 denoted by a located type $T@p$ (indicating how these channels should be used in the receiving process). A sort can be either an atomic type or a global type: the former represents communication of constants while the latter represents communication of initiating names.

In addition to the standard isomorphism on recursive types induced by their regular tree representation, local types are considered up to the isomorphism induced by the rules in Figure 7 (closed under all type constructors). The equations permute two consecutive outputs with different subjects, capturing asynchrony in communication. Note the first equation can be regarded as the restricted case of the second one when I and J are singletons.

4.3 Projection and Coherence

To link global types and local types, we introduce the projection function of the former to the latter, extracting the part of a given global type which concerns the local behaviour of a specific 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:

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

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

The mapping is intuitive. For finiteness, we regard the map to act on the syntactic (or finite) presentation of global types (we can easily check that the projection is invariant under different syntactic presentation of global types up to $\equiv$). Some remarks are due:

- In the branching, all projections should generate an identical local type (otherwise undefined).
- In the parallel composition, $\mathtt{p}$ should be contained in at most a single type, ensuring each type is single-threaded (cf.§6.1).

Below $\mathsf{pid}(G)$ denotes the set of participant numbers occurring in G (but not in carried types). In (2) $T_{\mathtt{p}} \mathbin{@} \mathtt{p}$ 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) $\{T_{\mathtt{p}} \mathbin{@} \mathtt{p}\}_{\mathtt{p} \in I}$ 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 by a quadratic algorithm.*

Proof. By Theorem 3.6 (2) and because the tree equivalence is a quadratic problem. [31, §21:12]. □

We also believe the coherence of $\{T_{\mathtt{p}} \mathbin{@} \mathtt{p}\}_{\mathtt{p}}$ is decidable but this is not necessary for the present technical development.

4.4 Examples of Coherence

As an example of a linear global type which is not coherent, consider:

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

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

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

we can define the coherent projection as follows:

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

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

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

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

4.5 Typing System

Purpose of Typing The purpose of the typing system in the present section is to type behaviours *which are built by programmers*, i.e. without runtime elements such as queues and channel hiding. Formally:

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

Example 4.5 *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 the grammar of sortings and typings as follows. Below in "$\Gamma, u : S$", we assume u does not occur in Γ. Similarly in "$\Delta, \tilde{s} : \{T @ \mathtt{p}\}_{\mathtt{p} \in I}$", we assume *no* channel in $\tilde{s}$ occurs in the domain of Δ.

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

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to sorts and from process variables to sequences of sorts and types. *Typing* $(\Delta, \Delta', ..)$ records linear usage of session channels. In the binary sessions, it mapped each channel in its domain to a type: now it maps each vector of session channels in its domain to a family of located types. We write $\tilde{s} : T @ \mathtt{p}$ for a singleton typing $\tilde{s} : \{T @ \mathtt{p}\}$. 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 also write $\mathsf{sid}(G)$ for the set of session channel numbers in G.

Definition 4.6 A partial operator $\circ$ is defined as:

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

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

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

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

Typing System The type assignment system for processes is given in Figure 8. We use the judgement $\Gamma \vdash P \triangleright \Delta$ which reads: "under the environment Γ, process P has typing Δ". If we set $|\tilde{s}| = 1$ and $\mathtt{n} = 2$, and delete $\mathtt{p}$ from located type, the shape of rules is essentially identical with the original binary session typing [41]. Below we only illustrate the main rules.

[MCAST] is the rule for the session request. The type for $\tilde{s}$ is the *first* projection of the declared global type for a in Γ. [MACC] is for the session accept, taking the $\mathtt{p}$-th projection. The condition $|\tilde{s}| = \max(\mathsf{sid}(G))$ ensures the number of session channels meets those in G. The typing $\tilde{s} : T @ \mathtt{p}$ (which means $\tilde{s} : \{T @ \mathtt{p}\}$) ensures each prefix does not contain parallel threads which share $\tilde{s}$.

Fig. 8 Typing System for Expressions and Processes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[SEND] and [RCV] are the rules for sending and receiving values. Since the k-th name s_k of $\tilde{s}$ is used as the subject, we record the number k. In both rules, "p" in $T@\mathtt{p}$ ensures that P is (being inferred as) the behaviour for participant $\mathtt{p}$, and its domain should be $\tilde{s}$. Then the relevant type prefixes ($k!\langle\tilde{S}\rangle$ for the output and $k?\langle\tilde{S}\rangle$ for the input) are composed in the conclusion's session environment.

[DELEG] and [SREC] are the rules for delegation of a session and its dual. Delegation of a multiparty session passes the whole capability to participate in a multiparty session: thus operationally we send the whole vector of session channels. The carried type T' is located, making sure that the behaviour by the receiver at the passed channels takes the role of a specific participant (here $\mathtt{p}'$) in the delegated multiparty session. The rest follows the standard delegation rule [41], observing [DELEG] says that $\tilde{t}:T'@\mathtt{p}'$ does not appear in P symmetrically to [SREC] which uses the channels in P.

Rules [SEL] and [BRANCH] are identical with [41]. [CONC] uses $\asymp$ to ensure well-formedness of the session typing (note program phrases would practically lack any non-trivial composition at free channels, which is expected to happen only at runtime).

The remaining rules [IF], [INACT],[NRES], [VAR], and [DEF] are standard. In [INACT] and [VAR], "end only" means Δ only contains end as session types.

An *annotated* P is the result of annotating P's bound names as e.g. $(\nu a:\langle G\rangle)P$ and $s?(x:\langle G\rangle)P$. Assuming these annotations is natural from our design framework. For typing annotated processes we assume the obvious updates for [RCV] and [NRES] in Fig. 8.

Theorem 4.7 *Assume given an annotated program phrase P and Γ. Then it is decidable by a quadratic algorithm if there exists Δ such that $\Gamma\vdash P\triangleright\Delta$ or not. If such Δ exists there is an algorithm to construct one.*

NB: The typing constructed for a program phrase (if it exists) in the second clause of Theorem 4.7 is its principal typing, in the sense that other typings of the program phrase can be derived from it.

Proof. By the generation rules which follow the typing rules. The generation involves consistency check which in particular includes the subtyping judgement decidable by the standard quadratic algorithm. See Appendix A.1 for a concrete inference system. □

Corollary 4.8 (typing for programs) *Given an annotated program P and Γ, it is decidable if $\Gamma\vdash P\triangleright\emptyset$.*

NB: The typing of a program without spurious weakening is always empty.

Proof. Immediate since a program is always a program phrase. □

4.6 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 global type from § 3.2, that is:

$$
\begin{array}{rcl}
G & \stackrel{\text{def}}{=} & \begin{array}{l}
\mathtt{B1} \to \mathtt{S} : s\langle \mathsf{string} \rangle. \\
\mathtt{S} \to \mathtt{B1} : b_1\langle \mathsf{int} \rangle. \\
\mathtt{S} \to \mathtt{B2} : b_2\langle \mathsf{int} \rangle. \\
\mathtt{B1} \to \mathtt{B2} : b_2'\langle \mathsf{int} \rangle. \\
\mathtt{B2} \to \mathtt{S} : s\{\mathsf{ok} : \mathtt{B2} \to \mathtt{S} : s\,\langle \mathsf{string} \rangle.\mathtt{S} \to \mathtt{B2} :\, b_2\,\langle \mathsf{date} \rangle.G,\ \ \mathsf{quit} : \mathsf{end}\}
\end{array}
\end{array}
$$

where we set $\mathsf{Bi} = i$, $\mathsf{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$ and $s = 4$. Assuming for simplicity P_1, P_2, Q are $\mathbf{0}$, we obtain the following typing:

$$\Gamma \vdash Q_1 \triangleright \tilde{s} : s!\langle \mathsf{string} \rangle; b_1?\langle \mathsf{int} \rangle; b_2'!\langle \mathsf{int} \rangle \mathbin{@} \mathtt{B1}$$
$$\Gamma \vdash Q_2 \triangleright \tilde{s} : b_2?\langle \mathsf{int} \rangle; b_2'?\langle \mathsf{int} \rangle; b_1 \oplus \{\mathsf{ok} : s!\langle \mathsf{string} \rangle; b_2?\langle \mathsf{date} \rangle; T,\ \ \mathsf{quit} : \mathsf{end}\} \mathbin{@} \mathtt{B2}$$

Similarly for Seller. After prefixing at a, we can compose all three by [CONC]. Note these typings are calculable by Corollary 4.8.

Streaming Protocol Let $\Gamma' = \{a : \langle G' \rangle\}$ where G' is again from § 3.2. To wit:

$$G' \stackrel{\text{def}}{=} \mu \mathbf{t}. \begin{pmatrix} \mathtt{DP} \to \mathtt{K} :\, d\,\langle \mathsf{bool} \rangle. \\ \mathtt{KP} \to \mathtt{K} :\, k\,\langle \mathsf{bool} \rangle. \\ \mathtt{K} \to \mathtt{C} :\, c\,\langle \mathsf{bool} \rangle.\, \mathbf{t} \end{pmatrix}$$

where we set $d = 1$, $k = 2$, $c = 3$, $\mathtt{K} = 1$, $\mathtt{DP} = 2$, $\mathtt{C} = 3$ and $\mathtt{KP} = 4$. Now 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 of these processes as:

$$\begin{array}{rcl}
\Gamma' & \vdash & R_1 \triangleright dkc \ :\ \mu \mathbf{t}.d?\langle \mathsf{bool} \rangle; k?\langle \mathsf{bool} \rangle; c!\langle \mathsf{bool} \rangle; \mathbf{t} \mathbin{@} \mathtt{K} \\
\Gamma' & \vdash & R_2 \triangleright dkc \ :\ \mu \mathbf{t}.d!\langle \mathsf{bool} \rangle; \mathbf{t} \mathbin{@} \mathtt{DP} \\
\Gamma' & \vdash & R_4 \triangleright dkc \ :\ \mu \mathbf{t}.c?\langle \mathsf{bool} \rangle; \mathbf{t} \mathbin{@} \mathtt{C}
\end{array}$$

(R_4 is similar as R_2). Note these types correspond to the projection of G' onto respective participants: thus we have

$$\Gamma' \vdash \text{Kernel} \triangleright \emptyset.$$

Similarly DataProducer, Consumer and KeyProducer are typable under Γ' with the empty typings, which can be composed to make the well-typed initial configuration.

Delegation One of the expressiveness of the session types comes from a facility of *delegation* (often called *higher-order session passing*). We will type and see the relationship with global and local types for delegation. Consider the following three participants:

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

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

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

where the type $s_1!\langle \mathsf{int}\rangle @\, \mathtt{B}$ means the capability to send an integer from participant `B` 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 $(\text{Alice} \mid \text{Bob} \mid \text{Carol}) \longrightarrow\!\!\!\!\!\rightarrow (\nu\tilde{t}\tilde{s})(A \mid B \mid C \mid R)$ where A,B,C are the processes of Alice, Bob and Carol after their respective initial multicasting prefix 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 $P \equiv \mathbf{0}$:

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

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

5 Safety and Progress

This section establishes the fundamental behavioural properties of typed processes, communication safety and progress. For this purpose we follow the following three technical steps:

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

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

NB: 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 typing system in Section 4 which is for typing programs and program phrases, with the corresponding type inference algorithm.

5.1 How to Type a Queue

We first illustrate a key idea underlying our runtime typing using the following example.

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

We type the two processes excepting the middle queue with

$$s : 1!\langle \mathsf{nat}\rangle;1!\langle \mathsf{bool}\rangle;\mathsf{end}@\mathtt{p} \tag{4}$$

and

$$s : 1?\langle \mathsf{nat}\rangle;1?\langle \mathsf{bool}\rangle;\mathsf{end}@\mathtt{q}. \tag{5}$$

After a reduction, (3) changes into:

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

Note that (6) is identical with (3) except that an output prefix in (3) changes its place to the queue. Thus we can go back from (6) to (3) by placing this message on the top of the process. A key idea in our runtime typing is *to carry out this*

"rollback of a message" in typing, using a local type with a hole (a type context) for typing a queue. For example we type the queue in (6) as:

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

where $[\,]$ indicates a hole. Now we cover the type $1!\langle \mathsf{bool}\rangle;\mathsf{end}$ with the type context for $\mathtt{p}$ given above, $1!\langle \mathsf{nat}\rangle;[\ ]$, obtaining the type $1!\langle \mathsf{nat}\rangle;1!\langle \mathsf{bool}\rangle;\mathsf{end}$ for $\mathtt{p}$, 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

$$s_k \triangleleft ok; P \mid s_k : \emptyset \quad \longrightarrow \quad P \mid s_k : \mathsf{ok}. \tag{9}$$

Assume we type the left-hand side as

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

After the reduction, we obtain the type for P as

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

and the type for the queue as:

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

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

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

We now observe that the located type in (13) is a *subtype* of the located type in (10) in the standard session subtyping [2, 12], which is formally defined as:

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

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

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.

5.2 Type Contexts

We formally introduce the type contexts ($\mathcal{T}, \mathcal{T}', ...$) and the extended session typing ($\Delta, \Delta', ...$ as before) as follows.:

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

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 Figure 7 in §4.2 (page 20, applied to type contexts in the obvious way) together with the standard isomorphism on recursive types.[9] Each assignment in Δ may contain both local types and type contexts. We then set, writing $\mathsf{sid}(\mathcal{T})$ for the set of 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 cap a type (for a process) with a type context (for queues). In the second, we compose the type contexts for two mutually disjoint 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. Then the composition of typings, which extends Definition 4.6, is given as follows.

Definition 5.2 A partial operator $\circ$ is defined as:

$$\begin{array}{rcl} \{H_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p}\in I} \circ \{H'_{\mathtt{p}'} @ \mathtt{p}'\}_{\mathtt{p}'\in J} & = & \{H_{\mathtt{p}} @ \mathtt{p} \circ H'_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p}\in I\cap J} \cup \\ & & \{H_{\mathtt{p}} @ \mathtt{p}\}_{\mathtt{p}\in I\setminus J} \cup \\ & & \{H'_{\mathtt{p}'} @ \mathtt{p}'\}_{\mathtt{p}'\in J\setminus I} \end{array}$$

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:

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

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

[9] The subtyping does not make sense for type contexts since $\leq_{\mathsf{sub}}$ adds new branches.

Fig. 9 Selected Typing Rules for Runtime Processes

$$\Gamma \vdash s : \emptyset \triangleright_s \tilde{s} : \{[\,] @ \mathtt{p}\}_{\mathtt{p}} \qquad [\textsc{Qnil}]$$

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

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

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

$$\frac{\Delta \text{ end only} \quad \Delta' [\,] \text{ only}}{\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \Delta, \Delta'} \qquad [\textsc{Inact}]$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'} \qquad [\textsc{Subs}]$$

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

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

5.3 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. Some of the key typing rules for runtime are given in Figure 9 (the full typing rules are given in Appendix B).

[QNIL] starts from the empty hole for each participant, recording the session channel in the judgement. [QVAL] says that when we enqueue $\tilde{v}$, the type for $\tilde{v}$ is added at the tail. [QSESS] and [QSEL] are the corresponding rules for delegated channels and a label. [INACT] allows weakening for empty queue types while [SUBS] allows subsumption ($\leq_{\mathsf{sub}}$ is extended point-wise from $\leq_{\mathsf{sub}}$ on types).

[CONC] is refined to prohibit duplicated message queues. The rule does not use coherence (cf. Def.4.2 (2)) since coherence is meaningful only when all participants and queues are ready. In [CRES], since we are hiding session channels, we now know no other participants can be added. Hence we check all message queues are composed and the given configuration at $\tilde{s}$ is coherent.

For the rest, we refine the original typing rules in Figure 8 not appearing in Figure 9 as follows (the full typing rules are listed in Appendix B).

- For [MCAST],[MACC],[RCV],[SREC],[BRANCH] and [DEF], we replace $\Gamma \vdash P \triangleright \Delta$ with $\Gamma \vdash P \triangleright_{\emptyset} \Delta$.
- [VAR] is similar to [INACT] (so that a queue can never occur in processes realising participants).
- For both [DEF] and [NRES], we replace $\Gamma \vdash P \triangleright \Delta$ by $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$.

Using these typing rules, we can check that the redex (3) and its reductum (6) at the beginning of this section (page 28) 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 Section 4 iff $\Gamma \vdash P \triangleright_{\emptyset} \Delta$ is derived without using [*SUBS*] in the typing system in this section.*

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

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

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

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

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

Fig. 10 Reduction of Typings

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

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

$$\frac{H_1 \mathbin{@} \mathtt{p}_1, H_2 \mathbin{@} \mathtt{p}_2 \to H_1' \mathbin{@} \mathtt{p}_1, H_2' \mathbin{@} \mathtt{p}_2 \quad \mathtt{p}_1, \mathtt{p}_2 \in I}{\tilde{s}\colon \{\_,\ H_1 \mathbin{@} \mathtt{p}_1, H_2 \mathbin{@} \mathtt{p}_2,\ \_\}_{i\in I} \to \tilde{s}\colon \{\_,\ H_1' \mathbin{@} \mathtt{p}_1, H_2' \mathbin{@} \mathtt{p}_2,\ \_\}_{i\in I}} \qquad \text{[TR-Cxt-1]}$$

$$\frac{\Delta_0 \to \Delta_0'}{\Delta_0, \Delta_1 \to \Delta_0', \Delta_1} \qquad \text{[TR-Cxt-2]}$$

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

Proposition 5.4 *If $\Gamma \vdash P \triangleright_{s_1..s_m} \Delta$ then P has a unique queue at s_i ($1 \le i \le m$), no other queue at a free channel occurs in P, and no queue in P is under prefix.*

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

5.4 Type Reduction

Next we introduce reduction over session typings, which abstractly corresponds to interaction (including both sending and message delivery) in processes. The generation rules are given in Figure 10. In the rules we assume well-formedness of the types and typings. "_" in [TR-Cxt-1] stands for either an arbitrary located contexts or selections.

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

Definition 5.5 **(1)** *(full projection) Assume G is coherent and let $G \upharpoonright \mathtt{p}_i = T_i$ for each $\mathtt{p}_i \in \mathsf{pid}(G)$. Then $[\![G]\!]$, called* full projection of P, *denotes the family $\{T_i @ \mathtt{p}_i\}$.* **(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 5.6 *Each causal edge in G is preserved and reflected through the projection onto $[\![G]\!]$.*

Proof. This is because these causal edges record prefixing on the same participant preserved by projection. □

Definition 5.7 (merge set) *Assume G is coherent. Then we say two prefixes in G in different branches of a branching prefix are* mergeable *with each other when they are collapsed in its projection. A prefix is always mergeable with itself. Given a prefix* n*, its* merge set *is the set of prefixes mergeable with* n*.*

Proposition 5.8 *Two prefixes in G are mergeable iff they are related to one common input prefix and one common output prefix in* $[\![G]\!]$ *through projection.*

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

Corollary 5.9 *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 5.8, taking the map induced by the projection as the required bijection. □

Definition 5.10 *If a merge set of G is related to an input prefix and an output prefix in* $[\![G]\!]$ *by the bijection noted in Corollary 5.9 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 local types.

Proposition 5.11

(1) *If a pair of prefixes in* $[\![G]\!]$ *(taken up to the type isomorphism* $\approx$*) form a redex with respect to* $\rightarrow$ *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 A.3. □

Remark 5.12 Proposition 5.11 (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/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 5.13 (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 5.14 (1) $\Delta_1 \to \Delta_1'$ *and* $\Delta_1 \asymp \Delta_2$ *imply* $\Delta_1' \asymp \Delta_2$ *and* $\Delta_1 \circ \Delta_2 \to \Delta_1' \circ \Delta_2$. (2) *Let* Δ *be coherent. Then* $\Delta \to \Delta'$ *implies* Δ' *is coherent.* (3) *Let* Δ *be partially coherent. Then* $\Delta \to \Delta'$ *implies* Δ' *is partially coherent.*

Proof. For (1) suppose $\Delta_1 \to \Delta_1'$ and $\Delta_1 \asymp \Delta_2$. Note $\Delta_1 \asymp \Delta_2$ means that each pair of vectors of channels from $\Delta_{1,2}$ either coincide or are disjoint, and that, if they coincide, their image are participant-wise composable by $\circ$. Since no rule for $\to$ in Figure 10 invalidate either condition we conclude $\Delta_1' \asymp \Delta_2$.

For (2), suppose Δ is coherent and $\Delta \to \Delta'$. Suppose the associated redex is in $\Delta(\tilde{s})$. By coherence we can write $\Delta(\tilde{s})$ as $[\![G]\!]$ for some coherent G. Now consider the preimage of the associated redex in $[\![G]\!]$, whose existence is guaranteed by Proposition 5.11 (3). This preimage is not suppressed by causal edges by Proposition 5.11 (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 5.11 (2) we are done.

Finally (3) is immediate from (1) and (2). □

5.5 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) $\mathcal{T}_1 @ p \in \Delta_1(\tilde{s})$ and $\mathcal{T}_2 @ p \in \Delta_2(\tilde{s})$; 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).[10]

Lemma 5.15 (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 A.4. □

[10] 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 *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 5.17 (permutation) (1) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [*SUBS*] at its last two steps. Then $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ has a derivation identical with the original one except its last two steps are replaced by a single application of [*SUBS*].* (2) *Assume given a derivation of $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ which uses [*SUBS*] as its last rule and another rule which is not one of [*SUBS*], [*SEL*] and [*BRANCH*]. 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 [SUBS]. (2) is mechanical. □

Lemma 5.18 (queue) *The following rules are admissible in the typing system for runtime processes. Below let $\tilde{s} = s_1..s_k..s_n$ and assume in each rule $\circ$ in the premise is well-defined.*

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

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

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

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

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

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

Proof. See Appendix A.5. □

Below we do not require the substitution lemmas for session channels and process variables, cf. [41].

Lemma 5.19 (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 [41]. □

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.

Theorem 5.20 (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) $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ *with* Δ *coherent and* $P \longrightarrow P'$ *imply* $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ *where* $\Delta = \Delta'$ *or* $\Delta \longrightarrow \Delta'$.
(3) $\Gamma \vdash P \triangleright_{\emptyset} \emptyset$ *and* $P \longrightarrow P'$ *imply* $\Gamma \vdash P' \triangleright_{\emptyset} \emptyset$.

Proof. (1) is by rule induction on $\equiv$ showing, in both ways, that if one side has a typing then the other side has the same typing. In the following we safely ignore uninteresting (permutable) final applications of [SUBS] in derivations by way of Lemma 5.17.

Case $P \mid \mathbf{0} \equiv P$: First assume $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$. By $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \emptyset$ and by applying [CONC] to these two sequents we immediately obtain $\Gamma \vdash P|\mathbf{0} \triangleright \Delta$, as required. For the converse direction assume $\Gamma \vdash P|\mathbf{0} \triangleright_{\tilde{s}} \Delta$. We can safely assume (via Lemma 5.17) that the last rule applied is [CONC]. Thus we can set $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta_1$ and $\Gamma \vdash \mathbf{0} \triangleright_{\emptyset} \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. Note we can safely regard $\Gamma \vdash \mathbf{0} \triangleright \Delta_2$ as being inferred by the axiom [INACT] since applying [SUBS] to empty types and empty type contexts again 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 5.19 (2) we are done.

Case $P \mid Q \equiv Q \mid P$: By symmetry of the rule we have only to show one direction. Suppose $\Gamma \vdash P|Q \triangleright_{\tilde{s}} \Delta$. We can safely assume the last rule applied is [CONC]. We can thus set $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$ and $\Gamma \vdash Q \triangleright_{\tilde{r}} \Delta_2$ such that $\Delta_1 \asymp \Delta_2$, $\Delta_1 \circ \Delta_2 = \Delta$ and $\tilde{t} \uplus \tilde{r} = \tilde{s}$. By Lemma 5.15 we know $\Delta_2 \asymp \Delta_1$ and $\Delta_2 \circ \Delta_1 = \Delta$ hence by applying [CONC] with the premises reversed we are done.

Case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$: By the establishment of the previous case again we have only to show one direction. Suppose $\Gamma \vdash (P \mid Q) \mid R \triangleright_{\tilde{s}} \Delta$. We can safely assume: $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta_1$, $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta_2$ and $\Gamma \vdash P \triangleright_{\tilde{q}} \Delta_3$ such that $\Delta_1 \asymp \Delta_2$, $(\Delta_1 \circ \Delta_2) \asymp \Delta_3$ and $(\Delta_1 \circ \Delta_2) \circ \Delta_3 = \Delta$, as well as $\tilde{t} \uplus \tilde{r} \uplus \tilde{q} = \tilde{s}$. By the last condition, no two of Δ_1, Δ_2 and Δ_3 share a common target channel in their type contexts (in the sense given just before Lemma 5.15, page 35) 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 5.15 to know $\Delta_2 \asymp \Delta_3$, $\Delta_1 \asymp (\Delta_2 \circ \Delta_3)$ and $\Delta_1 \circ (\Delta_2 \circ \Delta_3) = \Delta$. By applying [CONC] in an appropriate order we are done.

The remaining rules are reasoned exactly as in [41] (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. 5.13). 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 5.14 (1). Further by Proposition 5.14 (2), Δ' above is again partially coherent. Below we again ignore irrelevant final application of [SUBS] through Lemma 5.17. All rule names are those of the typing rules (as given in Appendix B).

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

$$\Gamma \vdash R \triangleright \Delta \tag{14}$$

By Lemma 5.16 we know $a \in \mathsf{dom}(\Gamma)$. Let $\Gamma(a) = G$. Since (14) can only be inferred by the sequence of [CONC] (up to permutable [SUBS], similarly in the following), we know $\Gamma \vdash R_i \triangleright \Delta_i$ $(1 \leq i \leq n)$ such that $\Delta_1 \circ \ldots \circ \Delta_n = \Delta$. By [MCAST] and [MACC] this means:

$$\Gamma \vdash P_i \triangleright \Delta_i, \tilde{s} : \{(G \upharpoonright i)@i\} \tag{15}$$

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

$$\Gamma \vdash (\Pi_i P_i) \mid (\Pi_i s_i : \emptyset) \triangleright_{\tilde{s}} \Delta, \tilde{s} : \{(G \upharpoonright i)@i\}_{1 \leq i \leq n} \tag{16}$$

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

$$\Gamma \vdash (\nu \tilde{s})(\ (\pi_i P_i) \mid (\Pi_i s_i : \emptyset)\ ) \triangleright \Delta \tag{17}$$

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

Case [SEND]: We use the first rule of Lemma 5.18 for "rolling back" a message. Suppose we have:

$$\Gamma \vdash s!\langle \tilde{e} \rangle; P \mid s : \tilde{h} \triangleright_s \Delta \tag{18}$$

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

$$\Gamma \vdash s!\langle \tilde{e} \rangle; P \triangleright_{\emptyset} \Delta_1 \tag{19}$$

and

$$\Gamma \vdash s : \tilde{h} \triangleright_s \Delta_2 \tag{20}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (19) can only be inferred from [SEND] we know, first:

$$\Gamma \vdash e_j : S_j \tag{21}$$

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

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s} : k! \langle \tilde{S} \rangle . T \,@\, \mathtt{p} \tag{22}$$

and moreover

$$\Gamma \vdash P \triangleright_{\emptyset} \Delta_1' \circ \tilde{s} : T \,@\, \mathtt{p}. \tag{23}$$

On the other hand by $\Delta_1 \asymp \Delta_2$ and (20) we know:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[\,] \,@\, \mathtt{p} \tag{24}$$

Now assume $\tilde{e} \downarrow \tilde{v}$. Notice by (21) we have $\Gamma \vdash v_j : S_j$ for each v_j in $\tilde{v}$. Thus by Lemma 5.18, [QVAL], we infer:

$$\Gamma \vdash s : \tilde{h} \cdot \tilde{v} \triangleright \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[k! \langle \tilde{S} \rangle . [\,]] \,@\, \mathtt{p}. \tag{25}$$

By the algebra of located types and type contexts:

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

Thus by applying [CONC] to (19) and (20) we obtain:

$$\Gamma \vdash P \mid s : \tilde{h} \cdot \tilde{v} \triangleright \Delta \tag{26}$$

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

Case [DELEG]: Similar to [SEND] using the second rule of Lemma 5.18, see Appendix A.6.

Case [LABEL]: We use the third rule of Lemma 5.18 together with the subtyping $\leq_{\mathsf{sub}}$. Suppose we have:

$$\Gamma \vdash s \lhd l; P \mid s{:}\tilde{h} \rhd_s \Delta \tag{27}$$

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

$$\Gamma \vdash s \lhd l; P \rhd_{\emptyset} \Delta_1 \tag{28}$$

and

$$\Gamma \vdash s{:}\tilde{h} \rhd_s \Delta_2 \tag{29}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (28) can only be inferred from [SEL] as the last rule (up to permutable applications of [SUBS]), we know, for some p and for some $\tilde{s}$ which includes s and for some $\{l_i\}$ which includes l,

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s} : k \oplus \{l_i\colon T_i\}_{i\in I}.T \,@\, \mathtt{p} \tag{30}$$

and moreover

$$\Gamma \vdash P \rhd_{\emptyset} \Delta_1' \circ \tilde{s}\colon T \,@\, \mathtt{p}. \tag{31}$$

On the other hand we can write:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[\,] \,@\, \mathtt{p} \tag{32}$$

By (29), (32) and Lemma 5.18, [QSEL], we infer:

$$\Gamma \vdash s{:}\tilde{h} \cdot \tilde{v} \rhd \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[k \oplus l.[\,]] \,@\, \mathtt{p}. \tag{33}$$

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

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

Thus we obtain, by applying [CONC] to (31), (33) then applying [SUBS] (the subsumption rule):

$$\Gamma \vdash P \mid s{:}\tilde{h} \cdot l \rhd \Delta \tag{34}$$

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

Case [RECV]: By the first of the latter three rules of Lemma 5.18 together with Lemma 5.19. Suppose

$$\Gamma \vdash s?(\tilde{x});P \mid s:\tilde{v}\cdot\tilde{h} \triangleright_s \Delta \tag{35}$$

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

$$\Gamma \vdash s?(\tilde{x});P \triangleright_\emptyset \Delta_1 \tag{36}$$

and

$$\Gamma \vdash s:\tilde{v}\cdot\tilde{h} \triangleright_s \Delta_2 \tag{37}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (36) can only be inferred from [RCV] we know, for some p and for some $\tilde{s}$ which includes s,

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s}:k?\langle\tilde{S}\rangle.T@\mathtt{p} \tag{38}$$

and moreover

$$\Gamma,\tilde{x}:\tilde{S} \vdash P \triangleright_\emptyset \Delta_1' \circ \tilde{s}:T@\mathtt{p}. \tag{39}$$

By Lemma 5.19, we obtain:

$$\Gamma \vdash P[\tilde{v}/\tilde{x}] \triangleright_\emptyset \Delta_1' \circ \tilde{s}:T@\mathtt{p}. \tag{40}$$

Further by $\Delta_1 \asymp \Delta_2$ and (37) we know:

$$\Delta_2 \;=\; \Delta_2' \circ \tilde{s}:k!\langle\tilde{S}\rangle.\mathcal{T}@\mathtt{p} \tag{41}$$

By Lemma 5.18, [QVALDQ], we infer:

$$\Gamma \vdash s:\tilde{h} \triangleright \Delta_2' \circ \tilde{s}:\mathcal{T}@\mathtt{p}. \tag{42}$$

Using Proposition 5.14 we obtain:

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

Thus by applying [CONC] to (36) and (37) we obtain:

$$\Gamma \vdash P[\tilde{v}/\tilde{x}] \mid s:\tilde{h} \triangleright \Delta' \tag{43}$$

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

Case [SREC], [BRANCH]: Similar to [RECV], using the latter two rules of Lemma 5.18, see Appendix A.6.

Case [IFT], [IFF], [DEF], [DEFIN]: Standard, cf. [41]. No difference in the typing.

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

$$\Gamma \vdash (\nu a)P \triangleright \Delta \tag{44}$$

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

$$\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta. \tag{45}$$

By induction hypothesis we know

$$\Gamma, a : \langle G \rangle \vdash P' \triangleright \Delta' \tag{46}$$

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

$$\Gamma \vdash P' \triangleright \Delta' \tag{47}$$

as required. When session channels are hidden, suppose

$$\Gamma \vdash (\nu \tilde{s})P \triangleright \Delta \tag{48}$$

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

$$\Gamma \vdash P \triangleright \Delta, \{s\} : \{T_p @ p\}_{p \in I} \tag{49}$$

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

$$\Gamma \vdash P' \triangleright \Delta', \{s\} : \{T'_p @ p\}_{p \in I} \tag{50}$$

where either $\Delta \to^{0,1} \Delta'$ or $\{s\} : \{T_p @ p\}_{p \in I} \to^{0,1} \{s\} : \{T'_p @ p\}_{p \in I}$. By Lemma 5.14 (1) $\{T'_p @ p\}_{p \in I}$ is again coherent. Hence by [CRES] we obtain

$$\Gamma \vdash (\nu \tilde{s})P' \triangleright \Delta' \tag{51}$$

as required.

Case [PAR]: Suppose we have $\Gamma \vdash P|Q \triangleright \Delta$ and $P \to P'$. By [CONC] we have $\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta_2$ such that $\Delta_1 \circ \Delta_2 = \Delta$. By induction hypothesis we have $\Gamma \vdash P' \triangleright \Delta'_1$ such that $\Delta_1 \to^{0,1} \Delta'_1$. By Proposition 5.14 (1) we have

$\Delta_1' \asymp \Delta_2$ hence $\Gamma \vdash P'|Q \triangleright \Delta_1' \circ \Delta_2$. Noting Proposition 5.14 (1) also says that $(\Delta_1 \circ \Delta_2) \rightarrow^{0,1} (\Delta_1' \circ \Delta_2)$ we are done.

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

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

Remark 5.21 Theorem 5.20 (3) and the subsequent results (in particular Theorems 5.23 and 5.31 below) tell us, through Proposition 5.3, that the typing system in Section 4, which is for programs and program phrases, guarantees type safety and other significant behavioural properties for typable programs. Further this typability of programs is effective by Corollary 4.8.

Theorem 5.20 immediately entails the lack of the standard type errors in expressions (such as $\mathsf{true} + 3$). The type discipline also satisfies, as in the preceding session type disciplines (cf. [17, 36, 41]), communication error freedom, including linear usage of channels. We first introduce the reduction context $\mathcal{E}$ as follows:

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

Henceforth we assume the standard bound name convention. 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 an prefix or an `if` branch, after any unfolding by [DEF]. We write $P\langle\!\langle s \rangle\!\rangle$ if P contains an active subject at s after applying [DEF], and $P\langle\!\langle s! \rangle\!\rangle$ (resp. $P\langle\!\langle s? \rangle\!\rangle$) if P contains an emitting (resp. receiving) active prefix at s.
- We say P has a *redex at s* if its redex pair (the process part) is a prefix at s.

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

Lemma 5.22 *Assume* $\Gamma \vdash P \triangleright_\emptyset \Delta$ *s.t.* $\Delta \circ \Delta_0$ *is partially coherent for some* Δ_0. (1) *If* $P\langle\!\langle s \rangle\!\rangle$ *then P contains either a unique active prefix at s or a unique active emitting prefix and a unique active receiving prefix at s.* (2) *If P contains an active emitting (resp. receiving) prefix at s then* Δ *contains an emitting (resp. receiving) minimal prefix at s.*

Proof. By easy rule induction, see Appendix A.7. □

We can now establish the communication safety for typable processes. The following statement decomposes the corresponding properties known for synchronous session types [17, 36, 41] into the sending side and the receiving side, due to the use of queues in the present asynchronous semantics. In the following we again work under the standard bound name convention.

Theorem 5.23 (Communication Safety) *Suppose* $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ *s.t.* Δ *is coherent and* P *has a redex at free* s*. Then:*

1. *(linearity)* $P \equiv \mathcal{E}[s : \tilde{h}]$ *such that either*
   (a) $P \langle\!\langle s? \rangle\!\rangle$ *and* s *occurs exactly once in* $\mathcal{E}$ *with* $\tilde{h} \neq \emptyset$*; or*
   (b) $P \langle\!\langle s! \rangle\!\rangle$ *and* s *occurs exactly once in* $\mathcal{E}$*; or*
   (c) $P \langle\!\langle s? \rangle\!\rangle$ *and* $P \langle\!\langle s! \rangle\!\rangle$.
2. *(error-freedom) if* $P \equiv \mathcal{E}[R]$ *with* $R \langle\!\langle s? \rangle\!\rangle$ *being a redex:*
   (a) *If* $R \equiv s?(\tilde{y}); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{v} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{v}| = |\tilde{x}|$.
   (b) *If* $R \equiv s?(\!(\tilde{s})\!); Q$ *then* $P \equiv \mathcal{E}'[s : \tilde{t} \cdot \tilde{h}]$ *for some* $\mathcal{E}'$ *and* $|\tilde{s}| = |\tilde{t}|$*; and*
   (c) *If* $R \equiv s \rhd \{l_i : Q_i\}_{i \in I}$ *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 5.22 we know P_0 has either a single active prefix or a pair of a receiving active prefix and an emitting active prefix. So we have three cases:

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

Hence as required.

For (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 5.20 and 5.23, 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 5.22 (2) and by the typing of the associated queue, this delivery precisely corresponds to a redex in the session typing.

5.6 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 [8, 17]. First we define:

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

A process is queue-full when it has a queue for each session channel. Observe:

Proposition 5.25 *Let P be typable and queue-full. Then for each free or ν-bound session channel, there is a unique queue with that channel. Further $P \rightarrow^* P'$ implies P' is also typable and queue-full.*

Proof. The first part is by the typing rule [CRES]. The second part is by Theorem 5.20 and shape of the reduction rules involving queues including [LINK].□

The following condition 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 5.26 (simple) A process P is *simple* when it is typable with a type derivation where the session typing of both the premise and conclusion of each prefix rule in Figure 8 is restricted to at most a singleton.

Proposition 5.27 *Let P_0 be simple and $P_0 \rightarrow^* P$. 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 A.8. □

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

Definition 5.28 (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 following gives the converse of Lemma 5.22 (2).

Lemma 5.29 *Let $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ and P is simple. If there is an active receiving (resp. active emitting) prefix in Δ at s and none of prefixes at s in P is under a prefix at a shared name or under an* `if`*-branch, then $P \langle\!\langle s? \rangle\!\rangle$ (resp. either $P \langle\!\langle s! \rangle\!\rangle$ or the queue at s is not empty).*

Proof. By rule induction using Proposition 5.27, see Appendix A.9. □

Below we say *P is empty* when it is the parallel composition of zero or more empty queues possibly under restrictions ($P \equiv \mathbf{0}$ when the number is zero).

Proposition 5.30 *Let $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$, Δ is coherent, P is simple, queue-full, well-linked and non-empty. Then $P \to P'$ for some P'.*

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 5.27, it should form a redex (together with another prefix to form the image of a merge set). By Lemma 5.29 and Theorem 5.23 (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 5.29 there are the following cases:

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

In each case there is a reduction hence done. □

Theorem 5.31 (progress) Let P be a simple and well-linked program. Then P has the *progress property* in the sense that $P \to^* P'$ implies either P' is empty or $P' \to P''$ for some P''.

Proof. Immediate from Proposition 5.27, Lemma 5.29 and Theorem 5.30. □

A simple application of Theorems 5.20 (3), 5.23 and 5.31 for processes from §2.3 follow. Below *communication mismatch* stands for the violation of the conditions given in Theorem 5.23 (2).

Proposition 5.32 (properties of two protocols)

(1) Let $Buyer1|Buyer2|Seller \rightarrow^* P$. *Then* P *is well-typed,* P *has no communication mismatch, and if* P *is not empty then* $P \rightarrow P'$ *for some* P'.
(2) Similarly for $DataProducer|KeyProducer|Kernel|Consumer$.

Proof. Immediate from Theorem 5.31 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). □

Remark 5.33 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

6.1 Extensions

We discuss several direct extensions of the proposed theory.

Existing Extensions of Session Types In the literature, several extensions of binary session type disciplines have been proposed, including subtyping [12] (which can be used in typing for program phrases unlike the system in Section 4, bounded polymorphism [11], integration with security annotations to guarantee authentication properties [1], and integration with higher-order π-calculus [27]. 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 local 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. This extension is also effective with the generalised global types discussed next.

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 description cannot be represented in the current global types since two "prefixes" crisscross each other. Interestingly, our linearity conditions in § 3.3, based on input/output dependencies, can directly capture the safety of this configuration. All we need to do is to take the graphs of prefixes and II, IO and OO-edges (cf. Figure 5) under the linearity condition (precisely following §3.3) as global types, augmented with an acyclicity condition on chains of these causal edges. All other definitions and results stay the same.

Synchrony and Asynchrony Most of the current session types are binary and synchronous [17]. 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). We also augment the well-formedness condition based on "connectedness" in the sense of [2].

A different direction is to consider asynchronous message passing without order-preservation [16] 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 $s!\langle V\rangle;t!\langle W\rangle$ who 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 present type structures: we extend $\mathtt{p} \to \mathtt{p}' : k$ in the prefix of global types to $\mathtt{p} \to \mathtt{p}_1,..,\mathtt{p}_n : \{k_1,...,k_n\}$, representing the multicast of a message to $\mathtt{p}_1,..,\mathtt{p}_n$ via channels $k_1,...,k_n$ by participant $\mathtt{p}$, similarly we extend local session types to $\tilde{k}!\langle U\rangle$ from $k!\langle U\rangle$. Branching can similarly be extended. We may also incorporate a practical adaptation such as group addressing. Causality analysis remains the same by decomposing each multicasting prefix into its unicasting elements and considering causality for each of them. In type inference, a multicasting process prefix is typed by a a multicasting type prefix.

6.2 Related Work

There is a large literature on session types for both process calculi (in particular π-calculi) and programming languages.

Asynchronous Session Types Our multiparty session types are based on message-order preserving asynchronous communication. The operational semantics of binary sessions based on asynchronous communication was first considered by Neubauer and Thiemann [30]. Recently Gay and Vasconcelos [13] study the asynchronous version of binary sessions for an ML-like language based on their preceding work [38]. In [13], a message queue is given two endpoint channels and a direction.

Coppo, Dezani-Ciancaglini and Yoshida study the asynchronous binary session types for Java in [5], extending the previous work in [8], 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 with possibly interleaving multiple sessions, significantly enlarging the framework in [5].

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 [2] and by Corin et al. [6]. Our work [2] presented an *executable language* for directly describing Web interactions from a global viewpoint and provided the framework for projecting a description in the language to local processes. The use of global description for *types* and its associated theories have not been developed in [2]. The type disciplines for the two (global and local) calculi studied in [2] are based on binary synchronous session types, hence safety and progress for multiparty interactions are not considered. Thus these two theories are in fact orthogonal.

Corin et al. [6] 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. A verification method for session integrity – that a concrete instance of a session is carried out properly following the graph – is introduced as a run-time analysis on tokens carried in messages at each communication, for which the soundness is proved.

The purpose of the message flow graphs in [6] 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 [17] 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 [12, 41] allows substitution of session channels as in [RECV], which also satisfies type safety and progress through annotations on channels and types. With this change the whole theo-

ries remain intact with exactly the same operational semantics and typing for programs.

Linear and Behavioural Types for Mobile Processes The session type disciplines are related with linear/IO-typed π-calculi with causality information. The causality analysis in global types in the present work is partly inspired by the graph-based linear types developed in [40] where ordering among multiple linear names (which correspond to session channels) establishes deadlock-freedom of typed processes. Kobayashi and his colleagues, e.g [20, 21] study generalised forms of linear typing for guaranteeing different kinds of deadlock-freedom, incorporating synchronisations and locking. A combination of linear channel types and CCS-like processes is used as types, for describing fine-grained channel usage and dependency. A detailed type inference system for local processes is also studied in [22].

A main difference of session type disciplines from these and other preceding works on types for processes is a notion of rigorously structured conversations and their direct type abstraction. By raising the level of abstraction through the use of structured primitives such as separate session initiation, branching and recursion, session types can describe complex interaction structures more intelligibly and enable efficient type checking. These features would have direct applicability for the design of programming languages with communication. This paper shows that this framework smoothly extends to multiparty interactions guaranteeing strong behavioural properties such as linearity and progress with simple proofs while allowing an efficient type inference.

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 syntaxes 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 [32]. Recent works focus on behavioural equivalences with lts [24, 25, 33] and static analysis [28] 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 types. Recent advanced location-based typing systems for distributed processes [15]

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

7 Conclusion

One of the main open problems of the session types repeatedly posed by researchers and industry partners [39] 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 both questions 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 theory enables a new programming methodology for communication-based applications.

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 for a clean and modular causal analysis systematically applicable to both synchronous and asynchronous communications, offering a basis of behavioural properties of typed processes.

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. Another topic is the use of this theory as a basis of communication-centred extensions of general purpose programming languages (cf. [19]). Other open questions 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. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *JFP*, 15(2):219–248, 2005.
2. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
3. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. To be published by W3C. Available at `www.dcs.qmul.ac.uk/~carbonem/cdlpaper`.
4. P. Collingbourne. Verification tools for multi-core programming. Master's thesis, Imperial College London, 2007.
5. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.
6. R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CFS'07*. IEEE-CS Press, 2007.
7. M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded Session Types for Object-Oriented Languages. In *FMCO'06*, LNCS. Springer-Verlag, 2007.
8. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
9. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
10. P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In M. Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.
11. S. Gay. Bounded polymorphism in session types. *MSCS*. To appear.
12. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
13. S. Gay and V. T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, may 2007.
14. J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
15. M. Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
16. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, 1991.
17. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
18. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007.
19. R. Hu, N. Yoshida, and K. Honda. Type-safe Communication in Java with Session Types. `http://www.doc.ic.ac.uk/~rh105/sessiondj.html`, March 2007.
20. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
21. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.
22. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR'00*, volume 1877 of *LNCS*, pages 489–503, 2000.

23. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
24. M. Merro. An observational theory for mobile ad hoc networks. In *Electronic Notes in Theoretical Computer Science*, volume 172, pages 275–293. Elsevier, 2007.
25. N. Mezzetti and D. Sangiorgi. Towards a calculus for wireless systems. In *Electronic Notes in Theoretical Computer Science*, volume 158, pages 331–353. Elsevier, 2006.
26. S. Microsystems Inc. The Java Tutorial: All About Sockets. http://java.sun.com/docs/books/tutorial/networking/sockets/.
27. D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, LNCS. Springer-Verlag, 2007.
28. S. Nanz, F. Nielson, and H. R. Nielson. Topology-dependent abstractions of broadcast network. In *CONCUR*, LNCS, 2007.
29. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
30. M. Neubauer and P. Thiemann. Session Types for Asynchronous Communication. Universität Freiburg, 2004.
31. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
32. K. Prasad. Broadcast calculus interpreted in ccs upto bisimulation. In *Electronic Notes in Theoretical Computer Science*, volume 52, pages 83–100. Elsevier, 2001.
33. K. Prasad. A prospectus for mobile broadcasting systems. In *Electronic Notes in Theoretical Computer Science*, volume 162, pages 295–300. Elsevier, 2006.
34. B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.
35. S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), 2006.
36. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
37. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.
38. V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.
39. Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.
40. N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCSe*, pages 371–386, 1996.
41. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisit. In *SecRet'06*, ENTCS. Elsevier, 2007.

A Proofs

A.1 Proof of Theorem 4.7

In the following we use, without loss of generality, the standard (annotated) single recursion $\mu X(\tilde{x}:\tilde{S},\tilde{t}_1,..,\tilde{t}_n).P$, with the typing rule:

$$\frac{\Gamma, X:\tilde{S}\tilde{T}, \tilde{x}:\tilde{S} \vdash P \triangleright \tilde{s}_1 : T_1 @ \mathtt{p}_1 .. \tilde{s_n} : T_n @ \mathtt{p}_n}{\Gamma \vdash \mu X(\tilde{x}^{\tilde{S}}, \tilde{s}_1, .., \tilde{s}_n).P \triangleright \Delta}$$

which is simpler for our discussion.

Recall that an *annotated program phrase* will have all annotations on the binding occurrences of shared names (which occur in the accept prefixes, the name hidings, and the recursions as given above). Assume given Γ and annotated program phrase P. First we annotate all free and bound occurrences of names in P, using the sorts from Γ and from the binders.

The following rules, regarded as generation rules from the premise to the conclusion, construct the typing of P under Γ following the syntactic structure of P, that is it gives the typing rules for typable processes. If the production does not succeed at some point the algorithm fails. We use the same sequent $\Gamma \vdash P \triangleright \Delta$ even though Γ is no longer necessary.

First, the constants and names are immediate (the latter from annotated types). For the expression the same rule as the typing rule:

$$\frac{\Gamma \vdash e_i \triangleright \mathsf{bool}}{\Gamma \vdash e_1 \mathsf{or}\ e_2 : \mathsf{bool}} \qquad [\textsc{Or}]$$

is enough.

For the remaining rules which construct processes, we need participant numbers. These are given so that (when introduced) each is fresh: then as needed we carry out unification. The following are the generation rules for session request/accept.

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

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

Above $T \ll T'$ stands for unifiability (which includes filling additional branches in selection types which is essentially part of the standard session subtyping [2, 12]). The unification is on participant numbers and type variables for delegation (see below). The use of session subtyping is because the expected selection type may have additional branches in comparison with the real branches the process will use. This subtyping is decidable by the standard quadratic algorithm.

We next give the rules for channel (session) prefixes.

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

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

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p}}{\Gamma \vdash s_k!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright \Delta, \tilde{s}\colon k!\langle \mathbf{T} \mathbin{@} \mathtt{p}' \rangle.T \mathbin{@} \mathtt{p},\ \tilde{t}\colon \mathbf{T} \mathbin{@} \mathtt{p}'} \quad [\textsc{Deleg}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p},\ \tilde{t}\colon T' \mathbin{@} \mathtt{p}'}{\Gamma \vdash s_k?(\!(\tilde{t})\!); P \triangleright \Delta, \tilde{s}\colon k?\langle T' \mathbin{@} \mathtt{p}' \rangle.T \mathbin{@} \mathtt{p}} \quad [\textsc{SRec}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p}}{\Gamma \vdash s_k \lhd l; P \triangleright \Delta, \tilde{s} : k \oplus l.T \mathbin{@} \mathtt{p}} \quad [\textsc{Sel}]$$

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

In these generation rules for prefixes, it is possible that the premise does not have a "base" typing (for example $\tilde{s}\colon T \mathbin{@} \mathtt{p}$ in [Send]). In this case we stipulate that we lazily add $\tilde{s}\colon \mathsf{end}$. In [Rcv], we demand that the sorts in the premise and the sorts annotating binders should coincide. In [Deleg] we use the type variable to be unified later (specifically written with the capital letter to make it distinct from type variables for recursion though they cannot be confused). In [Deleg] we check disjointness of carried channels and those channels occurring in P.

For the remaining construction rules except for recursion we have:

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

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta_1 \qquad \Gamma \vdash Q \triangleright \Delta_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta_1 \sqcup \Delta_2} \quad [\textsc{If}]$$

$$\Gamma \vdash \mathbf{0} \triangleright \emptyset \qquad \frac{\Gamma, a\colon \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu \langle G \rangle) P \triangleright \Delta} \quad [\textsc{Inact}],[\textsc{NRes}]$$

In [Conc] we compose two processes disjointly (note $\asymp$ is decidable). [If] is different where we need to unify two selection types which may have different selection branches. [Inact] introduces the empty typings (as noted end is introduced lazily as needed).

Finally for process variable and recursion we use the following rules:

$$\frac{\Gamma \vdash \tilde{e} : \tilde{S}}{\Gamma, X : \tilde{S}\mathbf{t}^1, .., \mathbf{t}^n \vdash X\langle \tilde{e}\tilde{s}_1..\tilde{s}_n\rangle \triangleright \tilde{s}_1 : \mathbf{t}^1 @ p_1, .., \tilde{s}_n : \mathbf{t}^n @ p_n} \quad [\textsc{Var}]$$

$$\frac{\Gamma, \vdash \tilde{e} : \tilde{T} \quad X : \tilde{S}, \mathbf{t}^1, .., \mathbf{t}^n \vdash P \triangleright \tilde{s}_1 : T_1 @ p_1, .., \tilde{s}_n : T_n @ p_n}{\Gamma \vdash \mu X(\tilde{x} : \tilde{S}, \tilde{s}_1, .., \tilde{s}_n).P \triangleright \tilde{s}_1 : \mu\mathbf{t}^1.T_1 @ p_1, \ .., \ \tilde{s}_n : \mu\mathbf{t}^n.T_n @ p_n} \quad [\textsc{Rec}]$$

In [VAR], we introduce type variables $\mathbf{t}^i$ which are associated with X in the premise. Then in [REC] this table is looked up and appropriate recursion is introduced. The algorithm aborts when the constraint in each rule (for example disjointness of channels in delegation in [DELEG] and [SREC]) is violated or unifiability of participants fails. This concludes the introduction of the generation algorithm. We observe:

- Each generation rule precisely mirrors the corresponding typing: hence if we can construct a typing (whose type variables and participant numbers can be instantiated at will), then it shows typability.
- If there *is* a derivation for P under Γ in the typing rules, then the typings used in each derivation at each typing rule specialises what the generation rule would give, so that we know the generation is possible.

Thus the generation algorithm gives a decidable procedure for the typability of annotated program phrases. □

A.2 Proof of Proposition 5.4

Assume $\Gamma \vdash P \triangleright_{s_1..s_m} \Delta$. We call $s_1...s_m$ in $\Gamma \vdash P \triangleright_{s_1..s_m} \Delta$, 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 [INACT]: Zero queue channel to zero queue.

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

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

Case [MCAST], [MACC], [SEND], [RCV], [DELEG], [SREC], [SEL] and [BRANCH], [IF], [VAR], [DEF]: Each of these process construction rules leaves the queue channels unchanged (empty).

Case [CONC]: in the premise, assume $\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta$ and $\Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta'$ the free queues

in P have channels $\tilde{t}_1$ 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 [NRES]: The rule leaves both the queues and the queue channels unchanged.

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

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

By the 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. □

A.3 Proof of Proposition 5.11

For (1), observe that redexes in the base rules in Figure 10, [TR-COM] and [TR-BRA], 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-ISO]. 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 preimages. 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. □

A.4 Proof of Lemma 5.15

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 5.15, page 35).

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:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{52}$$

$$\mathcal{T}_1 \circ \mathcal{T}_2 = \mathcal{T}_1[\mathcal{T}_2] \tag{53}$$

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

$$\mathcal{T}_2 \asymp \mathcal{T}_1 \tag{54}$$

$$\mathcal{T}_2 \circ \mathcal{T}_1 = \mathcal{T}_2[\mathcal{T}_1] \tag{55}$$

Because of the isomorphism by the permutation equivalence for target-disjoint type contexts (cf. Figure 7: 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:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{56}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp \mathcal{T}_3 \tag{57}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ \mathcal{T}_3 = \mathcal{T}_1[\mathcal{T}_2][\mathcal{T}_3] \tag{58}$$

By (57) we have:

$$\mathcal{T}_2 \asymp \mathcal{T}_3 \tag{59}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[\mathcal{T}_3] \tag{60}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[\mathcal{T}_3] = \mathcal{T}_1[\mathcal{T}_2[\mathcal{T}_3]] \tag{61}$$

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:

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{62}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp T \tag{63}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ T = \mathcal{T}_1[\mathcal{T}_2][T] \tag{64}$$

We also always have:

$$\mathcal{T}_2 \asymp T \tag{65}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{66}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[T] = \mathcal{T}_1[\mathcal{T}_2[T]] \tag{67}$$

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

$$\mathcal{T}_2 \asymp T \tag{68}$$
$$\mathcal{T}_1 \asymp \mathcal{T}_2[T] \tag{69}$$
$$\mathcal{T}_1 \circ \mathcal{T}_2[T] = \mathcal{T}_1[\mathcal{T}_2[T]] \tag{70}$$

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

$$\mathcal{T}_1 \asymp \mathcal{T}_2 \tag{71}$$
$$\mathcal{T}_1[\mathcal{T}_2] \asymp T \tag{72}$$
$$\mathcal{T}_1[\mathcal{T}_2] \circ T = \mathcal{T}_1[\mathcal{T}_2][T] \tag{73}$$

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. □

A.5 Proof of Lemma 5.18

First we observe (formally by induction on typing rules) that, in the typed single queue, the length of the queue content and the length of the sequence of types in the typing always coincide.

For the first three rules note they are close in their shape to the rules of the same names in the runtime typing system in Figure 9. In fact by rule induction it is easy to check that the composition $\Delta \circ \tilde{s} : \{\mathcal{T}@\mathtt{p}\}$ in each rule can always be written $\Delta, \tilde{s} : \{\mathcal{T}@\mathtt{p}\}$, that is the composition is always the disjoint union. Thus these three rules are in fact identical with those in 9.

For the next three rules we use induction on the length of the queue content, combined with the first three rules for enqueue. For example, [QVALDQ] is derivable when there is a single value in the queue in the premise, since the singleton case is derivable only through [QNIL] followed by [QVAL] (by the shape of the rules) hence we know $\mathcal{T}$ is empty, hence the premise of [QVALDQ] should have the shape:

$$\Gamma \vdash s : \tilde{v} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k!\langle \tilde{S} \rangle.[\,]@\mathtt{p}\}@\mathtt{p} \tag{74}$$

while the empty queue in the conclusion is immediately typed by [QNIL] as:

$$\Delta \circ \tilde{s} : \{[\,]@\mathtt{p}\}. \tag{75}$$

hence as required. For induction consider it holds for the premise of

$$\Gamma \vdash s : \tilde{v} \cdot \tilde{h} \cdot h' \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k!\langle \tilde{S} \rangle.\tilde{U} \cdot W'[\,]@\mathtt{p}\}@\mathtt{p} \tag{76}$$

where h' is a vector of values or a label or a vector of channels; and W' is the corresponding vector of sorts or a label or a located type. We do a case analysis on h'. First suppose h' is a vector of values. Then the only possible last rule is [QVAL] hence we know:

$$\Gamma \vdash s : \tilde{v} \cdot \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{k!\langle \tilde{S} \rangle.\tilde{U}[\,]@\mathtt{p}\}@\mathtt{p} \tag{77}$$

By induction hypothesis we derive:

$$\Gamma \vdash s : \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{\tilde{U}[\,]@\mathtt{p}\}@\mathtt{p} \tag{78}$$

We now use [QVAL] to obtain:

$$\Gamma \vdash s : h' \cdot \tilde{h} \triangleright_{\tilde{s}'} \Delta \circ \tilde{s} : \{W' \cdot \tilde{U}[\,]@\mathtt{p}\}@\mathtt{p} \tag{79}$$

as required. By precisely the same argument we can treat the two other cases hence as required. The same reasoning applies to the remaining two rules. □

A.6 Remaining Cases of Theorem 5.20

Case [DELEG]: We use the second rule of Lemma 5.18. Suppose we have:

$$\Gamma \vdash s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \mid s : \tilde{h} \triangleright_s \Delta \tag{80}$$

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

$$\Gamma \vdash s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright_{\emptyset} \Delta_1 \tag{81}$$

and

$$\Gamma \vdash s : \tilde{h} \triangleright_s \Delta_2 \tag{82}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (81) can only be inferred from [DELEG] we know, for some p and for some $\tilde{s}$ which includes s,

$$\Delta_1 \;=\; \Delta_1' \circ (\tilde{s} : k!\,\langle T' @ \mathtt{p}' \rangle.T @ \mathtt{p},\ \tilde{t} : T' @ \mathtt{p}') \tag{83}$$

and moreover

$$\Gamma \vdash P \triangleright_{\emptyset} \Delta_1',\ \tilde{s} : T @ \mathtt{p}. \tag{84}$$

On the other hand by $\Delta_1 \asymp \Delta_2$ and (20) we know:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : \mathcal{T}[\,] @ \mathtt{p} \tag{85}$$

By Lemma 5.18, [QSESS], we infer:

$$\Gamma \vdash s : \tilde{h} \cdot \tilde{t} \triangleright_{\tilde{s}'} \Delta_2' \circ \tilde{s} : \{\mathcal{T}[k!\,\langle T @ \mathtt{p}' \rangle.[\,]]@\},\ \tilde{t} : \{T @ \mathtt{p}'\}. \tag{86}$$

By the algebra of located types and type contexts:

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

Thus by applying [CONC] to (81) and (82) we obtain:

$$\Gamma \vdash P \mid s : \tilde{h} \cdot \tilde{t} \triangleright \Delta \tag{87}$$

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

Case [SREC]: By the second to the last rule of Lemma 5.18. Suppose

$$\Gamma \vdash s?(\!(\tilde{t})\!);P \mid s{:}\tilde{t}\cdot\tilde{h} \triangleright_s \Delta \tag{88}$$

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

$$\Gamma \vdash s?(\!(\tilde{t})\!);P \triangleright_\emptyset \Delta_1 \tag{89}$$

and

$$\Gamma \vdash s{:}\tilde{t}\cdot\tilde{h} \triangleright_s \Delta_2 \tag{90}$$

such that $\Delta_1 \circ \Delta_2 = \Delta$. Since (89) can only be inferred from [SREC] we know, for some p and for some $\tilde{s}$ which includes s,

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s} : k?\langle T' @ \mathtt{p}'\rangle.T @ \mathtt{p} \tag{91}$$

and moreover

$$\Gamma \vdash P \triangleright_\emptyset \Delta_1' \circ \tilde{s}{:}\, T @ \mathtt{p},\, \tilde{t}{:}\, T' @ \mathtt{p}' \tag{92}$$

By $\Delta_1 \asymp \Delta_2$ and (90) we know:

$$\Delta_2 \;=\; \Delta_2' \;\circ\; \tilde{s} : k!\langle T' @ \mathtt{p}'\rangle.\mathcal{T} @ \mathtt{p},\; \tilde{t} : T' @ \mathtt{p}' \tag{93}$$

By Lemma 5.18, [QSESSDQ], we infer:

$$\Gamma \vdash s{:}\tilde{h} \triangleright \Delta_2' \;\circ\; \tilde{s} : \mathcal{T} @ \mathtt{p}. \tag{94}$$

Using Proposition 5.14 we obtain:

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

Thus by applying [CONC] to (89) and (90) we obtain:

$$\Gamma \vdash P \mid s{:}\tilde{h} \triangleright \Delta' \tag{95}$$

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

Case [BRANCH]: By the last rule of Lemma 5.18. Suppose

$$\Gamma \vdash s \rhd \{l_i{:}\, P_i\}_{i\in I} \mid s{:}l_j\cdot\tilde{h} \triangleright_s \Delta \tag{96}$$

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

$$\Gamma \vdash s \rhd \{l_i\colon P_i\}_{i\in I} \rhd_\emptyset \Delta_1 \tag{97}$$

and

$$\Gamma \vdash s\colon l_j \cdot \tilde{h} \rhd_s \Delta_2 \tag{98}$$

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:

$$\Delta_2 \;=\; \Delta_2' \circ \tilde{s}\colon k \oplus l_j : \mathcal{T} @ \mathtt{p} \tag{99}$$

where by assumption we have $j \in I$. Since (97) can only be inferred from [BRANCH] and by $\Delta_1 \asymp \Delta_2$, we also know:

$$\Delta_1 \;=\; \Delta_1' \circ \tilde{s}\colon k \,\&\, l_j : T_j @ \mathtt{p} \tag{100}$$

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

$$\Gamma \vdash P_i \rhd_\emptyset \Delta_1' \circ \tilde{s}\colon T_i @ \mathtt{p} \tag{101}$$

for each $i \in I$ (so (100) is inferred using [SUBS]). By Lemma 5.18, [QSELDQ], we infer:

$$\Gamma \vdash s\colon \tilde{h} \rhd \Delta_2' \circ \tilde{s} : \mathcal{T} @ \mathtt{p}. \tag{102}$$

Using Proposition 5.14 we obtain:

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

Thus by applying [CONC] to (97) and (98) we obtain:

$$\Gamma \vdash P \mid s\colon \tilde{h} \rhd \Delta' \tag{103}$$

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

A.7 Proof of Lemma 5.22

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

(a) P contains a unique active receiving (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 [MCAST], [MACC]: Vacuous since in this case the unique active prefix in the process is at a shared name.

Case [SEND], [RCV], [DELEG], [SREC], [SEL] and [BRANCH]: Immediate since there can only be a unique active channel name which is by the given prefixing.

Case [INACT], [IF], [VAR], [DEF], [QNIL], [QVAL], [QSESS], [QSEL]: Vacuous.

Case [CONC]: Suppose

$$\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta, \qquad \Gamma \vdash_{\tilde{t}_2} Q \triangleright \Delta' \tag{104}$$

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 5.11 (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 [NRES]: Vacuous since there is no change either in the process nor in the typing.

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

Case [SUBS]: Vacuous again. □

A.8 Proof of Proposition 5.27

We show the following logically equivalent result:

Claim. (1) If P is simple then

(1-a) no delegation prefix (input or output) occurs in P and

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

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

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

Case [MCAST]: The rule reads:

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

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

Case [MACC]: The rule reads:

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

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

Case [SEND]: The rule reads:

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

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 [RCV]: The rule reads:

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

Precisely the same as in [SEND] above.

Case [DELEG]: The rule reads:

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p}}{\Gamma \vdash_\emptyset s_k!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright \Delta, \tilde{s}\colon k!\langle T' \mathbin{@} \mathtt{p}' \rangle.T \mathbin{@} \mathtt{p},\ \tilde{t}\colon T' \mathbin{@} \mathtt{p}'}$$

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

Case [SREC]: The rule reads:

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p},\ \tilde{t}\colon T' \mathbin{@} \mathtt{p}'}{\Gamma \vdash_\emptyset s_k?(\!(\tilde{t})\!); P \triangleright \Delta, \tilde{s}\colon k?\langle T' \mathbin{@} \mathtt{p}' \rangle.T \mathbin{@} \mathtt{p}}$$

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

Case [SEL]: The rule reads:

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s}\colon T \mathbin{@} \mathtt{p} \qquad j \in I}{\Gamma \vdash_\emptyset s_k \lhd l; P \triangleright \Delta, \tilde{s}\colon k \oplus \{l_i\colon T_i\}_{i \in I}.T \mathbin{@} \mathtt{p}}$$

Precisely the same as [SEND].

Case [BRANCH]: The rule reads:

$$\frac{\Gamma \vdash_\emptyset P_i \triangleright \Delta, \tilde{s}\colon T_i \mathbin{@} \mathtt{p} \qquad \forall i \in I}{\Gamma \vdash_\emptyset\ s_k \rhd \{l_i\colon P_i\}_{i \in I} \triangleright \Delta, \tilde{s}\colon k \,\&\, \{l_i\colon T_i\}_{i \in I} \mathbin{@} \mathtt{p}}$$

Precisely the same as [SEND].

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

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

Hence as required.

For (2) suppose a derivation of P is simple. By the proof of Theorem 5.20, 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. □

A.9 Proof of Lemma 5.29

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 [MCAST], [MACC]: By Proposition 5.27 there can be no free session channels hence vacuous (since (C3) is not satisfied).

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

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

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 [RCV]: The "simple" rule reads:

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

Same as [SEND].

Case [SREC], [DELEG]: By Proposition 5.27 these rules are not used in derivation of a simple process, hence vacuous.

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

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T \mathbin{@} \mathtt{p} \qquad j \in I}{\Gamma \vdash_\emptyset s_k \lhd l; P \triangleright \Delta, \tilde{s} : k \oplus \{l_i : T_i\}_{i \in I}.T \mathbin{@} \mathtt{p}}$$

Again the same as [SEND].

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

$$\frac{\Gamma \vdash_\emptyset P_i \triangleright \Delta, \tilde{s} : T_i \mathbin{@} \mathtt{p} \qquad \forall i \in I}{\Gamma \vdash_\emptyset \ s_k \rhd \{l_i : P_i\}_{i \in I} \triangleright \Delta, \tilde{s} : k \,\&\, \{l_i : T_i\}_{i \in I} \mathbin{@} \mathtt{p}}$$

Again the same as [SEND].

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

Case [CONC]: The rule reads:

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

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 $\mid$ preserves activeness of each prefix.

This is immediate from the definition.

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

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

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

Case [DEF]: As [SUBS] above.

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

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

Case [NRES]: This reads:

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

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 [CRES]: This reads:

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

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. □

B Full Typing Rules for Runtime Processes

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

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

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

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

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

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

$$\frac{\Gamma \vdash_\emptyset P \triangleright \Delta, \tilde{s} : T @ \mathtt{p} \qquad j \in I}{\Gamma \vdash_\emptyset s_k \lhd l ; P \triangleright \Delta, \tilde{s} : k \oplus \{l_i : T_i\}_{i \in I} . T @ \mathtt{p}} \quad [\textsc{Sel}]$$

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

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

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

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

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

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

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

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

Finally we present the typing rules for queues.

$$\Gamma \vdash s : \emptyset \triangleright_s \tilde{s} : \{[\,] \mathbin{@} \mathtt{p}\}_{\mathtt{p}} \qquad \text{[QNIL]}$$

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

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

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