Composable Partial Multiparty Session Types*

Claude Stolze, Marino Miculan[0000−0003−0755−3444], and
Pietro Di Gianantonio[0000−0002−0638−4610]

Dep. of Mathematics, Computer Science and Physics, University of Udine, Italy
firstname.lastname@uniud.it

Abstract. We introduce *partial sessions* and *partial (multiparty) session types*, in order to deal with open systems, i.e., systems with missing components. Partial sessions can be *composed*, and the type of the resulting system is derived from those of its components without knowing any suitable global type nor the types of missing parts. Incompatible types, due to e.g. miscommunications or deadlocks, are detected at the merging phase. We apply these types to a process calculus, for which we prove *subject reduction* and *progress*, so that well-typed systems never violate the prescribed constraints. Therefore, partial session types support the development of systems by incremental assembling of components.

Keywords: Multiparty session types, process algebras, open systems.

1 Introduction

(Multiparty) session types (MPST) are a well-established theoretical and practical framework for the specification of the interactions between components of a distributed systems [15,16,12,17,10,27]. The gist of this approach is to first describe the system's overall behaviour by means of a *global type*, from which a local specification (*local type*) for each component can be derived. The system will behave according to the global type if each component respects its local type, which can be ensured by means of, e.g., static type checking [25,18]. Therefore, session types support a *top-down* style of coding: first the designer specifies the behaviour from a global perspective, then the programmers are given the specifications for their modules. On the other hand, these session types do not fit well *bottom-up* programming models, where systems are built incrementally by composing existing reusable components, possibly with dynamic bindings. In these situations, components could offer "contracts" in the form of, e.g., session types; then, when these components are connected together, we would like to derive the contract for the resulting system from components' ones. The system becomes a new component which can be used in other assemblies, and so on.

To this end, we need to infer the type for an *open* system (i.e., where some parts may be still missing) using the types of the known components, in a compositional way and without knowing any global type. This is challenging. As an

* Work supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS (Methods and Tools for Trustworthy Smart Systems)*.

example, let us consider a protocol from [10] with three participants: a server s, an authorization server a and some client c. First, s sends to c either a request to login, or to cancel. In the first case, c sends a password to a, and a sends a boolean to s (telling whether c is authorized). In the second case, c tells a to quit. Using the syntax of [10], the two server processes have the following types:

$$S_s := c\oplus\left\{\begin{array}{l}\text{login}.a\&\text{auth(Bool)},\\ \text{cancel}\end{array}\right\} \qquad S_a := c\&\left\{\begin{array}{l}\text{passwd(Str)}.s\oplus\text{auth(Bool)},\\ \text{quit}\end{array}\right\}$$

Let us suppose that we have implementations a and s for S_a, S_s. To prevent miscommunications, we would like to verify that these two processes work well together; e.g., we have to ensure that a can send the message auth(Bool) to s iff s is waiting for it. This corresponds to see these two processes as a single system $a|s$, and to check that $a|s$ is well-typed *without knowing the behaviour of clients*; more precisely, we have to figure out a session type for $a|s$ from S_a and S_s. This is difficult, because the link that propagates the choice made by s to a is the missing client c, so we have to "guess" its role without knowing it.

In this paper, we address this problem by introducing *partial sessions* and *partial (multiparty) session types*. Partial session types generalise global types with the possibility to type also partial (or *open*) sessions, i.e., where some participant may be missing. The key difference is that while a global type is a complete, "platonistic" description of the protocol, partial session types represent the *subjective* views from participants' perspectives. We can *merge* two sessions with the same name but from two different "point of views", whenever their types are *compatible*; in this case, we can compute the new, unified, session type from those of the components. In this way, we can guarantee important properties (e.g., absence of deadlocks) about partial session without knowing all participants beforehand, and without a complete global type. In fact, the distinction between local and global types vanishes: local types correspond to partial session types for sessions with a single participant, and global types correspond to *finalized* partial session types, i.e., in which no participant is missing.

Defining "compatibility" and how to merge partial session types is technically challenging. Intuitively, the semantics of a partial session type is the set of all possible execution traces (which depend on internal and external choices). We provide a merging algorithm computing a type covering all the possible synchronizations of these traces. Incompatible types, due to, e.g., miscommunications or deadlocks, are detected when no synchronization is possible. Also the notion of *progress* has to be revisited, to accommodate the case when a partial session cannot progress not due to a deadlock, but due to some missing participant.

The rest of the paper is organized as follows. In Section 2 we introduce a formal calculus for processes communicating over multiparty sessions. Partial session types are presented in Section 3, and the type system is in Section 4. Central to this type system is the merging algorithm, which we describe in Section 5. Subject reduction and progress are given in Section 6. Finally, comparison with related work are in Section 7, and conclusions are in Section 8.

A prototype implementation of the merging algorithm can be found at `https://github.com/cstolze/partial-session-types-prototype`.

2 A calculus for processes over multiparty sessions

Our language for processes is inspired by [10], in turn inspired by [29]; as in those works, we consider *synchronous* communications. We note $p, q, p_1, p', \ldots$ for participant names, belonging to some set $\mathfrak{P}$; $\tilde{p}$ for a finite non-empty set of participants $\{p_1, \ldots, p_n\}$. The syntax of processes is as follows:

$$\begin{aligned} P, Q, R ::= {} & \overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_i.P \mid x^{pq} \triangleleft (P, Q) \mid \overline{x}^{p\tilde{q}}(y).P \mid x^{pq}(y).(P \parallel Q) \\ & \mid \mathrm{close}(x) \mid \mathrm{wait}(x).P \mid (P \mid_x Q) \mid (\nu x)P \mid P + Q \end{aligned}$$

The process $\overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_i.P$ sends label in_i in session x, as participant p, to $\tilde{q}$, and proceeds as P. This label is received by processes of the form $x^{qp} \triangleleft (Q_1, Q_2)$, which then proceeds as Q_i. The process $\overline{x}^{p\tilde{q}}(y).P$ creates a fresh subsession handler y, sends it to $\tilde{q}$, and proceeds as P. This handler is received by processes of the form $x^{qp}(y).(Q \parallel R)$ which forks a process Q (dedicated to y) in parallel to the continuation R (on x)[1]. We compose the processes P and Q through session x with $P \mid_x Q$. $\mathrm{close}(x)$ is the neutral element for $\mid_x$, while $\mathrm{wait}(x).P$ closes session x when all the other participants are gone. $(\nu x)P$ is the standard restriction, and $P + Q$ is the standard non-deterministic choice.

The session name y is bound in expressions of the form $(\nu y)P$, $\overline{x}^{p\tilde{q}}(y).P$, and $x^{pq}(y).(P \parallel Q)$. Free names of a process P (noted $\mathrm{fn}(P)$) are the set of free names of sessions appearing in P.

In order to define the operational semantics, we first introduce the usual notion of contextual equivalence.

Definition 2.1 (Contexts). $\mathcal{C}[\_] ::= \_ \mid (\nu x)\mathcal{C}[\_] \mid (\mathcal{C}[\_] \mid_x P) \mid (P \mid_x \mathcal{C}[\_])$

Definition 2.2 (Equivalence $\equiv$). *The relation $\equiv$ is the smallest equivalence relation closed under contexts (that is, $P \equiv Q \Rightarrow \mathcal{C}[P] \equiv \mathcal{C}[Q]$) satisfying the following rules (we suppose that x, y and z are different session names):*

$$\begin{array}{ll} P \mid_x Q \equiv Q \mid_x P & (P \mid_x Q) \mid_x R \equiv P \mid_x (Q \mid_x R) \\ P \mid_x \mathrm{close}(x) \equiv P & ((\nu x)P) \mid_z Q \equiv (\nu x)(P \mid_z Q) \quad x \notin \mathrm{fn}(Q) \\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & (P \mid_x Q) \mid_y R \equiv P \mid_x (Q \mid_y R) \quad x \notin \mathrm{fn}(R), y \notin \mathrm{fn}(P) \end{array}$$

We can see that processes have the structure of a commutative monoid, thus we will use $\Pi_i^z P_i$ as a shorthand for $P_1 \mid_z \cdots \mid_z P_n$.

Definition 2.3 (Reductions for processes). *The actions α for processes are defined as:*

$$\alpha ::= + \mid x : p \to \tilde{q} : \langle \cdot \rangle \mid x : p \to \tilde{q} : \&\mathrm{in}_i \mid \tau$$

We may write $x : \gamma$ for either $x : p \to \tilde{q} : \langle \cdot \rangle$ or $x : p \to \tilde{q} : \&\mathrm{in}_i$.

We note $P \xrightarrow{\alpha} Q$ for a transition from P to Q under the action α. This relation is defined by the rules in Fig. 1.

[1] From a computational point of view, this "parallel input" corresponds to the programming practice to selectively share sessions between processes. This constructor allows us to enforce a discipline on the shared sessions in order to avoid deadlocks between processes. Moreover, it is motivated by connections with linear logic [29,10].

$$\overline{x}^{p\tilde{q}}(y).R \mid_x \Pi_i^x (x^{q_i p}(y).(P_i \parallel Q_i)) \xrightarrow{x:p\to\tilde{q}:\langle\cdot\rangle} (\nu y)(R \mid_y \Pi_i^y P_i) \mid_x \Pi_i^x Q_i \qquad \text{(send)}$$

$$\overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_j.R \mid_x \Pi_i^x x^{q_i p} \triangleleft (P_{1,i}, P_{2,i}) \xrightarrow{x:p\to\tilde{q}:\&\mathrm{in}_j} R \mid_x \Pi_i^x P_{j,i} \qquad \text{(case)}$$

$$(\nu x)(\mathrm{wait}(x).P) \xrightarrow{\tau} P \quad \text{if } x \notin \mathrm{fn}(P) \qquad \text{(wait)}$$

$$P_1 + P_2 \xrightarrow{+} P_j \quad j \in \{1,2\} \qquad \text{(choice)}$$

$$\frac{P \xrightarrow{x:\gamma} Q}{(\nu x)P \xrightarrow{\tau} (\nu x)Q} \qquad \frac{P \xrightarrow{\alpha} Q \quad \forall \gamma, \alpha \neq x:\gamma}{(\nu x)P \xrightarrow{\alpha} (\nu x)Q}$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid_x R \xrightarrow{\alpha} Q \mid_x R} \qquad \frac{P \xrightarrow{\alpha} Q}{R \mid_x P \xrightarrow{\alpha} R \mid_x Q} \qquad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}$$

Fig. 1. Reduction for processes (where $\tilde{q} = \{q_1, \ldots, q_n\}$ and i ranges over $1..n$).

Note that the typing rules will ensure x does not escape its scope when reducing $(\nu x)(\mathrm{wait}(x).P)$ into P.

Example 2.1. As a running example, let us consider three participants p, q, r. p chooses whether to send a message to r or not; this choice is communicated to r through an intermediate participant q.

$$\begin{aligned} P_p &:= (\overline{x}^{pq} \triangleright \mathrm{in}_1.\overline{x}^{pr}(y).\mathrm{wait}(y).\mathrm{close}(x)) + (\overline{x}^{pq} \triangleright \mathrm{in}_2.\mathrm{close}(x)) \\ P_q &:= x^{qp} \triangleleft (\overline{x}^{qr} \triangleright \mathrm{in}_1.\mathrm{close}(x), \overline{x}^{qr} \triangleright \mathrm{in}_2.\mathrm{close}(x)) \\ P_r &:= x^{rq} \triangleleft (x^{rp}(y)(\mathrm{close}(y) \parallel \mathrm{close}(x)), \mathrm{close}(x)) \end{aligned}$$

Here is an example of execution:

$$\begin{aligned} P_p \mid_x P_q \mid_x P_r &\xrightarrow{+} \overline{x}^{pq} \triangleright \mathrm{in}_2.\mathrm{close}(x) \mid_x P_q \mid_x P_r \\ &\xrightarrow{x:p\to q:\&\mathrm{in}_2} \overline{x}^{qr} \triangleright \mathrm{in}_2.\mathrm{close}(x) \mid_x P_r \\ &\xrightarrow{x:q\to r:\&\mathrm{in}_2} \mathrm{close}(x) \end{aligned}$$

Notice that p can start the session with just q and then wait for input from r:

$$\begin{aligned} P_p \mid_x P_q &\xrightarrow{+} \overline{x}^{pq} \triangleright \mathrm{in}_2.\mathrm{close}(x) \mid_x P_q \\ &\xrightarrow{x:p\to q:\&\mathrm{in}_2} \overline{x}^{qr} \triangleright \mathrm{in}_2.\mathrm{close}(x) \end{aligned}$$

3 Partial multiparty session types

Partial multiparty session types (or just "session types") define the behaviour of a partial session. Their syntax is as follows (where $i \in \{1,2\}$):

$$G ::= p \to \tilde{q} : \&\mathrm{in}_i; G \mid p \to \tilde{q} : \langle G \rangle; G \mid G \oplus G \mid G \,\&\, G \mid \mathrm{end} \mid \mathrm{close} \mid 0 \mid \omega$$

The set of participant names appearing in G is denoted by $\mathrm{fn}(G)$.

Informally, $p \to \tilde{q} : m; G$ means that the participant p sends the message m to the participants in $\tilde{q}$, then the session continues with G. The message $\&\mathrm{in}_i$ is a label, while $\langle G \rangle$ is a fresh session handler of type G. end means that the session ends and the process survives, while close means that the session and the process end. $G_1 \oplus G_2$ (resp. $G_1 \,\&\, G_2$) denotes an *internal* (resp. *external*) choice. Internal choices are made by local participants of the session, contrary to external choices; notice that, in contrast with standard practice, sending or receiving a label $\&\mathrm{in}_i$ is unrelated from the choices done with $\oplus$ or $\&$. Finally, we add the *empty* type 0, which denotes no possible executions (and it is the unit of $\oplus$), and the *inconsistent* type ω, which denotes an error in the session.

Example 3.1. Continuing our running Example 2.1, the following should be the types of each participant.

$$G_p := (p \to q : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close}) \oplus (p \to q : \&\mathrm{in}_2; \mathrm{close})$$
$$G_q := (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_1; \mathrm{close}) \,\&\, (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_2; \mathrm{close})$$
$$G_r := (q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{close} \rangle; \mathrm{close}) \,\&\, (q \to r : \&\mathrm{in}_2; \mathrm{close})$$

These types are actually assigned to P_p, P_q, P_r by the type system we will present in Section 4. But moreover, we would like to be able to type also compositions of these processes; e.g. the types of $P_p \mid_x P_q$ and $P_q \mid_x P_r$ should be the following:

$$\begin{aligned} G_{p,q} := {} & (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close}) \\ & \oplus (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_2; \mathrm{close}) \\ G_{q,r} := {} & (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{close} \rangle; \mathrm{close}) \\ & \,\&\, (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_2; \mathrm{close}) \end{aligned}$$

Moreover, notice that $G_{p,q}$ describes also the behaviour of $P_p \mid_x P_q \mid_x P_r$. As we will see in the next section, these types can be inferred from G_p, G_q, G_r in a compositional way. □

A *chain of communications* is a type of the form $C_1; C_2; \ldots; C_n$. Messages m and *communications* C are defined as follows:

$$m ::= \&\mathrm{in}_i \mid \langle G \rangle \qquad C ::= p \to \tilde{q} : m \mid \mathrm{end} \mid \mathrm{close} \mid 0 \mid \omega \mid 1$$

The communications end, close, 0, ω are also types and they are called *terminal*; the only non-terminal communications are $p \to \tilde{q} : m$ and 1, the latter representing any communication which is not observable from the current process. As such, we can prefix 1 to any session type G by defining $1; G := G$. We denote by $\mathfrak{C}_\omega$ the set of all communications, and by $\mathfrak{C} = \mathfrak{C}_\omega \setminus \{\omega, 0\}$ the set of *executable* communications. We pose $\mathfrak{G}_\omega$ the set of all session types, and $\mathfrak{G}$ the set of session types where there is no occurence of 0 and ω. By default, we will use $\mathfrak{G}_\omega$, while $\mathfrak{G}$ will be used to type processes in Section 4.

In the following, we denote by $S, S_1, \ldots$ sets of participants, and we use the shorthand $S_1 \sharp S_2$ for $S_1 \cap S_2 = \varnothing$. A set of participant S will be called *viewpoint*.

Definition 3.1 (Independence relation). *We define the independence of communications relative to a set of participants S as the smallest symmetric relation I_S such that $C\ I_S\ 1$ for any C, and $(p \to \tilde{q} : m)\ I_S\ (p' \to \tilde{q}' : m')$ whenever $(\{p\} \cup \tilde{q}) \cap (\{p'\} \cup \tilde{q}') \sharp S$.*

Informally, $C_1\ I_S\ C_2$ means that the common participants of C_1 and C_2 are not in S. This independence is relative to the viewpoint of S, because when $C_1\ I_S\ C_2$, the viewpoint of S cannot discriminate between $C_1; C_2; G$ and $C_2; C_1; G$, as is shown in Eq. (1) below. In fact, we can define an equivalence relation between session types relative to S:

Definition 3.2 (Equivalence relation). *For any set of participants S, we define the relation $\simeq_S$ on session types as the smallest congruence verifying the following properties:*

$$C_1; C_2; G \simeq_S C_2; C_1; G \quad (\textit{if } C_1\ I_S\ C_2) \tag{1}$$

$$G_1 \,\&\, (G_2 \oplus G_3) \simeq_S (G_1 \,\&\, G_2) \oplus (G_1 \,\&\, G_3) \qquad G \,\&\, \omega \simeq_S G \qquad G \,\&\, 0 \simeq_S 0$$

$$G_1 \,\&\, (G_2 \,\&\, G_3) \simeq_S (G_1 \,\&\, G_2) \,\&\, G_3 \qquad G \,\&\, G \simeq_S G \qquad G_1 \,\&\, G_2 \simeq_S G_2 \,\&\, G_1$$

$$G_1 \oplus (G_2 \oplus G_3) \simeq_S (G_1 \oplus G_2) \oplus G_3 \qquad G \oplus G \simeq_S G \qquad G \oplus 0 \simeq_S G$$

$$C; (G_1 \,\&\, G_2) \simeq_S (C; G_1) \,\&\, (C; G_2) \qquad C; \omega \simeq_S \omega \qquad C; 0 \simeq_S 0$$

$$C; (G_1 \oplus G_2) \simeq_S (C; G_1) \oplus (C; G_2) \qquad G_1 \oplus G_2 \simeq_S G_2 \oplus G_1$$

We can see that the operations $\oplus$ and $\&$, together with the constants 0 and ω, form a unital commutative semiring. We note $\bigoplus\{G_1, \ldots, G_n\}$ for $G_1 \oplus \ldots \oplus G_n$, and $\mathop{\&}\{G_1, \ldots, G_n\}$ for $G_1 \,\&\, \ldots \,\&\, G_n$. In particular, $\bigoplus \varnothing = 0$, and $\mathop{\&} \varnothing = \omega$. Eq. (1) allows for the "out of order" execution of independent communications. Notice that in general $G \oplus \omega \not\simeq_S \omega$ because the behaviour of a process of type $G \oplus \omega$ is not necessarily always inconsistent.

The fact that choices $G_1 \oplus G_2$ or $G_1 \,\&\, G_2$ are unrelated from the action of sending a choice allows us to move these operators around without changing the meaning. Hence, we can consider *disjunctive normal forms* of session types.

Definition 3.3 (Disjunctive Normal Form). *A session type G is in Disjunctive Normal Form (DNF), if it is of the form $\bigoplus\{\mathop{\&} A_1, \ldots, \mathop{\&} A_n\}$ with the A_i being sets of chains of communications where every message $\langle G' \rangle$ is in DNF.*

In DNF a type can be seen as a set of sets of traces (sequences of communications), the intuition being that a trace describes a single possible interaction of a process. A set of traces defines a deterministic strategy followed by a single process P, describing how P reacts for any possible choice from other processes. A set of sets of traces describes all the possible strategies that P can follow once it has selected all its possible internal choices. So, describing a behaviour in DNF is like saying that a process P starts by anticipating all possible internal choices for all possible interactions during execution. After that, P becomes deterministic and reacts in a single possible way to communications of other processes.

The equivalence relation on types allows us to rewrite any type in a DNF.

Proposition 3.1. *For any type G and set of participants S, we can compute a G' in DNF such that $G' \simeq_S G$.*

4 Type system

In this section we introduce the type system for processes. A key point is that the type of a session are relative to the participants of that session.

Definition 4.1 (Environment). *A* typing declaration *for session x is a triple $x : \langle G \mid S \rangle$ where $G \in \mathfrak{G}$ and $S \subseteq \mathfrak{P}$. S is the set of* local participants *of x.*

An environment Γ is a finite set of typing declarations $\Gamma = x_1 : \langle G_1 \mid S_1 \rangle, \ldots, x_n : \langle G_n \mid S_n \rangle$, such that $x_1, \ldots x_n$ are all distinct.

The main differences between our environments and those in [10] are that session types replace local types, and each session is endowed with a set of local participants, in addition to its session type.

Definition 4.2 (Equivalent environments). *We define $\simeq$ on environments as the smallest equivalence relation satisfying the following rule:*

$$\frac{\Gamma_1 \simeq \Gamma_2 \quad G_1 \simeq_S G_2}{\Gamma_1, x : \langle G_1 \mid S \rangle \simeq \Gamma_2, x : \langle G_2 \mid S \rangle}$$

The typing judgment is $P \vdash \Gamma$, whose rules are shown in Fig. 2.

Rules (*send*), (*recv*), (sel_i), and (*case*) deal with communication. Differently from most type systems (see e.g. [10]), the send and receive actions are typed by the same global type, and not by dual types: in our approach the duality is given by the set of participants, which is either the sender or the receiver.

Rules (*close*) and (*wait*) correspond respectively to the 1 and $\bot$ rules in linear logic, and they both assume there is no named participant, therefore the set of inner participants in the conclusion is empty.

Rule (+) types an internal choice between two processes, but this internal choice is not done for a single session but for the whole process, hence we need to add $\oplus$ to every type. If the internal choice is irrelevant for some session x, that is, we have $x : \langle G \mid S \rangle$ in the two premises, then in the conclusion we would have $x : \langle G \oplus G \mid S \rangle$, which is equivalent to the former. We can of course rewrite types into equivalent ones with rule ($\simeq$).

Rule (ν) allows us to create a local, restricted session. To correctly type the local session, we need to check that the its type is complete, since no other participants will be able to join that session afterward. To this end, we introduce the notion of *finalized* session type. Intuitively, a type is finalized for a given viewpoint (i.e., a set of participants) if all participants involved in the session are in the viewpoint, there are no occurence of ω or close (because we need to avoid deadlocks and miscommunications), and that the end of the session is not the end of the process (because we are within a subsession).

Definition 4.3 (Finalized session type). *The judgment $G \downarrow S$, meaning that the session type G is finalized for the set of participants S, is defined as follows:*

$$\frac{P \vdash \Gamma, y : \langle G_1 \mid \{p\}\rangle, x : \langle G_2 \mid \{p\}\rangle}{\overline{x}^{p\tilde{q}}(y).P \vdash \Gamma, x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \mid \{p\}\rangle} \ (send)$$

$$\frac{P \vdash \Gamma_1, y : \langle G_1 \mid \{q\}\rangle \quad Q \vdash \Gamma_2, x : \langle G_2 \mid \{q\}\rangle \quad q \in \tilde{q}}{x^{qp}(y).(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \mid \{q\}\rangle} \ (recv)$$

$$\frac{P \vdash \Gamma, x : \langle G \mid \{p\}\rangle}{\overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_i.P \vdash \Gamma, x : \langle p \to \tilde{q} : \&\mathrm{in}_i; G \mid \{p\}\rangle} \ (sel_i)$$

$$\frac{P \vdash \Gamma, x : \langle G_1 \mid \{q\}\rangle \quad Q \vdash \Gamma, x : \langle G_2 \mid \{q\}\rangle \quad q \in \tilde{q}}{x^{qp} \triangleleft (P, Q) \vdash \Gamma, x : \langle (p \to \tilde{q} : \&\mathrm{in}_1; G_1) \,\&\, (p \to \tilde{q} : \&\mathrm{in}_2; G_2) \mid \{q\}\rangle} \ (case)$$

$$\frac{P \vdash x_1 : \langle G_1 \mid S_1\rangle, \ldots, x_n : \langle G_n \mid S_n\rangle \quad Q \vdash x_1 : \langle G'_1 \mid S_1\rangle, \ldots, x_n : \langle G'_n \mid S_n\rangle}{P + Q \vdash x_1 : \langle G_1 \oplus G'_1 \mid S_1\rangle, \ldots, x_n : \langle G_n \oplus G'_n \mid S_n\rangle} \ (+)$$

$$\frac{P \vdash \Gamma_1, x : \langle G_1 \mid S_1\rangle \quad Q \vdash \Gamma_2, x : \langle G_2 \mid S_2\rangle \quad S_1 \sharp S_2 \quad G_3 \simeq_{S_1 \uplus S_2} G_1 \ {}^{S_1}\!\vee^{S_2} G_2}{P \mid_x Q \vdash \Gamma_1, \Gamma_2, x : \langle G_3 \mid S_1 \uplus S_2\rangle} \ (|)$$

$$\frac{}{\mathrm{close}(x) \vdash x : \langle \mathrm{close} \mid \varnothing\rangle} \ (close) \qquad \frac{P \vdash \Gamma}{\mathrm{wait}(x).P \vdash \Gamma, x : \langle \mathrm{end} \mid \{p\}\rangle} \ (wait)$$

$$\frac{P \vdash \Gamma, x : \langle G \mid S\rangle \quad G \downarrow S}{(\nu x)P \vdash \Gamma} \ (\nu) \qquad \frac{P \vdash \Gamma \quad \Gamma \simeq \Gamma'}{P \vdash \Gamma'} \ (\simeq)$$

$$\frac{P \vdash \Gamma, x : \langle G \mid S_1\rangle \quad S_2 \sharp \mathrm{fn}(G)}{P \vdash \Gamma, x : \langle G \mid S_1 \cup S_2\rangle} \ (extra)$$

Fig. 2. Type system for processes.

$$\frac{\{p\} \cup \tilde{q} \subseteq S \quad G_1 \downarrow \{p\} \cup \tilde{q} \quad G_2 \downarrow S}{p \to \tilde{q} : \langle G_1\rangle; G_2 \downarrow S} \qquad \frac{G_1 \downarrow S \quad G_1 \simeq_S G_2}{G_2 \downarrow S} \qquad \frac{}{\mathrm{end} \downarrow S}$$

$$\frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \oplus G_2 \downarrow S} \qquad \frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \,\&\, G_2 \downarrow S} \qquad \frac{\{p\} \cup \tilde{q} \subseteq S \quad G \downarrow S}{p \to \tilde{q} : \&\mathrm{in}_i; G \downarrow S} \qquad \frac{}{0 \downarrow S}$$

Rule (|) is one of the key novelties of this type system. This rule allows us to connect two processes through a shared session *merging* their respective types. The shared session has a merged type, computed by $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$. The definition of this operator is quite complex and is postponed to Section 5. For the time being, it is enough to know that $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$ may not be in $\mathfrak{G}$, e.g. when G_1, G_2 are not compatible. To guarantee that only valid types are used for the merged session, we have to find some $G_3 \in \mathfrak{G}$ such that $G_3 \simeq_{S_1 \uplus S_2} G_1 \ {}^{S_1}\!\vee^{S_2} G_2$.

The (*extra*) rule allows us to add participants which actually do not interact with the sessions; this is needed for the Subject Reduction.

Remark 4.1. Our rule for parallel composition is similar to a cut rule for linear logic. It may be interesting to compare our rule with the cut rule for linear logic [14], that for binary session types [29], and that for multiparty session types [10]:

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta} \qquad \frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}}{(\nu x : A)(P \mid Q) \vdash \Gamma, \Delta} \qquad \frac{P_i \vdash \Gamma_i, x^{p_i} : A_i \quad G \vDash \{p_i : A_i\}_i}{(\nu x : G)(\Pi_i^x P_i) \vdash \{\Gamma_i\}_i}$$

Each of these rules corresponds to the applications of two rules of our system: the rule (|) which merges partial sessions, and the rule (ν) which closes the session. For instance, if we assume that A_1, A_2, and B are suitable session types, we have the following derivation:

$$\frac{\dfrac{P \vdash \Gamma, x : \langle A_1 \mid S_1 \rangle \quad Q \vdash \Delta, x : \langle A_2 \mid S_2 \rangle \quad A_1 \ {}^{S_1}\!\vee^{S_2} A_2 \simeq_{S_1 \uplus S_2} B}{P \mid_x Q \vdash \Gamma, \Delta, x : \langle B \mid S_1 \uplus S_2 \rangle} \quad B \downarrow S_1 \uplus S_2}{(\nu x)(P \mid_x Q) \vdash \Gamma, \Delta}$$

In the case of a multiparty session involving n participants, we would apply (|) $n-1$ times, and then the (ν) rule to close the session. This correspondence (in a logical setting and for binary choreographies) have been previously observed in [9], where the cut rule above is split into two rules (called (Conn) and (Scope)).

5 Merging partial session types

The central part of the type system is the merging algorithm that infers the result of interaction of two partial session types. In this section, we will define the merge function $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$, where G_1, G_2 describe the behavior of a session from the viewpoint of the local participants found in the set S_1 and S_2, respectively. $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$ then describes the behaviour of the session from the unified viewpoint $S_1 \cup S_2$. In particular, if G_1 and G_2 are intuitively incompatible, then $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$ should contain some occurrence of ω.

To merge two types, we can consider them in DNF; in this way we can recursively reduce the problem to merging chains of communications. Informally, we merge two sequences of communications by considering all possible reorderings which are compatible with each other. This give us a set of all possible merged behaviours, which we glue together using external choices (&). Thus, two types are compatible if they can agree on at least a pair of merged sequences of communications, whatever their internal choices; if no such sequences exist, we get ω as result. Extra complexity is given by the fact that a single communication in the form $p \to \tilde{q} : \langle G \rangle$ contains a general type; therefore, the function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ for merging single communications and the function $G_1 \ {}^{S_1}\!\vee^{S_2} G_2$ for merging session types are mutually recursive.

We also need the following helper functions:

- the partial function $\mathsf{cont}_S(G, C)$ takes a chain of communications G and a communication C as input, and returns a type that corresponds to what remains in G after having executed C (up to $\simeq_S$)

- the decidable predicate $C_1 \ {}^{S_1}\heartsuit^{S_2} \ C_2$ tells us whether C_1 and C_2 are mergeable (from their respective viewpoints S_1, S_2)
- the total function $\mathsf{sync}_{S_1,S_2}(G_1, G_2)$ takes two chains of communications G_1 and G_2 as input, and returns all possible tuples (C_1, G_1', C_2, G') such that $C_1; G_1' \simeq_{S_1} G_1$, $C_2; G_2' \simeq_{S_2} G_2$ and C_1 and C_2 are mergeable
- finally, the partial function $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ takes a (partial) function $f : \mathfrak{C} \times \mathfrak{C} \rightharpoonup \mathfrak{C}$ and two session types in DNF as arguments, and maps f on the pair (G_1, G_2).

We can then define the partial function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ and the total function $G_1 \ {}^{S_1}\vee^{S_2} \ G_2$. These functions are actually non-deterministic, but $G_1 \ {}^{S_1}\vee^{S_2} \ G_2$ is deterministic up to $\simeq$.

In order to prove termination of these functions, we define the *length* l of session types; besides, we define also the *height* h of communications and session types as the maximal number of nested subsessions. Formally, we have:

$$\begin{aligned} l(G_1 \,\&\, G_2) &:= l(G_1) + l(G_2) & h(G_1 \,\&\, G_2) &:= \max(h(G_1), h(G_2)) \\ l(G_1 \oplus G_2) &:= l(G_1) + l(G_2) & h(G_1 \oplus G_2) &:= \max(h(G_1), h(G_2)) \\ l(C; G) &:= 1 + l(G) & h(C; G) &:= \max(h(C), h(G)) \\ l(G) &:= 1 \quad \text{otherwise.} & h(p \to \tilde{q} : \langle G \rangle) &:= 1 + h(G) \\ & & h(C) &:= 0 \quad \text{otherwise.} \end{aligned}$$

5.1 Mapping merging functions over session types

Definition 5.1 (cont). *The partial function* $\mathsf{cont}_S(G, C)$ *takes as input a chain of communications* G *and a communication* C*, and returns some* G' *in DNF such that* $G \simeq_S C; G'$*. It is undefined if such* G' *does not exist.*

Intuitively, $\mathsf{cont}_S(G, C)$ is a kind of Brzozowski derivative that tells us what happens in G after the communication C.

Proposition 5.1. *The function* cont *is computable, and moreover* $l(C; G') = l(G)$ *and* $h(C; G') = h(G)$*.*

Note that $\mathsf{dom}(\mathsf{cont}_S(G, \_))$ is finite, and can be computed using Eq. (1) repeatedly.

Definition 5.2 (Function sync). *Let* G_1, G_2 *be chains of communications in DNF. Let* $A_1 = \{(C, G') \mid C \in \mathsf{dom}(\mathsf{cont}_S(G_1, \_)), G' = \mathsf{cont}_S(G_1, C)\}$*, and* $A_2 = \{(C, G') \mid C \in \mathsf{dom}(\mathsf{cont}_S(G_2, \_)), G' = \mathsf{cont}_S(G_2, C)\}$*. We then define:*

$$\mathsf{sync}_{S_1,S_2}(f)(G_1, G_2) := \{(C_1, C_2, G_1', G_2') \mid (C_1, C_2) \neq (1, 1), f(C_1, C_2) \text{ is defined}, (C_1, G_1') \in A_1, (C_2, G_2') \in A_2\}.$$

Intuitively, $\mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ returns a set containing all possible pairs of communications that can be merged, as well as their continuations.

It is important to know whether a communication C_1 (from the viewpoint S_1) and a communication C_2 (from the viewpoint S_2) can correspond to the same communication; in this case, we say that they are *mergeable*. Formally, this notion is defined by the following relation.

Definition 5.3 (Mergeability). *We define* $C_1 \ {}^{S_1}\heartsuit^{S_2} \ C_2$ *as follows:*

$$\frac{\{p\} \cup \tilde{q}_1 \cup \tilde{q}_2 \subseteq S_1 \cup S_2 \Rightarrow (G_1 \ {}^{S_1}\vee^{S_2} \ G_2) \downarrow S_1 \cup S_2 \quad p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \to \tilde{q}_1 : \langle G_1 \rangle \ {}^{S_1}\heartsuit^{S_2} \ p \to \tilde{q}_2 : \langle G_2 \rangle}$$

$$\frac{p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \to \tilde{q}_1 : \&\mathrm{in}_i \ {}^{S_1}\heartsuit^{S_2} \ p \to \tilde{q}_2 : \&\mathrm{in}_i} \qquad \frac{}{1 \ {}^{S_1}\heartsuit^{S_2} \ 1}$$

$$\frac{C_2 \ {}^{S_2}\heartsuit^{S_1} \ C_1}{C_1 \ {}^{S_1}\heartsuit^{S_2} \ C_2} \qquad \frac{(\{p\} \cup \ \tilde{q}) \sharp S_1}{1 \ {}^{S_1}\heartsuit^{S_2} \ p \to \tilde{q} : m} \qquad \frac{}{\mathrm{close} \ {}^{S_1}\heartsuit^{S_2} \ \mathrm{close}} \qquad \frac{}{\mathrm{close} \ {}^{S_1}\heartsuit^{S_2} \ \mathrm{end}}$$

The first rule deserves some explanations. In the first hypothesis, G_1 and G_2 describe sessions whose participants can be only in $\{p\} \cup \tilde{q}_1 \cup \tilde{q}_2$; if all these participants are in $S_1 \cup S_2$, then after the merge all the participants are present and therefore the communication must be safe, because no other participant may join later. This means that, in this case, we have to check that the merge of G_1 and G_2 is finalized. The second hypothesis (and dually the third one) corresponds to the fact that in the (*send*) rule of Fig. 2, the sender specifies all receiving participants, while in (*recv*) a receiver may not know about other receivers; therefore, if $p \to \tilde{q}_1 : \langle G_1 \rangle$ describes the communication from the point of view of the sender (i.e., $p \in S_1$), then $\tilde{q}_2$ is a set of receivers only, and must be contained in $\tilde{q}_1$. The fourth (and dually the fifth) hypothesis means that if a participant which is known to a process (i.e., in S_1) appears as receiver for other process (i.e., in $\tilde{q}_2$), then it must appear as a received also by the first process.

Proposition 5.2. $C_1 \ {}^{S_1}\heartsuit^{S_2} \ C_2$ *is decidable.*

Definition 5.4 (Function map). *Let* S_1, S_2 *be two sets of participants, two types* $G_1, G_2 \in \mathfrak{C}$ *and a (partial) function* $f : \mathfrak{C} \times \mathfrak{C} \rightharpoonup \mathfrak{C}$ *such that*

- G_1 *and* G_2 *are in DNF*
- *for any* C_1, C_2, *we have that* $f(C_1, C_2)$ *is a terminal communication iff it is defined and both* C_1 *and* C_2 *are terminal*
- *if* $f(C_1, C_2)$ *is defined, then either both* C_1 *and* C_2 *are terminal, or none of them are.*

Then, $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ *is defined recursively over* G_1, G_2 *as follows:*

- *First cases:*

$$\mathsf{map}_{S_1,S_2}(f)(G_1 \oplus G_2, G_3) := \mathsf{map}_{S_1,S_2}(f)(G_1, G_3) \oplus \mathsf{map}_{S_1,S_2}(f)(G_2, G_3)$$
$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2 \oplus G_3) := \mathsf{map}_{S_1,S_2}(f)(G_1, G_2) \oplus \mathsf{map}_{S_1,S_2}(f)(G_1, G_3)$$

- *If neither of the cases above apply, then we have:*

$$\mathsf{map}_{S_1,S_2}(f)(G_1 \,\&\, G_2, G_3) := \mathsf{map}_{S_1,S_2}(f)(G_1, G_3) \,\&\, \mathsf{map}_{S_1,S_2}(f)(G_2, G_3)$$
$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2 \,\&\, G_3) := \mathsf{map}_{S_1,S_2}(f)(G_1, G_2) \,\&\, \mathsf{map}_{S_1,S_2}(f)(G_1, G_3)$$

– *If G_1, G_2 are both chains of communications and at least one of them is not a terminal communication, we pose $B := \mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ and we have:*
  - *If G_1 or G_2 ends with 0, $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := 0$.*
  - *If G_1 or G_2 ends with ω, or if $B = \varnothing$, then $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \omega$.*
  - *Otherwise:*

$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \&\{f(C_1, C_2); \mathsf{map}_{S_1,S_2}(f)(G_1', G_2') \mid (C_1, C_2, G_1', G_2') \in B\}$$

– *If G_1 and G_2 are both terminal communications, then:*

$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \begin{cases} 0 & \text{if } G_1 \text{ or } G_2 \text{ is } 0 \\ f(G_1, G_2) & \text{if } f(G_1, G_2) \text{ is defined} \\ \omega & \text{otherwise.} \end{cases}$$

The two conditions on f guarantee that $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ is well-defined in the last two cases, when f is applied to G_1, G_2 or to the chains C_1, C_2.

Proposition 5.3. *Termination of* map *is ensured by induction on $l(G_1)+l(G_2)$.*

Note that, when we computing $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$, every application of f is of the form $f_{S_1,S_2}(C_1, C_2)$, where $h(C_1) + h(C_2) \leqslant h(G_1) + h(G_2)$.

5.2 Merging communications and session types

We now define the partial function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ which merges compatible communications C_1 (from the viewpoint S_1) and C_2 (from the viewpoint S_2) and returns, if possible, the new communication from the merged viewpoints $S_1 \cup S_2$. We also define by mutual recursion the merging function for session types, which is just a shorthand for map applied to mcomm:

$$G_1 \ {}^{S_1}\vee^{S_2} \ G_2 := \mathsf{map}_{S_1,S_2}(\mathsf{mcomm}_{S_1,S_2})(G_1, G_2)$$

We suppose that G_1 and G_2 are in DNFs, but it can be applied to any session types by rewriting them in DNF.

Definition 5.5 (Function mcomm). *If $C_1 \ {}^{S_1}\heartsuit^{S_2} \ C_2$, then:*

$$\begin{aligned} \mathsf{mcomm}_{S_1,S_2}(p \to \tilde{q} : \&\mathrm{in}_i, p \to \tilde{q}' : \&\mathrm{in}_i) &:= p \to (\tilde{q} \cup \tilde{q}') : \&\mathrm{in}_i \\ \mathsf{mcomm}_{S_1,S_2}(p \to \tilde{q} : \langle G_1 \rangle, p \to \tilde{q}' : \langle G_2 \rangle) &:= p \to (\tilde{q} \cup \tilde{q}') : \langle G_1 \ {}^{S_1}\vee^{S_2} \ G_2 \rangle \\ \mathsf{mcomm}_{S_1,S_2}(1, C) &:= C \\ \mathsf{mcomm}_{S_1,S_2}(C, 1) &:= C \\ \mathsf{mcomm}_{S_1,S_2}(C, \mathrm{close}) &:= C \\ \mathsf{mcomm}_{S_1,S_2}(\mathrm{close}, C) &:= C \end{aligned}$$

Otherwise, $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ is undefined.

Proposition 5.4. *Termination is ensured by induction on $h(C_1) + h(C_2)$.*

Example 5.1. Continuing Example 3.1, let us recall the types of participants p, r:

$$G_p := G'_p \oplus G''_p \qquad G_r := G'_r \,\&\, G'_r$$
$$G'_p := p \to q : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close} \qquad G''_p := p \to q : \&\mathrm{in}_2; \mathrm{close}$$
$$G'_r := q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{close} \rangle; \mathrm{close} \qquad G''_r := q \to r : \&\mathrm{in}_2; \mathrm{close}$$

We have that:

$$\mathsf{dom}(\mathsf{cont}_{\{p\}}(G'_p)) = p \to q : \&\mathrm{in}_1 \qquad \mathsf{dom}(\mathsf{cont}_{\{p\}}(G''_p)) = p \to q : \&\mathrm{in}_2$$
$$\mathsf{dom}(\mathsf{cont}_{\{r\}}(G'_r)) = q \to r : \&\mathrm{in}_1 \qquad \mathsf{dom}(\mathsf{cont}_{\{r\}}(G''_r)) = q \to r : \&\mathrm{in}_2$$

As a consequence, we have for instance that: $\mathsf{sync}_{\{p\},\{q\}}(G_1, G'_1) = \{(p \to q : \&\mathrm{in}_1, 1, (p \to r : \langle \mathrm{end} \rangle; \mathrm{close}), G'_1), (1, q \to r : \&\mathrm{in}_1, G_1, (p \to r : \langle \mathrm{close} \rangle; \mathrm{close}))\}$.

We have that:

$$G'_p \ {}^{\{p\}}\!\vee^{\{r\}} G'_r = (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close}) \,\&\, (q \to r : \&\mathrm{in}_1; p \to q : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close})$$
$$G'_p \ {}^{\{p\}}\!\vee^{\{r\}} G''_r = (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_2; \omega) \,\&\, (q \to r : \&\mathrm{in}_2; p \to q : \&\mathrm{in}_1; \omega)$$
$$G''_p \ {}^{\{p\}}\!\vee^{\{r\}} G'_r = (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_1; \omega) \,\&\, (q \to r : \&\mathrm{in}_1; p \to q : \&\mathrm{in}_2; \omega)$$
$$G''_p \ {}^{\{p\}}\!\vee^{\{r\}} G''_r = (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_2; \mathrm{close}) \,\&\, (q \to r : \&\mathrm{in}_2; p \to q : \&\mathrm{in}_2; \mathrm{close})$$

and finally

$$\begin{aligned} G_p \ {}^{\{p\}}\!\vee^{\{r\}} G_r &= ((G'_p \ {}^{\{p\}}\!\vee^{\{r\}} G'_r) \,\&\, (G'_p \ {}^{\{p\}}\!\vee^{\{r\}} G''_r)) \oplus \\ &\quad ((G''_p \ {}^{\{p\}}\!\vee^{\{r\}} G'_r) \,\&\, (G''_p \ {}^{\{p\}}\!\vee^{\{r\}} G''_r)) \\ &\simeq_{\{p,r\}} (p \to q : \&\mathrm{in}_1; q \to r : \&\mathrm{in}_1; p \to r : \langle \mathrm{end} \rangle; \mathrm{close}) \oplus \\ &\quad (p \to q : \&\mathrm{in}_2; q \to r : \&\mathrm{in}_2; \mathrm{close}) \end{aligned}$$

6 Subject reduction and Progress

In this section we state two main properties of session types, *subject reduction* and *progress*, which guarantee that "well-typed systems cannot go wrong". To this end, we first define a reduction semantics for partial session types.

Definition 6.1 (Reductions for session types). *Actions γ for session types are defined as*

$$\gamma ::= + \mid p \to \tilde{q} : \langle \cdot \rangle \mid p \to \tilde{q} : \&\mathrm{in}_i$$

We write $G_1 \xrightarrow{\gamma}_S G_2$ for a transition from G_1 to G_2 from the viewpoint of S under the action γ. This relation is defined as follows:

$$G_1 \oplus G_2 \xrightarrow{+}_S G_i \qquad p \to \tilde{q} : \langle G_1 \rangle; G_2 \xrightarrow{p \to \tilde{q} : \langle \cdot \rangle}_S G_2$$

$$p \to \tilde{q} : \&\mathrm{in}_i; G \xrightarrow{p \to \tilde{q} : \&\mathrm{in}_i}_S G \qquad \frac{G_1 \xrightarrow{\gamma}_S G' \quad G_1 \simeq_S G_2}{G_2 \xrightarrow{\gamma}_S G'}$$

Note that transitions are not deterministic, in particular $G \simeq_S G \oplus G$, therefore we always have $G \xrightarrow{+} G$, which is useful in case we are reducing an internal choice which is irrelevant for G.

Definition 6.2 (Reduction for environments). *Reductions for environments are labelled by actions for processes α, and are defined as follows:*

$$\frac{}{\cdot \xrightarrow{\alpha} \cdot} \qquad \frac{}{\Gamma \xrightarrow{\tau} \Gamma} \qquad \frac{G_1 \xrightarrow{+}_S G_2 \quad \Gamma_1 \xrightarrow{+} \Gamma_2}{x : \langle G_1 \mid S\rangle, \Gamma_1 \xrightarrow{+} x : \langle G_2 \mid S\rangle, \Gamma_2}$$

$$\frac{G_1 \xrightarrow{\gamma}_S G_2}{x : \langle G_1 \mid S\rangle, \Gamma \xrightarrow{x:\gamma} x : \langle G_2 \mid S\rangle, \Gamma} \qquad \frac{\Gamma_1 \xrightarrow{y:\gamma} \Gamma_2 \quad x \neq y}{x : \langle G \mid S\rangle, \Gamma_1 \xrightarrow{y:\gamma} x : \langle G \mid S\rangle, \Gamma_2}$$

The type system enjoys the following properties:

Theorem 6.1 (Subject equivalence). *If $P \vdash \Gamma$ and $P \equiv Q$, then $Q \vdash \Gamma$.*

From now on, we can consider processes equal modulo $\equiv$.

Theorem 6.2 (Subject reduction). *If $P_1 \vdash \Gamma_1$ and $P_1 \xrightarrow{\alpha} P_2$, then for some Γ_2, we have $P_2 \vdash \Gamma_2$ and $\Gamma_1 \xrightarrow{\alpha} \Gamma_2$.*

Remark 6.1. In earlier work about MPST, usually subject reduction requires some *consistency* condition over the typing environment Γ (see, e.g., [27]). In our development, this condition is not explicitly needed because the type rules for processes ensure that environments are consistent; hence, the derivability of $P_1 \vdash \Gamma_1$ implies that no session in Γ_1 has the type ω.

Progress In usual session types, the progress property means that well-typed systems can always proceed, and in particular they are deadlock-free. In our case, well-typed systems can still contain processes which cannot proceed not due to a deadlock or miscommunication but due to some missing participant.

Example 6.1. Let us consider $P = \overline{x}^{pq} \triangleright \&\mathrm{in}_1.\mathrm{close}(x)$. This process is typable ($P \vdash x : \langle p \to q : \&\mathrm{in}_1; \mathrm{close} \mid \{p\}\rangle$), yet it is stuck. It can be completed into a redex $P \mid_x Q$, with $Q = x^{qp} \triangleleft (Q_1, Q_2)$. In fact, P can be seen as the *restriction* of $P \mid_x Q$ on session x with participants in $\{p\}$. Hence, P is preempted by x and so it can be considered a correct process, waiting for the missing participant.

Therefore, in order to define the progress property for our system, we need to define the restriction of a process to a given set of local participants.

Definition 6.3 (Restriction). *We define the restriction of a term P on session x with participants in S (noted $P \lfloor_S x$) as follows:*

$$\overline{x}^{p\tilde{q}}(y).P \lfloor_S x = \mathrm{close}(x) \text{ if } p \notin S \qquad x^{pq}(y).(P \parallel Q) \lfloor_S x = \mathrm{close}(x) \text{ if } p \notin S$$

$$\overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_i.P \lfloor_S x = \mathrm{close}(x) \text{ if } p \notin S \qquad x^{pq} \triangleleft (P, Q) \lfloor_S x = \mathrm{close}(x) \text{ if } p \notin S$$

$$P \mid_x Q \lfloor_S x = (P \lfloor_S x) \mid_x (Q \lfloor_S x) \qquad P \lfloor_S x = P \text{ otherwise}$$

Definition 6.4 (Preemption). *We say that a session x with type $G \in \mathfrak{G}$ and local participants S* preempts *P (noted $x : \langle G \mid S\rangle \gg_{\mathrm{g}} P$) when one of these condition occurs:*

- $x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \mid S\rangle \gg_{\mathrm{g}} ((\overline{x}^{p\tilde{q}}(y).R \mid_x \Pi_i^x(x^{q_i p}(y).(P_i \parallel Q_i))) \downarrow_S x) \mid_x P$ *if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G_2 \mid S - \{p, \tilde{q}\}\rangle \gg_{\mathrm{g}} P$*
- $x : \langle p \to \tilde{q} : \&\mathrm{in}_i; G \mid S\rangle \gg_{\mathrm{g}} (\overline{x}^{p\tilde{q}} \triangleright \mathrm{in}_i.R \mid_x \Pi_j^x x^{q_j p} \triangleleft (P_{1,j}, P_{2,j}) \downarrow_S x) \mid_x P$ *if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G \mid S - \{p, \tilde{q}\}\rangle \gg_{\mathrm{g}} P$*
- $x : \langle \mathrm{close} \mid S\rangle \gg_{\mathrm{g}} \mathrm{close}(x)$
- $x : \langle \mathrm{end} \mid S\rangle \gg_{\mathrm{g}} \mathrm{wait}(x).P$
- $x : \langle G_1 \oplus G_2 \mid S\rangle \gg_{\mathrm{g}} U$ *if* $x : \langle G_1 \mid S\rangle \gg_{\mathrm{g}} P$ *or* $x : \langle G_2 \mid S\rangle \gg_{\mathrm{g}} P$
- $x : \langle G_1 \,\&\, G_2 \mid S\rangle \gg_{\mathrm{g}} P$ *if* $x : \langle G_1 \mid S\rangle \gg_{\mathrm{g}} P$ *and* $x : \langle G_2 \mid S\rangle \gg_{\mathrm{g}} P$
- $x : \langle G \mid S\rangle \gg_{\mathrm{g}} P$ *if* $x : \langle G \mid S\rangle \gg_{\mathrm{g}} P'$ *and* $P \equiv P'$

Definition 6.5 (Contextual preemption). *We define $x : \langle G \mid S\rangle \gg_{\mathrm{c}} P$ if for some $\mathcal{C}[\_]$, P', we have that $P \equiv \mathcal{C}[P']$, $x \notin \mathrm{fn}(\mathcal{C}[\_])$, and $x : \langle G \mid S\rangle \gg_{\mathrm{g}} P'$.*

Intuitively, $x : \langle G \mid S\rangle \gg_{\mathrm{c}} P$ means that every local participant in S is ready to trigger its respective communication described in G. As a consequence, there is no deadlock for x: if all the concerned participants are present there is a redex, otherwise we are blocked due to the absence of some sender or receiver. The following lemma states that if a session is finalized and preempted, then the process (with the session restricted) contains a redex.

Lemma 6.1. *1. If $G \downarrow S$ and $x : \langle G \mid S\rangle \gg_{\mathrm{g}} P$, then $(\nu x)P$ has a redex.*
2. If $G \downarrow S$ and $x : \langle G \mid S\rangle \gg_{\mathrm{c}} P$, then $(\nu x)P$ has a redex.

Theorem 6.3 (Progress). *If $P \vdash \Gamma$ then there is a redex in P, or for some $x : \langle G \mid S\rangle \in \Gamma$ we have $x : \langle G \mid S\rangle \gg_{\mathrm{c}} P$.*

7 Related work

The problem of composing session types has been faced in several related work. Compositional choreographies are discussed in [24], with the same motivations as ours, but from a different perspective. The authors manage to compose choreographies using global types, but the global type of shared channels has to be the same. This is in contrast with our approach, where the processes may have different session types that we merge during the composition. Moreover, also their typing judgments use sets of participants (there called *roles*); more precisely, the types for channels keep track of the "active" role, the set of all roles in the global type, and the roles actually implemented by the choreography under typing. On the other hand, we do not need to specify neither the complete set of participants nor the "active" role, in typing sessions.

Synthesis of choreography from local types has been studied also in [21], but with no notion of "partial types" and no distinction between internal/external choice. Graphical representations of choreographies (as communicating finite-state machines) and global types have been used in [22], where an algorithm for constructing a global graph from asynchronous interactions is given.

An interesting approach based on *gateways* has been investigated in [5,2,4,3]: two independent global types G_1 and G_2 with different participants can be composed through participant h in G_1 and k in G_2 where h and k relay the message they receive to each other. Therefore, in this approach the two session types G_1, G_2 are connected by the gateway but not really merged, as in our approach. Finally, [27] do not use global types altogether: behaviours of systems are represented by sets of local types, over which no consistency conditions are required, and behavioural properties can be verified using model checking techniques.

A problem similar to ours is considered in [8], where the authors introduce a type system for the Conversation Calculus, a model for service-oriented computing. Conversation types of parallel processes can be merged like in our approach, but the underlying computational model is quite different.

Semantics of concurrent processes can be given using Mazurkiewicz trace languages [26]. Semantics can also be defined using event structures, as in [11], where they are used for defining equivalent semantics for processes and their global types. Interestingly, the semantics for global types proposed in [11] is similar to the representation of Mazurkiewicz trace languages as event structures given in [26]. Mazurkiewicz trace languages have been also used to characterize testing preorders on multiparty scenarios [13]. A denotational semantics based on Brzozowski derivatives that corresponds to bisimilarity is given in [20].

Another semantics of processes (but for binary session types) that records exchanged informations is given in [1]. This semantics is similar to the relation-based model of linear logic [6], and not based on traces. It would interesting to investigate if this alternative semantics can be extended to MPST and to interpret the merge operation. The relationship between category theory and session types has been investigated also in [28,19].

8 Conclusions

In this paper, we have introduced *partial sessions* and *partial (multiparty) session types*, extending global session types with the possibility to type also sessions with missing participants. Sessions with the same name but observed by different participants can be merged if their types are *compatible*; in this case, the type for the unified session can be derived compositionally from the types of components. To this end we have provided a merging algorithm, which allows us to detect incompatible types, due to miscommunications or deadlocks, as early as possible; this differs from usual session type systems which delay all the checks to when the system is completed (i.e., at the restriction rule). Therefore, in this theory the distinction between local and global types vanishes. We have also generalised the notion of *progress* to accommodate the case when a partial session cannot progress not due to a deadlock, but to some missing participant.

Future work. An interesting application of partial session types would be in the verification of composition of components, like e.g. containers *a la* Docker; to this end, we can think of defining a typing discipline similar to the one presented in this paper, but tailored for a formal models of containers, like that in [7].

We claim that for the type system presented in this paper both type checking and type inference are decidable. The idea is that, in order to be typable, the structure of a process has to match the structure of the type(s), up-to type equivalence; hence, the typing derivation is bounded by the complexity of process terms. At worst, this bound is exponential, as in the application of type equivalence rule we have to explore a possibly exponential space of equivalent types; however, this limit could be improved by some algorithmic machinery concerning the normal form of types, which we leave to future work.

The current merging algorithm returns types that may contain many equivalent subterms; a future work could be to define shorter and more efficient representations. Another interesting aspect of this algorithm is that it is defined by two functions (map and mcomm), which can be updated separately in future variations; in particular, adding recursion only requires to update the function map, while adding new kinds of communication, or changing how communications are merged, only requires to update the function mcomm. The correctness of this algorithm can be proved with respect to a categorical semantics for session types based on traces, which we leave to the extended version of this work.

In this paper we have considered a calculus with synchronous multicast, along the lines of [29,10] and others. However, it would be interesting to extend the definitions and results of this paper to an *asynchronous* version of the calculus. This is not immediate, as it requires non-trivial changes in the typing systems and especially in the (already quite complex) merging operation.

Following the Liskov substitution principle, we could define a subtyping relation by seeing & and $\oplus$ as the meet and join operator of a lattice, respectively. However, a semantical understanding of this subtyping relation is not clear yet.

One intriguing possible extension would be to add some form of *encapsulation*. For instance, if we have the type $p \to q : m_1; q \to r : m_2; p \to r : m_3; \mathrm{close}$ from the viewpoint of $\{q, r\}$, then we could be tempted to "erase" the communication $q \to r : m_2$, since this communication is purely internal, but this erasure would not be compatible with equivalence:

$$p \to q : m_1; q \to r : m_2; p \to r : m_3; \mathrm{close} \not\simeq_{\{q,r\}} p \to q : m_3; q \to r : m_2; p \to r : m_1; \mathrm{close}$$

$$\text{but} \qquad p \to q : m_1; p \to r : m_3; \mathrm{close} \simeq_{\{q,r\}} p \to q : m_3; p \to r : m_1; \mathrm{close}$$

How to add a form of encapsulation to our type system is an open question.

Finally, to guarantee the correctness of most complex proofs and definitions of this paper, it would be useful to formalise them in a proof assistant, like Coq.

Acknowledgments We are grateful to Mariangiola Dezani-Ciancaglini, Marco Peressotti and the anonymous reviewers for their useful remarks on the preliminar version of this paper.

References

1. Atkey, R.: Observed communication semantics for classical processes. In: Yang, H. (ed.) Programming Languages and Systems. pp. 56–82. Springer (2017)

2. Barbanera, F., Dezani-Ciancaglini, M.: Open multiparty sessions. In: Proc. ICE. EPTCS, vol. 304, pp. 77–96 (2019)
3. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., Tuosto, E.: Composition and decomposition of multiparty sessions. J. Log. Algebraic Methods Program. **119** (2021)
4. Barbanera, F., Lanese, I., Tuosto, E.: Composing communicating systems, synchronously. In: ISoLA. Lecture Notes in Computer Science, vol. 12476, pp. 39–59. Springer (2020)
5. Barbanera, F., de' Liguoro, U., Hennicker, R.: Global types for open systems. In: Proc. ICE. EPTCS, vol. 279, pp. 4–20 (2018)
6. Barr, M.: *-autonomous categories and linear logic. Mathematical Structures in Computer Science **1**(2), 159–178 (1991). https://doi.org/10.1017/S0960129500001274
7. Burco, F., Miculan, M., Peressotti, M.: Towards a formal model for composable container systems. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing. pp. 173–175. ACM (2020). https://doi.org/10.1145/3341105.3374121
8. Caires, L., Vieira, H.T.: Conversation types. Theoretical Computer Science **411**(51-52), 4399–4440 (2010)
9. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018)
10. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. Acta Informatica **54**(3), 243–269 (2017)
11. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Event structure semantics for multiparty sessions. In: Models, Languages, and Tools for Concurrent and Distributed Programming, pp. 340–363. Springer (2019)
12. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 238–302 (2016)
13. De Nicola, R., Melgratti, H.: Multiparty testing preorders. In: Trustworthy Global Computing. pp. 16–31. Springer (2015)
14. Girard, J.Y.: Linear logic. Theoretical computer science **50**(1), 1–101 (1987)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Proc. ESOP'98. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proc. POPL 2008. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
18. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming. pp. 13–22 (2015)
19. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: A coalgebraic view on session types and communication protocols. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 375–403. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_14

20. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: A fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290337
21. Lange, J., Tuosto, E.: Synthesising choreographies from local session types. In: CONCUR. Lecture Notes in Computer Science, vol. 7454, pp. 225–239. Springer (2012)
22. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL. pp. 221–232. ACM (2015)
23. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux journal **2014**(239), 2 (2014)
24. Montesi, F., Yoshida, N.: Compositional choreographies. In: Proc. CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 425–439. Springer (2013)
25. Neubauer, M., Thiemann, P.: An implementation of session types. In: International Symposium on Practical Aspects of Declarative Languages. pp. 56–70. Springer (2004)
26. Nielsen, M., Winskel, G.: Models for concurrency. In: Proc. Mathematical Foundations of Computer Science. pp. 43–46 (1991)
27. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
28. Toninho, B., Yoshida, N.: Polymorphic session processes as morphisms. In: Alvim, M.S., Chatzikokolakis, K., Olarte, C., Valencia, F. (eds.) The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday. Lecture Notes in Computer Science, vol. 11760, pp. 101–117. Springer (2019). https://doi.org/10.1007/978-3-030-31175-9_7
29. Wadler, P.: Propositions as sessions. Journal of Functional Programming **24**(2-3), 384–418 (2014)