Dynamic Multirole Session Types

Pierre-Malo Deniélou   Nobuko Yoshida

Imperial College London

Abstract

Multiparty session types enforce structured safe communications between several participants, as long as their number is fixed when the session starts. In order to handle common distributed interaction patterns such as cloud algorithms or peer-to-peer protocols, we propose a new role-based multiparty session type theory where roles are defined as classes of local behaviours that an arbitrary number of participants can dynamically join and leave. We offer programmers a polling operation that gives access to the current set of a role's participants in order to fork processes. Our type system with universal types for polling can handle this dynamism and retain type safety. A multiparty locking mechanism is introduced to provide communication safety, but also to ensure a stronger progress property for joining participants that has never been guaranteed in previous systems. Finally, we present some implementation mechanisms used in our prototype extension of ML.

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

General Terms Theory, Types, Design

Keywords communications, roles, multiparty, session types, mobile processes, dynamic protocols, join-leave, peer-to-peer, auction.

1. Introduction

As a type foundation for structured distributed, communications programming, *session types* [18, 31] have been studied over the last decade for a wide range of process calculi and programming languages. The original binary theory has been generalised to *multiparty session types* [19] in order to guarantee stronger conformance to stipulated session structures between cooperating multiple end-point participants. Since the first work [19] was proposed, the multiparty session type theory has been developed in process calculi [4, 10, 14, 22], and used in several different contexts such as distributed object communication optimisations [29], security [5, 9], design by contract [6], parallel and web service programming [25, 35, 36] and medical guidelines [23], some of which initiated industrial collaborations (see § 6 and 7). While many interaction patterns can be captured in the existing multiparty sessions framework, there are significant limitations for describing and validating *loosely-coupled, ungoverned, dynamic protocols*, since the number of participants is required to be fixed both when the session is designed and when the session execution starts. This makes it unable to express interaction patterns frequently found in messaging oriented middleware and service-oriented computing.

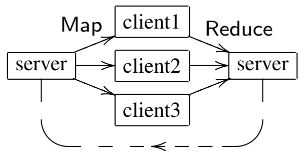
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to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

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

The central underpinning of multiparty session types is that critical properties, such as communication safety (essentially a correspondence between send and receive) and deadlock-freedom, are guaranteed by the combination of two means: first, a static type-checking methodology based on the existence of a *global type* (a description of a multiparty protocol from a global viewpoint) and of its *end-point projections* — the global type is projected to end-point types against which processes can be efficiently type-checked; second, a synchronisation mechanism which ensures that all the well-behaved (i.e. well-typed) participants actually present when the session starts. This paper introduces a new role-based multiparty type system and synchronisation mechanism that, together, can specify, verify and govern dynamically evolving protocols.

In the rest of this section, we illustrate our motivation, approach and solutions through protocols of increasing complexity: **(1) Map/Reduce** (introduction of the notion of roles and universal quantification); **(2) P2P chat** (projection challenges) and **(3) Auction** (branching and communication safety).

(1) Map/Reduce We imagine a server that wants a task to be computed on a cluster made of three clients: the server sends them jobs and they give back their answers. We give a picture illustrating this communication pattern and the corresponding *global multiparty session type* written in the original theory [19].



$$
\begin{aligned}
&G_{org} = \\
&\mu\mathbf{x}.(\mathsf{server}\to\mathsf{client1}\langle\mathsf{Map}\rangle;\\
&\quad \mathsf{server}\to\mathsf{client3}\langle\mathsf{Map}\rangle;\\
&\quad \mathsf{server}\to\mathsf{client2}\langle\mathsf{Map}\rangle;\\
&\quad \mathsf{client1}\to\mathsf{server}\langle\mathsf{Reduce}\rangle;\\
&\quad \mathsf{client2}\to\mathsf{server}\langle\mathsf{Reduce}\rangle;\\
&\quad \mathsf{client3}\to\mathsf{server}\langle\mathsf{Reduce}\rangle);\mathbf{x}
\end{aligned}
$$

This session starts with the server sending asynchronously the messages Map to participants client1, client2 and client3. Each of them answers back with a message Reduce.[1] Recursion $\mu\mathbf{x}$ expresses that an unbounded number of repeated interactions.

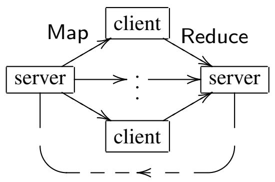
The problem here is that such a session cannot start without one of the clients and cannot handle a fourth client joining or one of the current clients leaving. In the original multiparty sessions, any of these scenarios requires ending the session, writing an appropriate global type for the new situation, and starting a fresh session again.

This paper proposes the theory of *dynamic multirole session types* that can describe global interactions between *roles*, which are classes of participants that share a common behaviour (e.g. the clients in the above example). Dynamism is disciplined by a simple *universally quantified type* that allows to spawn further interactions by polling the set of participants currently playing a given role.

In the above session, we notice that the three clients have the exact same behaviour (receiving a Map message and sending a Reduce message). We call this behaviour the client role and now expect a varying number of participants to inhabit it. On the other hand, the server role is as usual instantiated by exactly one participant and the session does not start without its presence. This dynamic protocol is illustrated by the following picture and global

[1] Since multiparty session types do not support explicit parallelism, we rely on asynchrony to express the desired behaviour.

type with new universal type.



$G = \mu\mathbf{x}.\quad \forall x\,{:}\,\text{client}.\quad \{\text{server}{\to}x\langle\mathsf{Map}\rangle; x{\to}\text{server}\langle\mathsf{Reduce}\rangle\}; \mathbf{x}$
The repeated interaction in the global type G involves a Map message to be sent by the server to every participants x of the client role; the server expecting a Reduce message in answer.

At the type level, such an operation is specified using a universal quantification:

> $\forall x\,{:}\,r.\{G'\}$ polls the current participants $\mathtt{p}_1, ..., \mathtt{p}_n$ of role r and, in parallel processes, binds x to each in the subsequent interaction, as in $G'\{\mathtt{p}_1/x\} \mid ... \mid G'\{\mathtt{p}_n/x\}$

In our example, $G' = \text{server}{\to}x\langle\mathsf{Map}\rangle; x{\to}\text{server}\langle\mathsf{Reduce}\rangle$ is executed in parallel for each client x. Then, the recursion variable $\mathbf{x}$ points the interaction back to its beginning.

Local Types Since the implementation, written here in a variant of the π-calculus, is distributed, the typing system first *projects* the global type to each end-point (local) type. For each role, the projection algorithm computes a local type that describes the behaviour of any participant that wants to play it. The local types for this session are the following:

$$T_{\text{client}} = \mu\mathbf{x}.\quad ?\langle\text{server}, \mathsf{Map}\rangle; !\,\langle\text{server}, \mathsf{Reduce}\rangle; \mathbf{x}$$
$$T_{\text{server}} = \mu\mathbf{x}.\quad \forall x\,{:}\,\text{client}.\{!\,\langle x, \mathsf{Map}\rangle; ?\langle x, \mathsf{Reduce}\rangle\}; \mathbf{x}$$

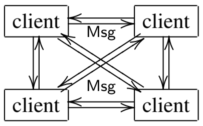
First, the client behaviour T_{client} is straightforward as it is only involved in two messages at each iteration with the server. The local type of the client expresses that it expects a message Map from the server ($?\langle\text{server}, \mathsf{Map}\rangle$) and that it sends a message Reduce as an answer ($!\,\langle\text{server}, \mathsf{Reduce}\rangle$). The server role is involved in all the messages of this session. We note the presence of the quantification over all x playing the client role.

Processes We write some process examples that would be well typed against the local types. The session identifier s denotes an active session:

$$P_{\text{client}}(z) = a[z\,{:}\,\text{client}](s).\mu X.s?\langle\text{server}, \mathsf{Map}\rangle; s!\,\langle\text{server}, \mathsf{Reduce}\rangle; X$$
$$P_{\text{server}}(z) = a[z\,{:}\,\text{client}](s).\mu X.s\forall(x\,{:}\,\text{client}).\{s!\,\langle x, \mathsf{Map}\rangle; s?\langle x, \mathsf{Reduce}\rangle\}; X$$

A session starts through the join operation ($a[z\,{:}\,\text{client}](s)$) which gets the session name s of a running session advertised on a. A participant z playing the client with $P_{\text{client}}(z)$ is simply exchanging messages Map and Reduce with the server through sending ($s!$) and receiving ($s?$) operations. The server needs to fork subprocesses for its interactions with each client. To this effect, the polling operation $s\forall(x\,{:}\,\text{client}).\{s!\,\langle x, \mathsf{Map}\rangle; s?\langle x, \mathsf{Reduce}\rangle\}$ creates as many processes $s!\,\langle x, \mathsf{Map}\rangle; s?\langle x, \mathsf{Reduce}\rangle$ as there are participants x playing the client role. Note that late joining client participants are incorporated in the session at each iteration: the repetition of the polling operation $s\forall(x\,{:}\,\text{client})$ is able to ensure a safe interaction between all parties.

(2) Peer-to-peer chat In this session, there is only one role, the client, whose behaviour is to always broadcast its messages to all the other clients. We give a representation of the interaction when four clients are present.



$G = \mu\mathbf{x}.(\forall x\,{:}\,\text{client}.\forall y\,{:}\,\text{client} \setminus x.\{x{\to}y\langle\mathsf{Msg}\rangle\}); \mathbf{x}$
This type features a double quantification which specifies that each pair of clients x, y will interact in the form of a unique message $x{\to}y\langle\mathsf{Msg}\rangle$. The explicit exclusion of x from the list of clients y prevents self-sent messages.

This second example shows the projection difficulties that arise from quantification.

Local Types To illustrate the projection of nested quantifiers, we first rely on our intuition: each client should send a message Msg to every other client and, concurrently, should expect a message Msg from each of them.

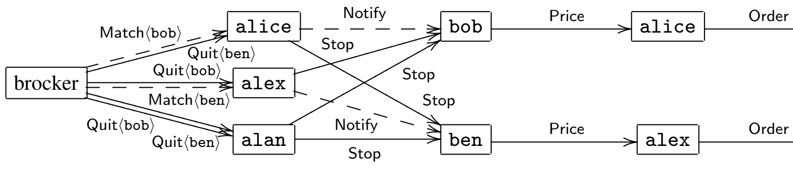
$$T_{\text{client}}(z) = \mu\mathbf{x}.(\forall y\,{:}\,\text{client} \setminus z.\{!\,\langle y\,{:}\,\text{client}, \mathsf{Msg}\rangle\} \mid \forall x\,{:}\,\text{client} \setminus z.\{?\langle x\,{:}\,\text{client}, \mathsf{Ms}$$

Let us examine how the projection algorithm gives this local type. Suppose we project for a generic client z. The first quantifier $\forall x\,{:}$ client of the global type necessarily involves z, meaning that among these parallel processes there is exactly one where x is z. In the other parallel processes, although x is not z, z can still be involved. The projection of the second nested quantifier $\forall y\,{:}\,\text{client} \setminus x$ works in the same way. This is why the first parallel part is $\forall y\,{:}\,\text{client} \setminus z.!\,\langle y\,{:}\,\text{client}, \mathsf{Msg}\rangle$, which explicitly excludes the possible $!\,\langle z, \mathsf{Msg}\rangle$.[2]

Process Once the local types are known, the client processes have a similar structure, including the explicit polling restriction, written $s\forall(y\,{:}\,\text{client} \setminus z)$.

$$P_{\text{client}}(z) = a[z\,{:}\,\text{client}](s).\mu X.(\quad s\forall(y\,{:}\,\text{client} \setminus z).\{s!\,\langle y, \mathsf{Msg}\langle m\rangle\rangle\} \mid s\forall(x\,{:}\,\text{client} \setminus z).\{s?\langle x, \mathsf{Msg}(w)\rangle\}\quad); X$$

(3) Auction We now illustrate the expressiveness of our universal types when combined with instantiation of participant identities and branching session. In this session, we have three roles: the multiple buyers (here participants `alice`, `alex`, `alan`) and sellers (here `bob`, `ben`) which all connect to a single broker. This broker will then form matching pairs (x, y) of buyers and sellers who will then continue their interaction Price-Order separately.



$$G = \forall x\,{:}\,\text{buyer}.\forall y\,{:}\,\text{seller}.\text{broker}{\to}x\{\quad \mathsf{Match}\langle y\rangle.x{\to}y\langle\mathsf{Notify}\rangle.y{\to}x\langle\mathsf{Price}\rangle.x{\to}y \quad \mathsf{Quit}\langle y\rangle.\ x{\to}y\ \langle\mathsf{Stop}\rangle\}; \text{end}$$

The quantifications $\forall x\,{:}\,\text{buyer}.\forall y\,{:}\,\text{seller}$ specify that every possible association between buyers and sellers is considered by the broker when he makes his choices. For each pair (x, y) of buyer and seller, the broker selects to send to x either a message $\mathsf{Match}\langle y\rangle$ if he has found y to be a match for x, or a message $\mathsf{Quit}\langle y\rangle$ otherwise. If the message Match was sent, x notifies y and the interaction Price-Order proceeds. In the other branch, x needs to warn y by the message Stop that the brocker chose the second branch.

For this example, we just write a process for a buyer:

$$P_{\text{buyer}}(z) = a[z\,{:}\,\text{buyer}](s).\ s\forall(y\,{:}\,\text{seller}).\{s?\langle\text{broker}, \{\quad \mathsf{Match}\langle y\rangle.s!\,\langle y, \mathsf{Notify}\rangle.s?\langle y, \mathsf{Price}\rangle.s! \quad \mathsf{Quit}\langle y\rangle.s!\,\langle y, \mathsf{Stop}\rangle\}\rangle\}; \mathtt{quit}\langle s\rangle$$

From the above process, we can see the importance of the communication of the participant identity y with the messages Match and Quit. The adjonction of y to the messages is necessary for x to know to which y to send the Notify message. Note that the y in $\mathsf{Match}\langle y\rangle$ is not a regular payload as all the sellers y are already known by x: at reception, x matches his known y against the one coming along Match or Quit.

This example presents a session where all participants are required to leave the session (through the expression $\mathtt{quit}\langle s\rangle$) at the end of their interaction. Since late joiners always start at the beginning of the session, they cannot safely interact with the participants that have already proceeded into the interaction. To guarantee

[2] If we want our global type to include those self-sent message, it can be done explicitly by writing a global type: $\mu\mathbf{x}.(\forall x\,{:}\,\text{client}.(x{\to}x\langle\mathsf{Msg}\rangle \mid \forall y\,{:}\,\text{client}.x{\to}y\langle\mathsf{Msg}\rangle)); \mathbf{x}$.

progress, we require that late joiners wait for the current participants to leave before joining themselves and beginning their actions. To provide consistent synchronisation, we introduce a *multiparty locking mechanism* to protect the global session executions.

Main contributions

- (§ **2**) A new role-based multiparty session type framework where participants can play several roles in a session. Its semantics allows participants to dynamically join and leave a running session, and create new parallel sessions.
- (§ **3,** § **4**) Introduction of the universal type, along with the explicit parallel composition, and a type system that provides subject reduction (Theorem 4.2) and type safety (Corollary 4.3: no type error for values and labels). The end-point projection and the well-formedness conditions of global types deal with the subtle interplay between universal quantifiers, parallel compositions, branching and instantiations of participant identities.
- (§ **5**) A semantics and type system with a simple locking mechanism by which communication safety (Theorem 5.3: every receiver has a corresponding sender with the right type), progress (Theorem 5.6: processes in a single multiparty session always progress) and join progress (Theorem 5.8: late joiners can always join to an existing session and progress) are established.
- (§ **5.5**) Practical implementation techniques used in our prototype extension of ML.

The proofs, detailed definitions, more examples and the prototype implementation are available from [3].

2. Multirole session calculus

We describe here an extension of the multiparty session calculus presented in [4]. Our new system handles roles and allows programs to participate in protocols that include multiple parallel interactions and dynamic role instantiation.

$$
\begin{array}{llll|llll}
u & ::= & x \mid a \mid b \mid \ldots & \text{Shared channel} & c & ::= & s[p] \mid y & \text{Session channel} \\
p & ::= & \mathtt{p}\!:\!r \mid x\!:\!r & \text{Participant with role} & e & ::= & v \mid x \mid \ldots & \text{Expression} \\
\vec{p} & ::= & \mathtt{p}::\vec{p} \mid x::\vec{p} \mid \varepsilon & \text{Participant list} & v & ::= & a \mid s[\mathtt{p}:r] \mid \mathsf{true} \mid \ldots & \text{Values}
\end{array}
$$

$$
\begin{array}{lll|ll}
P ::= & & \text{Processes} & & \\
 & \mid u\langle G\rangle & \text{Session Init} & \mid \text{if } e \text{ then } P \text{ else } P & \text{Conditional} \\
 & \mid u[p](y).P & \text{Join} & \mid \mu X.P \mid X \mid \mathbf{0} & \text{Recursion} \\
 & \mid \mathtt{quit}\langle c\rangle & \text{Quit} & \mid (\nu\ a\!:\!G)P & \text{Restriction} \\
 & \mid c!\langle p, l\langle\vec{p}\rangle(e)\rangle & \text{Send} & \mid (\nu\ s)P & \text{Session restriction} \\
 & \mid c?\langle p, \{l_i\langle\vec{p}_i\rangle(x_i).P_i\}_{i\in I}\rangle & \text{Receive} & \mid s\!:\!h & \text{Message buffer} \\
 & \mid c\forall(x\!:\!r\setminus\vec{p}).\{P\} & \text{Poll} & \mid a\langle s\rangle[\mathtt{R}] & \text{Session registry} \\
 & \mid P \mid P & \text{Parallel} & \mathtt{R} ::= r_1\!:\!\mathtt{P}_1, \ldots, r_n\!:\!\mathtt{P}_n & \text{Role set} \\
 & \mid P;P & \text{Sequential} & h ::= \varepsilon \mid h\cdot(\mathtt{p}_0\!:\!r_0, \mathtt{p}_1\!:\!r_1, l\langle\vec{\mathtt{p}}\rangle(v)) & \text{Buffer}
\end{array}
$$

Figure 1. Multirole session calculus

Syntax We give in figure 1 the syntax of the processes of our session variant of the π-calculus.

A session is always initialised by a process of the form $u\langle G\rangle$ where G is a global type (formally defined in § 3). Session initialisation attributes a particular global interaction pattern G to a shared channel u. Once the session has been initialised on channel u, participants can join with $u[p](y).P$. p designates a participant identity $\mathtt{p}$ or x associated with a particular role name r. Joining binds the variable y with the session channel that this particular participant can use when he plays the role r. Leaving the session is done by $\mathtt{quit}\langle c\rangle$, where c is the session channel corresponding to the participant and role.

The asynchronous emission $c!\langle p, l\langle\vec{p}\rangle(e)\rangle$ allows to send to p a value e labelled by a constant l and participant names $\vec{p}$. The reception $c?\langle p, \{l_i\langle\vec{p}_i\rangle(x_i).P_i\}_{i\in I}\rangle$ expects from p a message with a label among the $\{l_i\}_{i\in I}$ with participants $\vec{p}_i$. The message payload is then received in variable x_i, which binds in P_i. Messages are always labelled. The list of participants $\vec{p}_i$ enriches the label l_i in order for the receiver to be able to disambiguate messages that have the same sender and label, but different continuations.

The polling operation $c\forall(x\!:\!r\setminus\vec{p}).\{P\}$ is the main way to interact with the participants that instantiate a given role: P is replicated for each participant x playing role r, with the exception of the participants mentioned in $\vec{p}$.

Parallel and sequential composition are standard, as are the conditional and recursion. The creation of a shared rendez-vous name is done by $(\nu\ a\!:\!G)P$. This fresh name can then be used as a reference for future instances of a session specified by G.

Once a session is running, our semantics uses some artifacts that are not directly accessible to the programmer. First, session instances are represented by session restriction $(\nu\ s)P$. Second, the message buffer $s\!:\!h$ stores the messages in transit for the session s. Last, the session registry $a\langle s\rangle[\mathtt{R}]$ records the current association between participants and roles in the running session s.

For simplicity, we write $c?\langle p, l\langle\vec{p}\rangle(x_i)\rangle.P$ if there is a unique branch. Similarly, we omit the empty list of participant ($\langle\varepsilon\rangle$) and unit payloads (e.g. $c!\langle p, l\rangle$). We also do not write $\mathbf{0}$, and roles r (e.g. in $x\!:\!r$) if they are clear from the context.

We use syntactic sugar for the special roles that cannot be multiply instantiated. Their participant name ($\mathtt{p}$ or x) does not have to be explicitly mentioned. Polling is done implicitly for these roles: the mention of their role r is sufficient and unambiguous. In the Map/Reduce example from § 1, server is such a role.

We call a process which does not contain free variables and runtime syntax *initial*.

Semantics Figure 2 lists the reduction rules. The $\lfloor$Init$\rfloor$ rule proceeds to a session initialisation by reducing $a\langle G\rangle$. It creates a fresh session channel s and two processes. First, the session registry $a\langle s\rangle[\mathtt{R}]$ is an entity that centralises the association between participants and roles in the particular instance s of a session. Initially, $\mathtt{R}$ does not record any participant for any of the roles of G. The second process is the session's message buffer $s\!:\!\varepsilon$, which is also initially empty.

The rule $\lfloor$Join$\rfloor$ governs the registration of a participant to a running session. The participant asks with $a[\mathtt{p}:r](y).P$ to join the session advertised on channel a and specifies his identity $\mathtt{p}$ and which role r he wants to play. This information is added to the session registry $a\langle s\rangle[\mathtt{R}\cdot r:\mathtt{P}\uplus\{\mathtt{p}\}]$ and the session channel $s[\mathtt{p}:r]$ is communicated. The rule $\lfloor$Quit$\rfloor$ manages the departure of a participant from a session: $\mathtt{quit}\langle s[\mathtt{p}:r]\rangle$ forces the deletion of $\mathtt{p}:r$ from the registry.

The rule $\lfloor$Send$\rfloor$ describes asynchronous sending, which appends its labelled message to the buffer $s\!:\!h$. In rule $\lfloor$Recv$\rfloor$, the reception takes from the session buffer the first message $(\mathtt{p}'\!:\!r', \mathtt{p}\!:\!r, l_k\langle\vec{\mathtt{p}}_k\rangle(v))$ that has a proper address, label and participant list and selects the matching continuation P_k. The rule $\lfloor$Poll$\rfloor$ details the reduction of the polling process $s[\mathtt{p}\!:\!r']\forall(x\!:\!r\setminus\vec{\mathtt{p}}).\{P\}$. The set of participants $\{\mathtt{p}_1,\ldots,\mathtt{p}_k\}$ that play role r (once the ones in $\vec{\mathtt{p}}$ are removed) is received from the session registry and the process P is forked accordingly, with x substituted by each. The rules $\lfloor$True$\rfloor$ and $\lfloor$False$\rfloor$ are standard.

The reduction is defined modulo the standard structural equivalence $\equiv$. We just mention here the session garbage collection rule $(\nu\ a,s)(a\langle s\rangle[\mathtt{R}] \mid s:\varepsilon) \equiv \mathbf{0}$ (when $\forall r_i \in G, \mathtt{R}(r_i) = \varnothing$) and the permutation rule $s:(q,p,l\langle\vec{p}_1\rangle(v))\cdot(q',p',l'\langle\vec{p}_2\rangle(v'))\cdot h \equiv s:(q',p',l'\langle\vec{p}_2\rangle(v'))\cdot(q,p,l\langle\vec{p}_1\rangle(v))\cdot h$ which allows to put forward

$$
\begin{array}{rcll}
a\langle G\rangle & \rightarrow & (\nu\ s)(a\langle s\rangle[\mathtt{R}] \mid s:\varepsilon) \qquad (\forall r_i \in G, \mathtt{R}(r_i) = \varnothing) & \lfloor\textsc{Init}\rfloor \\
a[\mathtt{p}:r](y).P \mid a\langle s\rangle[\mathtt{R}\cdot r:\mathtt{P}] & \rightarrow & P\{s[\mathtt{p}:r]/y\} \mid a\langle s\rangle[\mathtt{R}\cdot r:\mathtt{P} \uplus \{\mathtt{p}\}] & \lfloor\textsc{Join}\rfloor \\
\mathtt{quit}\langle s[\mathtt{p}:r]\rangle \mid a\langle s\rangle[\mathtt{R}\cdot r:\mathtt{P}] & \rightarrow & a\langle s\rangle[\mathtt{R}\cdot r:\mathtt{P}\setminus \mathtt{p}] & \lfloor\textsc{Quit}\rfloor \\
s[\mathtt{p}:r]!\langle \mathtt{p}':r', l\langle\vec{\mathtt{p}}\rangle(v)\rangle \mid a\langle s\rangle[\mathtt{R}] \mid s:h & \rightarrow & a\langle s\rangle[\mathtt{R}] \mid s:h\cdot(\mathtt{p}:r,\ \mathtt{p}':r',\ l\langle\vec{\mathtt{p}}\rangle(v)) & \\
 & & \qquad\qquad (\mathtt{p}\in\mathtt{R}(r) \wedge \mathtt{p}'\in\mathtt{R}(r')) & \lfloor\textsc{Send}\rfloor \\
s[\mathtt{p}:r]?\langle \mathtt{p}':r', \{l_i\langle\vec{\mathtt{p}}_i\rangle(x_i).P_i\}_{i\in I}\rangle & & & \\
\mid a\langle s\rangle[\mathtt{R}] \mid s:(\mathtt{p}':r',\ \mathtt{p}:r,\ l_k\langle\vec{\mathtt{p}}_k\rangle(v))\cdot h & \rightarrow & P_k\{v/x_k\} \mid a\langle s\rangle[\mathtt{R}] \mid s:h \quad (\mathtt{p}\in\mathtt{R}(r) \wedge k\in I) & \lfloor\textsc{Recv}\rfloor \\
s[\mathtt{p}:r']\forall(x:r\setminus\vec{\mathtt{p}}).\{P\} \mid a\langle s\rangle[\mathtt{R}] & \rightarrow & P\{\mathtt{p}_1/x\} \mid \ldots \mid P\{\mathtt{p}_k/x\} \mid a\langle s\rangle[\mathtt{R}] & \\
 & & \qquad (\mathtt{R}(r)\setminus\vec{\mathtt{p}} = \{\mathtt{p}_1,..,\mathtt{p}_k\} \wedge \mathtt{p}\in\mathtt{R}(r')) & \lfloor\textsc{Poll}\rfloor
\end{array}
$$

$$
\text{if true then } P \text{ else } Q \rightarrow P \quad \lfloor\textsc{IfT}\rfloor \qquad\qquad \text{if false then } P \text{ else } Q \rightarrow Q \quad \lfloor\textsc{IfF}\rfloor
$$

$$
\frac{P \mid Q \rightarrow P' \mid Q'}{\mathscr{E}[P] \mid Q \rightarrow \mathscr{E}[P'] \mid Q'}\lfloor\textsc{Par}\rfloor \qquad \frac{P \rightarrow P'}{\mathscr{E}[P] \rightarrow \mathscr{E}[P']}\lfloor\textsc{Ctx}\rfloor \qquad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q}\lfloor\textsc{Cong}\rfloor
$$

$$
\begin{array}{rcl}
\mathscr{E} & ::= & [\_] \mid \mathscr{E} \mid P \mid \mathscr{E};P \mid (\nu\ a)\mathscr{E} \mid (\nu\ s)\mathscr{E} \\
 & \mid & s[\mathtt{p}:r]!\langle \mathtt{p}':r', l\langle\vec{\mathtt{p}}\rangle(\mathscr{E})\rangle \mid \text{if } \mathscr{E} \text{ then } P \text{ else } P \mid \mathscr{E}\wedge e \mid v \wedge \mathscr{E} \mid \ldots
\end{array}
$$

Figure 2. Reduction rules for the multirole session calculus

in the session buffers the messages that have different senders, recipients, labels or participants lists.

Reduction example We take the process $P_{\text{client}}(z)$ from the peer-to-peer chat mentioned in the introduction (§ 1(2)). Figure 3 gives reduction steps of a situation where we have two client processes $P_{\text{client}}(\mathtt{p}_1)$ and $P_{\text{client}}(\mathtt{p}_2)$ that want to interact on session channel a. We call $Q(z)$ the process $\mu X.(s[z]\forall(y:\text{client}\setminus z).\{s[z]!\langle y, \mathsf{Msg}\langle m\rangle\rangle\} \mid s[z]\forall(x:\text{client}\setminus z).\{s[z]?\langle x, \mathsf{Msg}(w)\rangle\}); X$ and abbreviate the registry $a\langle s\rangle[\text{client}:\{\mathtt{p}_1,\mathtt{p}_2\}]$ by R.

$$
\begin{array}{lcl}
 & & (\nu\ a)(a\langle \mathrm{G}\rangle \mid P(\mathtt{p}_1) \mid P(\mathtt{p}_2)) \\
\lfloor\textsc{Init}\rfloor & \rightarrow & (\nu\ a)((\nu\ s)(a\langle s\rangle[\text{client}:\varnothing] \mid s:\varepsilon) \mid P(\mathtt{p}_1) \mid P(\mathtt{p}_2))) \\
\lfloor\textsc{Join}\rfloor & \rightarrow & (\nu\ a,s)(a\langle s\rangle[\text{client}:\{\mathtt{p}_1\}] \mid s:\varepsilon \mid Q(\mathtt{p}_1) \mid P(\mathtt{p}_2)) \\
\lfloor\textsc{Join}\rfloor & \rightarrow & (\nu\ a,s)(a\langle s\rangle[\text{client}:\{\mathtt{p}_1,\mathtt{p}_2\}] \mid s:\varepsilon \mid Q(\mathtt{p}_1) \mid Q(\mathtt{p}_2)) \\
\lfloor\textsc{Poll}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:\varepsilon \mid Q(\mathtt{p}_2) \mid \\
 & & \qquad (s[\mathtt{p}_1]!\langle \mathtt{p}_2, \mathsf{Msg}\langle m\rangle\rangle \mid s[\mathtt{p}_1]\forall(x:\text{client}\setminus\mathtt{p}_1).\{s[\mathtt{p}_1]?\langle x,\mathsf{Msg}(w)\rangle\}\});Q(\mathtt{p}_1)) \\
\lfloor\textsc{Send}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_1,\mathtt{p}_2,\mathsf{Msg}\langle m\rangle) \mid Q(\mathtt{p}_2) \mid (s[\mathtt{p}_1]\forall(x:\text{client}\setminus\mathtt{p}_1).\{s[\mathtt{p}_1]?\langle x,\mathsf{Msg}(w)\rangle\}\};Q(\mathtt{p}_1)) \\
\lfloor\textsc{Poll}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_1,\mathtt{p}_2,\mathsf{Msg}\langle m\rangle)) \mid Q(\mathtt{p}_2) \mid s[\mathtt{p}_1]?\langle \mathtt{p}_2,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_1)) \\
\lfloor\textsc{Poll}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_1,\mathtt{p}_2,\mathsf{Msg}\langle m\rangle) \mid s[\mathtt{p}_1]?\langle \mathtt{p}_2,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_1) \mid \\
 & & \qquad (s[\mathtt{p}_2]!\langle \mathtt{p}_1, \mathsf{Msg}\langle m\rangle\rangle \mid s[\mathtt{p}_2]\forall(x:\text{client}\setminus\mathtt{p}_2).\{s[\mathtt{p}_2]?\langle x,\mathsf{Msg}(w)\rangle\}\});Q(\mathtt{p}_2)) \\
\lfloor\textsc{Send}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_1,\mathtt{p}_2,\mathsf{Msg}\langle m\rangle)\cdot(\mathtt{p}_2,\mathtt{p}_1,\mathsf{Msg}\langle m\rangle) \mid \\
 & & \qquad s[\mathtt{p}_1]?\langle \mathtt{p}_2,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_1) \mid s[\mathtt{p}_2]\forall(x:\text{client}\setminus\mathtt{p}_2).\{s[\mathtt{p}_2]?\langle x,\mathsf{Msg}(w)\rangle\}\};Q(\mathtt{p}_2)) \\
\lfloor\textsc{Poll}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_1,\mathtt{p}_2,\mathsf{Msg}\langle m\rangle)\cdot(\mathtt{p}_2,\mathtt{p}_1,\mathsf{Msg}\langle m\rangle) \mid \\
 & & \qquad s[\mathtt{p}_1]?\langle \mathtt{p}_2,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_1) \mid s[\mathtt{p}_2]?\langle \mathtt{p}_1,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_2)) \\
\lfloor\textsc{Recv}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:(\mathtt{p}_2,\mathtt{p}_1,\mathsf{Msg}\langle m\rangle) \mid s[\mathtt{p}_1]?\langle \mathtt{p}_2,\mathsf{Msg}(w)\rangle;Q(\mathtt{p}_1) \mid Q(\mathtt{p}_2)) \\
\lfloor\textsc{Recv}\rfloor & \rightarrow & (\nu\ a,s)(R \mid s:\varepsilon \mid Q(\mathtt{p}_1) \mid Q(\mathtt{p}_2))
\end{array}
$$

Figure 3. Reduction for the Peer-to-peer chat example

3. Multirole session types

In this section, we present the multirole session types that we use to specify the communication patterns that are to be enforced. We start with the definition of global and local types and follow with projection and well-formedness properties.

3.1 Global and local types

Global types G describe role-based global scenarios between multiple participants as a type signature. When a participant agrees with a global type G, his behaviour is defined by a local protocol (called *local type* T_i) is generated by the projection of G to the role he wants to play. If each of the local programs $P_1,\ldots,P_n$ can be type-checked against the corresponding projected local types $T_1,..,T_k$, then they

$$
\begin{array}{rlll}
G ::= & & & \text{Global types} \\
 & \mid & p \rightarrow p'\{l_i\langle\vec{p}_i\rangle(U_i).G_i\}_{i\in I} & \text{Labelled messages} \\
 & \mid & \forall x:r\setminus\vec{p}.G & \text{Universal quantification} \\
 & \mid & G \mid G' \ \mid\ G;G' & \text{Parallel, sequential} \\
 & \mid & \mu\mathbf{x}.G \ \mid\ \mathbf{x} & \text{Recursion, variable} \\
 & \mid & \varepsilon \ \mid\ \text{end} & \text{Inaction, End} \\
T ::= & & & \text{Local types} \\
 & \mid & !\langle p, \{l_i\langle\vec{p}_i\rangle(U_i).T_i\}_{i\in I}\rangle & \text{Selection} \\
 & \mid & ?\langle p, \{l_i\langle\vec{p}_i\rangle(U_i).T_i\}_{i\in I}\rangle & \text{Branching} \\
 & \mid & \forall x:r\setminus\vec{p}.T & \text{Universal quantification} \\
 & \mid & T \mid T' \ \mid\ T;T' & \text{Parallel, sequential} \\
 & \mid & \mu\mathbf{x}.G \ \mid\ \mathbf{x} \ \mid\ \varepsilon \ \mid\ \text{end} & \text{Recursion, inaction, end} \\
U ::= & & S \ \mid\ T & \text{Message types} \\
S ::= & & \langle G\rangle \ \mid\ \text{bool} \ \mid\ \text{unit} \ \mid\ \ldots & \text{Sorts}
\end{array}
$$

Figure 4. Global and local types

are automatically guaranteed to interact properly, following the intended scenario. The grammar of global types $(G,G',\ldots)$ and local types $(T,T',\ldots)$ is given in figure 4. There are four key extensions from the standard multiparty session types [4]: (1) association of each participant to a role; (2) universal quantifiers to bind participants identities; (3) parallel compositions for local types; and (4) labels that can be extended by lists of participants.

In the global types $(G,G',\ldots)$, a global interaction can be a labelled message exchange $(p\rightarrow p'\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.G_i\}_{i\in I})$, where p,p' denote the sending and receiving participants with roles (recall that p denotes either $\mathtt{p}:r$ or $x:r$), $\vec{p}_i$ is a list of participants, U_i is the payload type of the message and G_i the interaction that follows the choice of label l_i (I is a finite set of integers). Value types S include shared channel types $\langle G\rangle$ or base types (bool, unit ,...). Message types U are either value types S or local types T (which correspond to the behaviour of one of the session participants) for delegation.

Parallel composition is written as $G \mid G'$, and $G;G'$ denotes sequential composition. $\mu\mathbf{x}.G$ is a recursive type where type variable $\mathbf{x}$ is guarded in the standard way (they only appear under some prefix). Inaction ε marks the absence of communication, while end denotes the end of the session for all roles. The *universal quantification* is written $\forall x:r\setminus\vec{p}.G$ where the participants of role r bind free occurrences x in G. It corresponds to the operational semantics of $s[\mathtt{p}:r']\forall(x:r\setminus\vec{p}).\{P\}$ (see § 2), i.e. a parallel composition

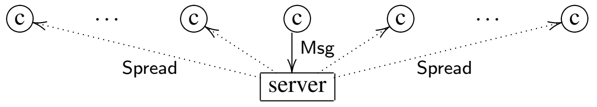
$G\{\mathtt{p}_1/x\} \mid ... \mid G\{\mathtt{p}_k/x\}$ for some list of participants $\{\mathtt{p}_1,...,\mathtt{p}_k\}$ playing the role r (which is decided at runtime), from which the list of participants $\vec{p}$ has been excluded.

In local types T, selection expresses the transmission to p of a label l_i taken from a set $\{l_i\}_{i\in I}$ with a list of participants $\vec{p}_i$ and a message type U_i, followed by T_i. Branching is its dual counterpart. The other local types are similar to their global versions.

We consider global and local types modulo the following equalities. For local types, we define: $(T \mid \varepsilon) = (\varepsilon \mid T) = (\varepsilon;T) = T$, $(T \mid \mathsf{end}) = (\mathsf{end} \mid T) = T$, $(T \mid T');\mathsf{end} = (T;\mathsf{end} \mid T';\mathsf{end})$ and $!\langle p,\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle;\mathsf{end} = !\langle p,\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.T_i;\mathsf{end}\}_{i\in I}\rangle$. Similar equalities are applied to global types.

We also use similar abbreviations for global and local types as the ones for processes (mentioned in § 2). In particular, we write $p{\rightarrow}q\ l\langle\vec{p}'\rangle\langle U\rangle;G$ or $!\langle p,l\langle\vec{q}\rangle\langle U\rangle\rangle;T$ when there is a single branch. We omit obvious role annotations, empty participant lists and unit payload types. end is also often eluded.

EXAMPLE 3.1 (Global types). For clarity of the semantics of global session types, we give here several variations on an additional example. We imagine a chat protocol (similar in spirit to the peer-to-peer chat session) where the clients must interact through a single server. We have thus two roles: the unique server and the multiple clients. Each client's behaviour is to send a message to the server who will then broadcast it to all the others. In the following picture, we only represent the Msg that one client sends to the server and that is followed by the server broadcasting its content (in message Spread) to all the other clients.



The global type for this session relies on the sequentiality that links each Msg with its following Spread. We write it as:

$$G_1 = \mu\mathbf{x}.(\forall x{:}\mathrm{client}.(x{\rightarrow}\mathrm{server}\langle\mathsf{Msg}\rangle.\forall y{:}\mathrm{client}\setminus x.(\mathrm{server}{\rightarrow}y\langle\mathsf{Spread}\rangle)));\mathbf{x}$$

It starts with a quantification over all clients x. Upon reception by the server of a message from x, the global type specifies that Spread should be sent to all the other clients: $\forall y:\mathrm{client}\setminus x.\mathrm{server}{\rightarrow}y\langle\mathsf{Spread}\rangle$.

An alternate chat server could be one where the server collects all incoming messages and then sends a digest to all clients. In that case, the global type would be written:

$$G_2 = \mu\mathbf{x}.(\forall x{:}\mathrm{client}.x{\rightarrow}\mathrm{server}\langle\mathsf{Msg}\rangle);(\forall y{:}\mathrm{client}.\mathrm{server}{\rightarrow}y\langle\mathsf{Spread}\rangle);\mathbf{x}$$

The central synchronisation between the two quantified types is important in our model. The semantics is radically different if this synchronisation is removed.

$$G_3 = \mu\mathbf{x}.(\forall x{:}\mathrm{client}.x{\rightarrow}\mathrm{server}\langle\mathsf{Msg}\rangle;\mathrm{server}{\rightarrow}x\langle\mathsf{Spread}\rangle);\mathbf{x}$$

The global type G_3 means that, independently for each client, the server first collects a message Msg and then immediately sends back to this same client a message Spread.

3.2 Projection from multirole global types to local types

We now define the projection operation, which, for any participant z playing a role r in a session G, computes the local type it has to conform to. We say *an end-point projection of G onto z:r*, written $G \uparrow z{:}r$, is the local type that the participant z should respect to play the role r in session G.

As mentioned in § 1, the main difficulty lies in the projection of the quantifiers. Let us first consider informally the global type $\forall x:r.G$. This global type has the same semantics as $G\{\mathtt{p}_1/x\} \mid ... \mid G\{\mathtt{p}_k/x\}$ for some $\mathtt{p}_1,...,\mathtt{p}_k$ playing the role r. If we write the projection of $\forall x{:}r.G$ for a participant $\mathtt{p}_i$ playing role r (written as $\forall x{:}r.G \uparrow \mathtt{p}_i{:}r$), we can single out the instance corresponding to $\mathtt{p}_i$:

$$(G\{\mathtt{p}_1/x\}\uparrow\mathtt{p}_i{:}r) \mid ... \mid (G\{\mathtt{p}_k/x\}\uparrow\mathtt{p}_i{:}r) = (G\{\mathtt{p}_i/x\}\uparrow\mathtt{p}_i{:}r) \mid \forall x{:}r\setminus\mathtt{p}_i.(G\uparrow\mathtt{p}_i{:}r)$$

Based on this intuition behind the projection of quantifiers, we give the projection definition in figure 5. Projection is role-based, i.e. for each role r of a session G, a local type $T = G \uparrow p$ is computed with $p = z{:}r$. The case $p = \mathtt{p}{:}r$ is defined by replacing z by $\mathtt{p}$.

$$\begin{array}{rcll}
p{\rightarrow}p'\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle:G_i\}_{i\in I}\uparrow p & = & !\langle p',\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.G_i\uparrow p\}_{i\in I}\rangle & \\
p'{\rightarrow}p\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle:G_i\}_{i\in I}\uparrow p & = & ?\langle p',\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.G_i\uparrow p\}_{i\in I}\rangle & \\
p{\rightarrow}p\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle:G_i\}_{i\in I}\uparrow p & = & !\langle p,\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.?\langle p,l_i\langle\vec{p}_i\rangle\langle U_i\rangle.G_i\uparrow p\rangle\}_{i\in I}\rangle & \\
p_1{\rightarrow}p_2\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.G_i\}_{i\in I}\uparrow p & = & \bigsqcup_{i\in I}\{G_i\uparrow p\} & \\
(\forall x{:}r\setminus\vec{p}.G)\uparrow z{:}r & = & G\{z/x\}\uparrow z{:}r \mid \forall x{:}r\setminus z{::}\vec{p}.(G\uparrow z{:}r) & (z\notin\vec{p}) \\
(\forall x{:}r\setminus\vec{p}.G)\uparrow p & = & \forall x{:}r\setminus\vec{p}.(G\uparrow p) & \text{(other}
\end{array}$$

$$(G \mid G)\uparrow p = (G\uparrow p \mid G\uparrow p) \qquad (G;G)\uparrow p = (G\uparrow p;G\uparrow p)$$

$$\mu\mathbf{x}.G\uparrow p = \mu\mathbf{x}.(G\uparrow p) \qquad \mathbf{x}\uparrow p = \mathbf{x} \qquad \varepsilon\uparrow p = \varepsilon \qquad \mathsf{end}\uparrow p = \mathsf{end}$$

Figure 5. Projection

The projection of communication leads to a case analysis: if the participant projected to is the sender, then the projection is a selection sent to p'; if p is the receiver then the projection is an input from p'; if participant p is both sender and receiver then the projection is an output followed by an input; otherwise, the communication is not observed locally and is skipped. The operator $\sqcup$ is then merging the different remote branches (this operation was introduced in [35, § 4]). Roughly speaking, it makes sure that the locally observable behaviour are either independent of the remotely chosen branch or can be properly identified through the label. It is defined by $T \sqcup T = T$ and the following equality:

$$\begin{array}{l}
?\langle p,\{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle \sqcup ?\langle p,\{l_j\langle\vec{p}_j'\rangle\langle U_j'\rangle.T_j'\}_{j\in J}\rangle = \\
\quad ?\langle p,\{l_k\langle\vec{p}_k\rangle\langle U_k\rangle.T_k\sqcup T_k'\}_{k\in I\cap J}\cup\{l_k\langle\vec{p}_k\rangle\langle U_k\rangle.T_k\}_{k\in I\setminus J}\cup\{l_k\langle\vec{p}_k'\rangle\langle U_k'\rangle.T_k\}_{k\in J\setminus} \\
\qquad\qquad \text{when } \forall k\in I\cap J, \vec{p}_k = \vec{p}_k' \wedge U_k = U_k'
\end{array}$$

Note that the merging operation may not return a result if the session uses labels ambiguously. An example can be found in § 3.3.

Finally, the most critical rules are the projection of a quantified global type $(\forall x:r\setminus\vec{p}.G)\uparrow p$. The first rule applies when the quantification acts on the same role r as the projection, and when p is not in the exclusion list $\vec{p}$. In that case, as explained above, the local type is the parallel composition of G where $x:r$ is substituted by p, projected for p, and a quantification excluding p. The second rule sees the projection acting homomorphically through the quantification on a different role, or if p is in $\vec{p}$. Other rules are homomorphic as well. We say that G is *projectable* if G can be projected (i.e. projection gives a result) for each of its roles.

EXAMPLE 3.2 (Projection). **(1) Peer-to-peer chat example.** We give an example of projection for the peer-to-peer chat session from § 1, which features nested quantifiers. The local type $T(z{:}\mathrm{client})$ is calculated in the following way (p is $z{:}\mathrm{client}$):

$$\begin{array}{rl}
& (\mu\mathbf{x}.(\forall x{:}\mathrm{client}.\forall y{:}\mathrm{client}\setminus x.x{\rightarrow}y\langle\mathsf{Msg}\rangle);\mathbf{x})\uparrow z{:}\mathrm{client} \\
= & \mu\mathbf{x}.((\forall x{:}\mathrm{client}.\forall y{:}\mathrm{client}\setminus x.x{\rightarrow}y\langle\mathsf{Msg}\rangle)\uparrow z{:}\mathrm{client});\mathbf{x} \\
= & \mu\mathbf{x}.((\forall y{:}\mathrm{client}\setminus z.z{\rightarrow}y\langle\mathsf{Msg}\rangle)\uparrow z{:}\mathrm{client} \mid \forall x{:}\mathrm{client}\setminus z.(\forall y{:}\mathrm{client}\setminus x.x{\rightarrow}y\langle\mathsf{Msg}\rangle)\uparrow z{:} \\
= & \mu\mathbf{x}.(\forall y{:}\mathrm{client}\setminus z.(z{\rightarrow}y\langle\mathsf{Msg}\rangle)\uparrow z{:}\mathrm{client} \mid \\
& \quad \forall x{:}\mathrm{client}\setminus z.((x{\rightarrow}z\langle\mathsf{Msg}\rangle)\uparrow z{:}\mathrm{client} \mid \forall y{:}\mathrm{client}\setminus z.(x{\rightarrow}y\langle\mathsf{Msg}\rangle)\uparrow z{:}\mathrm{client}));\mathbf{x} \\
= & \mu\mathbf{x}.(\forall y{:}\mathrm{client}\setminus z.!\langle y,\mathsf{Msg}\rangle \mid \forall x{:}\mathrm{client}\setminus z.(?\langle x,\mathsf{Msg}\rangle \mid \forall y{:}\mathrm{client}\setminus z.\varepsilon));\mathbf{x} \\
\equiv & \mu\mathbf{x}.(\forall y{:}\mathrm{client}\setminus z.!\langle y,\mathsf{Msg}\rangle \mid \forall x{:}\mathrm{client}\setminus z.?\langle x,\mathsf{Msg}\rangle);\mathbf{x}
\end{array}$$

(2) Chat-server from Example 3.1. We give the projections for each of the three global types. The projection of G_1 for the server and client roles gives:

$$T_1(z\!:\!\text{server}) = \mu\mathbf{x}.(\forall x\!:\!\text{client}.?\langle x\!:\!\text{client}, \mathsf{Msg}\rangle; \forall y\!:\!\text{client} \setminus x.!\,\langle y\!:\!\text{client}, \mathsf{Spread}\rangle); \mathbf{x}$$
$$T_1(z\!:\!\text{client}) = \mu\mathbf{x}.(!\,\langle \text{server}, \mathsf{Msg}\rangle \mid \forall x\!:\!\text{client} \setminus z.\{?\langle \text{server}, \mathsf{Spread}\rangle\}); \mathbf{x}$$

Note that the sequentiality between Msg and Spread is rightly present in the server's local type. The projection of G_2 results in:

$$T_2(z\!:\!\text{server}) = \mu\mathbf{x}.(\forall x\!:\!\text{client}.?\langle x\!:\!\text{client}, \mathsf{Msg}\rangle; \forall y\!:\!\text{client}.!\,\langle y\!:\!\text{client}, \mathsf{Spread}\rangle); \mathbf{x}$$
$$T_2(z\!:\!\text{client}) = \mu\mathbf{x}.(!\,\langle \text{server}, \mathsf{Msg}\rangle \mid \forall x\!:\!\text{client} \setminus z.\{?\langle \text{server}, \mathsf{Spread}\rangle\}); \mathbf{x}$$

We note that the server's local type represents a behaviour which first collects all incoming messages and then sends a digest to all clients. On the other hand, the client behavior is the same as in session G_1. The projection of G_3 is given as:

$$T_3(z\!:\!\text{server}) = \mu\mathbf{x}.(\forall x\!:\!\text{client}.?\langle x\!:\!\text{client}, \mathsf{Msg}\rangle; !\,\langle y\!:\!\text{client}, \mathsf{Spread}\rangle); \mathbf{x}$$
$$T_3(z\!:\!\text{client}) = \mu\mathbf{x}.(!\,\langle \text{server}, \mathsf{Msg}\rangle; ?\langle \text{server}, \mathsf{Spread}\rangle); \mathbf{x}$$

In the above types, for each client, the server first collects a message Msg and then immediately sends back to this client a Spread.

3.3 Well-formedness

For type-checking to work, global types need to follow a set of rules that will ensure a reliable and unambiguous session behaviour.

Syntax correctness First, we apply kinding rules [3] to construct syntactically correct types. First, every participant variable x is bound by a quantifier and that it is consistently used with role. Then we check that recursion variables do not appear under quantification or explicit parallel composition. Formally, if a global type is of the form $\forall x\!:\!r \setminus \vec{p}.G$ or $G \mid G'$, then G and G' are required no to contain any free recursion variables. This condition prevents any race condition between different iterations of the same loop. Other checks include verifying that the position of end is indeed correct.

Projectability As seen in § 3.2, projection does not always return a local type, due to the verification made when branches are merged. The merging operation verifies that each branch is properly labelled and that no local process can be confused by a label about which global branch to follow. We thus require that any global session type G should be projectable.

$$\begin{array}{lll} \times & G_1 = \text{broker}\to\text{buyer}\{ & \mathsf{Notify}.\text{buyer}\to\text{seller}\langle\mathsf{Msg}\rangle; \\ & & \quad\text{seller}\to\text{buyer}\langle\mathsf{Order}\rangle, \\ & & \mathsf{Quit}.\text{buyer}\to\text{seller}\langle\mathsf{Msg}\rangle\} \\ \surd & G_2 = \text{broker}\to\text{buyer}\{ & \mathsf{Notify}.\text{buyer}\to\text{seller}\langle\mathsf{Price}\rangle; \\ & & \quad\text{seller}\to\text{buyer}\langle\mathsf{Order}\rangle, \\ & & \mathsf{Quit}.\text{buyer}\to\text{seller}\langle\mathsf{Stop}\rangle\} \end{array}$$

The seller in G_1 cannot distinguish the two Msg sent by the buyer. In G_2, the seller knows which branch has been taken by the broker since the upper one is labelled by Price and the lower one by Stop.

Linearity The concept of linearity is introduced in [19] but, in our case, we use a relaxed version to allow flexible parallel compositions (explicit or through quantification) and branching. It makes sure that messages are always labelled in a way that prevents communication mix-ups.

To verify the linearity of a global type G, we first need to transform the quantifiers into explicit parallel compositions. To this effect, we associate to each role r of G a (big enough) list of participant names $\mathsf{p}_0, \mathsf{p}_1, \ldots$. Then, we compute for each role r the local type $T_r = G \uparrow \mathsf{p}_0 : r$ and homomorphically replace every subterm of T_r of the form[3] $\forall x\!:\!r \setminus \vec{p}.T_0$ by $T_0\{\mathsf{p}_i/x\} \mid T_0\{\mathsf{p}_j/x\}$ with $\mathsf{p}_i, \mathsf{p}_j$ the first two participant names for role r that do not appear in $\vec{p}$. This transformation is called dequantification.

[3] We leave the implicit quantifiers of the singly instantiated roles untouched.

DEFINITION 3.3 (Linearity). We say that a well-labelled global type G is *linear* if, for all roles r of G, the dequantification T'_r of $T_r = G \uparrow \mathsf{p}_0 : r$ satisfies: If $?\langle \mathsf{p}, \{l_i\langle\vec{p}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle$ and $?\langle \mathsf{p}, \{l'_j\langle\vec{p}_j'\rangle\langle U'_j\rangle.T'_j\}_{j\in J}\rangle$ are both subterms of T'_r, then, $\forall i, j \in I \times J$, $l_i = l'_j \Rightarrow (\{l_i\}_{i\in I} = \{l'_j\}_{j\in J} \wedge U_i = U'_j \wedge (\vec{p}_i = \vec{p}_j' \Rightarrow T_i = T'_j))$.

This definition checks that if two receptions exist in the local type of a role r, then either they share no label (and thus cannot be confused), or they share exactly the same set of message labels, with identical payload types and only the distinguishing lists of participants to reliably target different continuation types.

$$\begin{array}{ll} \times & G_6 = \forall x\!:\!\text{buyer}.\forall y\!:\!\text{seller}.\{\text{broker}\to x\langle\mathsf{Msg}\rangle.x\to y\langle\mathsf{Notify}\rangle\} \\ \surd & G_7 = \forall x\!:\!\text{buyer}.\forall y\!:\!\text{seller}.\{\text{broker}\to x\langle\mathsf{Msg}\langle y\rangle\rangle.x\to y\langle\mathsf{Notify}\rangle\} \end{array}$$

The dequantification of $G_6 \uparrow \mathsf{p}_0\!:\!\text{seller}$ is (buyers are $\mathsf{p}_0, \mathsf{p}_1$, sellers are $\mathsf{q}_0, \mathsf{q}_1$):

$$?\langle\text{broker}, \langle\mathsf{Msg}\rangle.!\,\langle\mathsf{q}_0, \langle\mathsf{Notify}\rangle\rangle\rangle \mid ?\langle\text{broker}, \langle\mathsf{Msg}\rangle.!\,\langle\mathsf{q}_1, \langle\mathsf{Notify}\rangle\rangle\rangle$$

These two concurrent threads have identical guards but different continuations. The dequantification of $G_7 \uparrow \mathsf{p}_0\!:\!\text{seller}$ is:

$$\begin{array}{l} ?\langle\text{broker}, \langle\mathsf{Msg}\langle\mathsf{q}_0\rangle\rangle.!\,\langle\mathsf{q}_0, \langle\mathsf{Notify}\rangle\rangle\rangle \\ \qquad \mid ?\langle\text{broker}, \langle\mathsf{Msg}\langle\mathsf{q}_1\rangle\rangle.!\,\langle\mathsf{q}_1, \langle\mathsf{Notify}\rangle\rangle\rangle \end{array}$$

In that case, the participant identity $\langle y\rangle$ is added to the label Msg and is able to disambiguate the concurrent receptions.

Well-formedness We now give the formal version of the well-formedness condition. Note that these conditions are decidable.

DEFINITION 3.4 (Well-formed global types). *We say that a global type G is* well-formed *if the following conditions hold:*

1. *(Syntactically correct) G is syntactically correct [3].*
2. *(Projectability) $G \uparrow z\!:\!r$ is defined for each role r of G.*
3. *(Linearity) G is linear (definition 3.3).*

EXAMPLE 3.5 (Well-formedness). We test the well-formedness of the auction example from § 1 (the numbers below correspond to the well-formedness conditions). Recall the global type G]. The syntax correctness (1) is checked easily: there is no recursion, participant variables are bound and used for a unique role, and end is well-positioned. G is projectable (2) since the two branches (Match, Quit) do not forget to use different labels (Notify, Stop) to propagate to the seller y the choice that the broker makes.

Concerning linearity (3), the potential problem is in the first message: when a buyer x receives a message Match or Quit from the broker, x should know which parallel instance it concerns among the ones the quantification $\forall y\!:\!\text{seller}$ creates. We only give below the verification details for the buyer. With buyers $\mathsf{p}_0, \mathsf{p}_1$ and sellers $\mathsf{q}_0, \mathsf{q}_1$, the result of the dequantification of $G \uparrow \mathsf{p}_0\!:\!\text{buyer}$ is:

$$\begin{array}{ll} & ?\langle\text{broker}, \{\mathsf{Match}\langle\mathsf{q}_0\rangle.!\,\langle\mathsf{q}_0, \langle\mathsf{Notify}\rangle.?\langle\mathsf{q}_0, \langle\mathsf{Price}\rangle.!\,\langle\mathsf{q}_0, \langle\mathsf{Order}\rangle\rangle\rangle\rangle, \\ & \qquad\quad \mathsf{Quit}\langle\mathsf{q}_0\rangle.!\,\langle\mathsf{q}_0, \langle\mathsf{Stop}\rangle\rangle\}\rangle \\ \mid & ?\langle\text{broker}, \{\mathsf{Match}\langle\mathsf{q}_1\rangle.!\,\langle\mathsf{q}_1, \langle\mathsf{Notify}\rangle.?\langle\mathsf{q}_1, \langle\mathsf{Price}\rangle.!\,\langle\mathsf{q}_1, \langle\mathsf{Order}\rangle\rangle\rangle\rangle, \\ & \qquad\quad \mathsf{Quit}\langle\mathsf{q}_1\rangle.!\,\langle\mathsf{q}_1, \langle\mathsf{Stop}\rangle\rangle\}\rangle \end{array}$$

For $G \uparrow \mathsf{q}_0\!:\!\text{seller}$, the dequantification result gives:

$$\begin{array}{ll} & ?\langle\mathsf{p}_0, \{\mathsf{Notify}.!\,\langle\mathsf{p}_0, \langle\mathsf{Price}\rangle.?\langle\mathsf{p}_0, \langle\mathsf{Order}\rangle\rangle\rangle, \mathsf{Stop}\}\rangle \\ \mid & ?\langle\mathsf{p}_1, \{\mathsf{Notify}.!\,\langle\mathsf{p}_1, \langle\mathsf{Price}\rangle.?\langle\mathsf{p}_1, \langle\mathsf{Order}\rangle\rangle\rangle, \mathsf{Stop}\}\rangle \end{array}$$

We check linearity by looking at the different occurrences of the same label (for example Match in $G \uparrow \mathsf{p}_0\!:\!\text{buyer}$) being received from the same participant (e.g. broker): we verify that the list of participant identities are different whenever the continuations are different. Linearity is thus only achieved here thanks to the communication of the disambiguating y in messages Match and Quit, as it can be seen in the buyer's case. The seller's and broker's (here omitted) linearity verifications are trivial.

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \text{true}, \text{false} : \text{bool}}\,[\text{Bool}] \quad \frac{\Gamma \vdash e_i : \text{bool}\ (i=1,2)}{\Gamma \vdash e_1 \vee e_2 : \text{bool}}\,[\text{Or}] \quad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathtt{p} : r}\,[\text{Rl}] \quad \frac{\Gamma \vdash \mathsf{Env} \quad y : r \in \Gamma}{\Gamma \vdash y : r}\,[\text{Rlv}]$$

$$\frac{\Gamma \vdash S \triangleright \mathsf{Type} \quad u : S \in \Gamma}{\Gamma \vdash u : S}\,[\text{Id}] \quad \frac{\Gamma \vdash a : \langle G\rangle \quad \Gamma \vdash \Delta : \mathsf{End}}{\Gamma \vdash a\langle G\rangle \triangleright \Delta}\,[\text{Init}] \quad \frac{\Gamma \vdash u : \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright p}{\Gamma \vdash u[p](y).P \triangleright \Delta}\,[\text{Join}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : \text{end}}{\Gamma \vdash \mathtt{quit}\langle c\rangle; P \triangleright \Delta, c : \text{end}}\,[\text{Leave}] \quad \frac{\Gamma \vdash p \quad \Gamma \vdash \vec{p}_j \quad \Gamma \vdash e : S_j \quad \Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash c!\langle p, l_j\langle \vec{p}_j\rangle\langle e\rangle\rangle; P \triangleright \Delta, c : !\langle p, \{l_i\langle \vec{p}_j\rangle\langle S_i\rangle.T_i\}_{i\in I}\rangle}\,[\text{Sel}]$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash \vec{p}_j \quad \Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash c!\langle p, l_j\langle \vec{p}_j\rangle\langle c'\rangle\rangle; P \triangleright \Delta, c : !\langle p, \{l_i\langle \vec{p}_j\rangle\langle T\rangle.T_i\}_{i\in I}\rangle, c' : T}\,[\text{SelS}]$$

$$\frac{\begin{array}{ll}\Gamma \vdash p \quad \forall i \in I \quad \Gamma \vdash \vec{p}_i & \Gamma, y : S_i \vdash P_i \triangleright \Delta, c : T_i \quad (U_i = S_i) \\ & \text{or} \quad \Gamma \vdash P_i \triangleright \Delta, c : T_i, y : T'_i \quad (U_i = T'_i)\end{array}}{\Gamma \vdash c?\langle p, \{l_i\langle \vec{p}_i\rangle(y_i).P_i\}_{i\in I}\rangle \triangleright \Delta, c?\langle p, \{l_i\langle \vec{p}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle}\,[\text{Bra}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'}\,[\text{Par}] \quad \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P; Q \triangleright \Delta; \Delta'}\,[\text{Seq}]$$

$$\frac{\Gamma, x : r \vdash P \triangleright c : T \quad \Gamma \vdash \vec{p}}{\Gamma \vdash c\forall(x : r \setminus \vec{p}).\{P\} \triangleright c : \forall x : r \setminus \vec{p}.T}\,[\text{Polling}] \quad \frac{\Gamma \vdash e : \text{bool} \ \Gamma \vdash P_i \triangleright \Delta\ (i = 1,2)}{\Gamma \vdash \text{if } e \text{ then } P_1 \text{ else } P_2 \triangleright \Delta}\,[\text{If}]$$

$$\frac{\Gamma, a : \langle G\rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a : G)P \triangleright \Delta}\,[\text{New}] \quad \frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta}\,[\text{Rec}] \quad \frac{\Gamma, X : \Delta \vdash \mathsf{Env}}{\Gamma, X : \Delta \vdash X \triangleright \Delta}\,[\text{Rvar}] \quad \frac{\Gamma \vdash \Delta : \mathsf{End}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}\,[\text{Nil}]$$

Figure 6. Multirole session typing for initial processes

4. Multirole session typing system

This section introduces the typing system and proves subject reduction (Theorem 4.2) and type safety (Corollary 4.3). There are three main differences with previous session systems. First, a participant x can appear free in environments, types and processes, and is necessarily bound by universal quantifiers. Second, previous systems did not allow any parallel composition of types which use common channels. Since the projection of a universal quantified type generates parallel compositions, we relax this restriction. Thanks to the well-formedness of the global types (Definition 3.4), the typing system for initial processes is kept simple. Third, our runtime typing system needs to track parallel behaviours by forks and joins.

4.1 Typing Systems

Environments We start with the grammar of environments.

$$\Gamma ::= \varnothing \mid \Gamma, u : S \mid \Gamma, y : r \mid \Gamma, X : \Delta \qquad \Delta ::= \varnothing \mid \Delta, c : T$$

Γ is the *standard environment* which associates variables to sort types or roles, shared names to global types, and process variables to session types. Δ is the *session environment* which associates channels to session types. We write $\Gamma, u : S$ only if $u \notin dom(\Gamma)$. Simiarly for other variables. We define the sequential ; and parallel $\circ$ composition for types as follows:

$$\begin{aligned}\Delta \sharp \Delta' = \quad & \Delta \backslash dom(\Delta') \cup \Delta' \backslash dom(\Delta) \\ & \cup \{c : \Delta(c) \sharp \Delta'(c) \mid c \in dom(\Delta) \cap dom(\Delta')\}\end{aligned}$$

where $\sharp \in \{\circ, ;\}$ and $\Delta(c) \sharp \Delta'(c)$ is syntactically well-formed.

Typing systems for initial processes We detail the typing system for expressions and processes in figure 6. The judgement for expression typing is given as $\Gamma \vdash e : S$. The judgement for process typing is given as $\Gamma \vdash P \triangleright \Delta$ which can be read as: "under the environment Γ, process P has session type Δ".

Rules [Bool,Or,Id] are standard. $\Gamma \vdash \mathsf{Env}$ means that Γ is well-formed, and $\Gamma \vdash S \triangleright \mathsf{Type}$ means S is well-formed under Γ. Since a participant variable with role can appear both in types and environments, we need to use kinding techniques to make sure that types with free variables do not appear before the variables' declarations and ensure well-formedness (see Definition 3.4). Rules [Rl,Rlv] are introduction rules for participants associated with roles.

Rule [Init] types the initialisation of a session with global type G. The judgement $\Gamma \vdash \Delta : \text{end}$ means that Δ only contains end or ε [18, 19]. The rule ensures the initialisation appears only at the top level in the reduction context, not under the prefix. Rule [Join] types a joining process that follows the projection to p. A leaving process is typed if the remaining session type is completed (i.e. end).

Rule [Sel] is for the selection of label l_i, participants $\vec{p}_i$ and payload e. We first infer the destination p from Γ. If e is an atomic type (e.g. bool) or a shared channel type, then it is typed as in standard selection rules [4, 19] for the expression by recording participants $\vec{p}_i$ in the resulting type. This way, we can preserve the dependency between the participants during polling and session communications. Rule [SelS] is a session delegation rule [4, 19]. Rule [Bra] is the dual of the selection rules. Note that the participants p and $\vec{p}_i$ in $c?\langle p, \{l_i\langle \vec{p}_i\rangle(y_i).P_i\}_{i\in I}\rangle$ are free so that they are bound by the polling and dynamically instantiated by reductions. Rules [Par,Seq] assume $\Delta \circ \Delta'$ and $\Delta;\Delta'$ are defined. Rule [Polling] is the introduction rule for the universal quantification. It only concerns a single session (otherwise other sessions are copied after forking). The other rules are standard [4, 19].

Since checking well-formedness is decidable, following the standard method [19, § 4], we have:

Proposition 4.1. *Assuming the bound names and variables in P are annotated (i.e. processes whose bound variables are annotated by types), type-checking of* $\Gamma \vdash P \triangleright \varnothing$ *terminates.*

Typing runtime processes While the session typing systems for initial processes are simple, typing runtime (which keeps tracking intermediate invariants to prove the theorems) is not trivial due to parallel processes and participant instantiations generated by polling. We first extend the syntax of types T to include *message selection* type $!\langle \mathtt{p} : r, l\langle \vec{\mathtt{p}}\rangle\langle U\rangle\rangle$, which is an intermediate type for labelled values stored in the message buffer.

To type runtime processes, we need to extend judgements to $\Gamma \vdash_\Sigma P \triangleright \Delta$, which means that P contains the message buffers whose session names are in Σ. We only show the most interesting typing rule for the register:

$$\frac{\Gamma \vdash a : \langle G\rangle \quad \{r_i\}_{i\in I} = dom(\mathtt{R}) \quad G \upharpoonright x_i : r_i = T_i}{\Gamma \vdash_\varnothing a\langle s\rangle[\mathtt{R}] \triangleright \{s[\mathtt{p}_{ji} : r_i] : T_i\{\mathtt{p}_{ji}/x_i\}\}_{i \in I, \mathtt{p}_{ji} \notin \mathtt{R}(r_i)}}\,[\text{Rgst}]$$

[Rgst] assigns to the registry a type which holds a set of projected local types for all roles with participants which are *not* recorded in $\mathtt{R}$. Session typing $s[r_i : \mathtt{p}_{ji}] : T_i\{\mathtt{p}_{ji}/x_i\}$ is erased once it interacts with the initialisation process $a[\mathtt{p}_{ji} : r_i](y).P$ (see rule ⌊Join⌋ in figure 2), and the resulting $P\{s[\mathtt{p}_{ji} : r_i]/y\}$ holds $s[r_i : \mathtt{p}_{ji}] : T_i\{\mathtt{p}_{ji}/x_i\}$ (see the proof of Subject reduction theorem in [3]).

4.2 Subject reduction

As session participants join, interact and leave, runtime session types need to follow. This dynamism is formalised by a type reduction relation $\Rightarrow$ on session environments as follows.

1. $\{s[\mathtt{q} : r'] : !\langle \mathtt{p} : r, \{l_i\langle \vec{\mathtt{p}}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathtt{q} : r'] : !\langle \mathtt{p} : r, l_k\langle \vec{\mathtt{p}}_i\rangle\langle U_k\rangle.T_k\rangle\}$
2. $s[\mathtt{p} : r] : !\langle \mathtt{q} : r', l_k\langle \vec{\mathtt{p}}_k\rangle\langle U_k\rangle\rangle, s[\mathtt{q} : r'] : ?\langle \mathtt{p} : r, \{l_i\langle \vec{\mathtt{p}}_k\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle$
   $\Rightarrow s[\mathtt{p} : r] : \varepsilon, s[\mathtt{q} : r'] : T_k \quad$ if $k \in I$
3. $s[\mathtt{p} : r] : !\langle \mathtt{p} : r, l_k\langle \vec{\mathtt{p}}_k\rangle\langle U_k\rangle\rangle; ?\langle \mathtt{p} : r, \{l_i\langle \vec{\mathtt{p}}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle \Rightarrow s[\mathtt{p} : r] : T_k$ if $k \in I$
4. $s[\mathtt{p} : r] : \forall x : r_i \setminus \vec{\mathtt{p}}.T \Rightarrow s[\mathtt{p} : r] : (T\{\mathtt{p}_1/x\} \mid .. \mid T\{\mathtt{p}_k/x\}) \quad$ with $\mathtt{p}_i \notin \vec{\mathtt{p}}$
5. $s[\mathtt{p} : r] : \mathscr{E}[T] \cup \Delta \Rightarrow s[\mathtt{p} : r] : \mathscr{E}[T'] \cup \Delta' \quad$ if $\quad s[\mathtt{p} : r] : T \cup \Delta \Rightarrow s[\mathtt{p} : r] : T' \cup \Delta'$
6. $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta'' \quad$ if $\quad \Delta \Rightarrow \Delta'$

In the above type reduction rules, message selection types are considered modulo the type equivalence relation $\equiv$ and $\mathscr{E}$ is a type evaluation context (i.e. $\mathscr{E} ::= [\_] \mid \mathscr{E} \mid T \mid T \mid \mathscr{E} \mid \mathscr{E}; T$).

Rule (1) corresponds to the choice of label l_i. Rule (2) corresponds to the exchange of a labelled value from participant $\mathtt{p}:r$ to participant $\mathtt{q}:r'$. Rule (3) is about self-sending and receiving. Rule (4) governs universal quantifiers and forks types with respect to the participants which are not in the exclusion list $\vec{p}$. Rules (5,6) are congruent rules.

Hereafter we assume all processes are derived from the initial processes (§ 2) (i.e. subterms of those who are reduced from initials). Using the above definitions,

THEOREM 4.2 (Subject reduction). *Suppose that* $\Gamma \vdash_\Sigma P \triangleright \Delta$ *and that* $P \longrightarrow^* P'$. *Then,* $\Gamma \vdash_\Sigma P' \triangleright \Delta'$ *for some* Δ' *such that* $\Delta \Rightarrow^* \Delta'$.

We say P has ***a type error*** if expressions in P does contain either a type error for a value or constant in the standard sense (e.g. if 3 then P else Q) or a label error (e.g. the sender sends a value with label l_0 while the receiver does not expect label l_0). From the subject reduction theorem and the well-formedness of global types (Definition 3.4), we can prove:

COROLLARY 4.3 (Type safety). *Suppose that* $\Gamma \vdash P \triangleright \Delta$. *For any* P' *such that* $P \longrightarrow^* P'$, P' *has no type error.*

5. Communication safety and progress

This section discusses the difficulties that a distributed session semantics creates when participants can dynamically join, leave and poll. We illustrate two limitations of the semantics and typing system presented so far and propose a solution based on multiparty locking that allows more flexibility for leaving a session and guarantees communication safety. We give two progress properties, one of which goes beyond existing achievements.

5.1 Limitations

Leaving a session While our operational semantics ($\lfloor$QUIT$\rfloor$ in figure 2) allows a participant to leave a session any time, the typing rule ([LEAVE] in figure 6) restricts a participant to leave only when its local type is end.

Recall the P2P chat example in (2) in § 1, $G = \mu\mathbf{x}.G_0;\mathbf{x};\text{end}$ with $G_0 = (\forall x{:}\text{client}.\forall y{:}\text{client} \setminus x.\{x \rightarrow y\langle \mathsf{Msg}\rangle\})$). The recursive type prevents any participant from ever leaving since a process will never reach type end. We however remark that a client can play just one interaction round (i.e. G_0) and leave safely before another session iteration occurs. If the starting and ending points of global types are known, some participants are able to leave a session safely while others stay.

Communication safety and progress In traditional multiparty sessions, the subject reduction theorem immediately brings *communication safety* and *progress* (in a single session) [19]. The reason is that standard multiparty session typing ensures that all parties can be eventually present and ready since the participant number is fixed when the session initiated. This does not hold in our system due to the interplay between joining, leaving and polling.

We illustrate this point with the peer-to-peer chat example from § 1. In that global type, every client is broadcasting Msg to all the others. Recall the client process $P_{\text{client}}(z)$ from § 1.

$$\begin{array}{l} a[z{:}\text{client}](s).\mu X.(s\forall(y{:}\text{client} \setminus z).\{s!\langle y, \mathsf{Msg}\langle m\rangle\rangle\} \\ \qquad\qquad\qquad\qquad \mid s\forall(x{:}\text{client} \setminus z).\{s?\langle x, \mathsf{Msg}(w)\rangle\} \quad);X \end{array}$$

At each iteration, every client does exactly two polling operations. Now suppose that a client does the first polling operation (to send Msg) before another client joins. It means that this new client will not receive the message it expects. More generally, the polls that correspond to the emissions need to always give the exact same result as the reception polls. This suggests that some mechanism to synchronise distributed polling processes is required to guarantee consistent polling results.

5.2 Multiparty locking for polling synchronisation

This subsection shows that a simple locking policy that can be automatically computed from the global type is able to ensure a safe synchronisation to allow flexible session departure and consistent polling results. The key point is to temporarily *block* late participants from joining in the middle of a session execution in order to prevent any interference with polling. This is simply done by automatically surrounding global types by locks: $\mathtt{lock}\{G\}$ means that the *interactions specified by G are protected from late joiners* and is called a *locked global type*. This condition is easily implementable using a standard two phase commitment protocol which minimises the necessary synchronisation between processes (figure 7) and is easily implementable in ML (§ 5.5).

The peer-to-peer chat example from § 1 is now defined by $\mu\mathbf{x}.\mathtt{lock}\{\forall x{:}\text{client}.\forall y{:}\text{client}.x \rightarrow y\langle \mathsf{Msg}\rangle\};\mathbf{x}$. This type allows participants to join at each recursive iteration, preventing interferences while the exchange of Msg is under way.

Syntax We first extend the syntax of processes (figure 1) and types (figure 4) as follows:

$$\begin{array}{l} P ::= \ldots \mid c\ \mathtt{lock} \mid c\ \mathtt{unlock} \mid a^{\circ}[\mathtt{R},\Lambda] \mid a^{\bullet}[\mathtt{R},\Lambda] \\ \Lambda ::= \varnothing \mid \Lambda \cup \{\mathtt{p}{:}r\} \\ G ::= \ldots \mid \mathtt{lock}\{G\} \qquad T ::= \ldots \mid \mathtt{lock} \mid \mathtt{unlock} \end{array}$$

The process syntax is extended to locking and unlocking operations. The registry has now two new states: $a^{\circ}[\mathtt{R},\Lambda]$ represents a registry that is in the process of being locked (so far by participants Λ), while $a^{\bullet}[\mathtt{R},\Lambda]$ represents a registry that is locked (and where participants Λ are still involved). For global types, we extend the well-formedness definition: $\mathtt{lock}\{G\}$ is well-formed if G does not contain free type and participant variables and $\mathtt{lock}$ does not appear in G. Local types $\mathtt{lock}$ and $\mathtt{unlock}$ come from the projection:

$$\mathtt{lock}\{G\} \upharpoonright z{:}r \quad = \quad \mathtt{lock};(G \upharpoonright z{:}r);\mathtt{unlock}$$

This way, correct locks are automatically inserted at the right points of the local types. In addition, a process can safely leave a session when $\mathtt{unlock}$. Typing $\mathtt{lock}$ and $\mathtt{unlock}$ is straightforward, and $\mathtt{quit}\langle c\rangle$ is typed when session channel c is unlocked.

$$\frac{\Gamma \vdash \Delta{:}\mathsf{End}}{\Gamma \vdash c\ \mathtt{lock} \triangleright c{:}\mathtt{lock},\Delta} \qquad \frac{\Gamma \vdash \Delta{:}\mathsf{End}}{\Gamma \vdash c\ \mathtt{unlock} \triangleright c{:}\mathtt{unlock},\Delta}$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c{:}\mathtt{unlock}}{\Gamma \vdash \mathtt{quit}\langle c\rangle;P \triangleright \Delta, c{:}\mathtt{unlock}}$$

Semantics The operational semantics with multiparty locking is given in figure 7. It defines the relations between the three states of the registry and this is based on a standard two phase locking protocol commonly found in distributed applications.

The first phase is the *registration state*: if the registry is of the form $a\langle s\rangle[\mathtt{R}]$, participants can join and leave the session through $\lfloor$JOIN$\rfloor$ and $\lfloor$QUIT$\rfloor$. The only other reduction rule that can be applied is $\lfloor$LOCK$\rfloor$, which puts the registry in its second state, the *locking state* $a^{\circ}[\mathtt{R},\ell{:}\Lambda]$. Then, the session can only wait for all the current participants in $\mathtt{R}$ to activate their locks by the rules $\lfloor$UP, TOP$\rfloor$ (no other rule can be used). The difference between $\lfloor$UP$\rfloor$ and $\lfloor$TOP$\rfloor$ lies in the side condition: $\mathtt{R} \approx \Lambda$ holds when $\forall \mathtt{p}{:}r.(\Lambda = \Lambda' \uplus \{\mathtt{p}{:}r\} \Leftrightarrow \mathtt{R} = \mathtt{R}' \cdot r{:}\mathtt{P} \uplus \{\mathtt{p}\})$. Consequently, $\lfloor$TOP$\rfloor$ is only triggered when the set Λ contains the exact same combinations of participants and role as the set $\mathtt{R}$, meaning that all participants have activated their locks.

The application of rule $\lfloor$TOP$\rfloor$ marks the beginning of the *interaction phase*, with a registry of the form $a^{\bullet}\langle s\rangle[\mathtt{R},\ell{:}\Lambda]$. Only in this state (or additionally in the locking state, see § 5.5) can the rules $\lfloor$ SEND, RECV, POLL$\rfloor$ be safely applied. The registry goes back to its registration state by the application of rule $\lfloor$UNLOCK$\rfloor$ which can occur only when everyone besides one participant has activated the $\mathtt{unlock}$ operation by rule $\lfloor$DOWN$\rfloor$.

$$
\begin{array}{rll}
a\langle G\rangle \rightarrow (\nu\ s)(a\langle s\rangle[\mathtt{R}] \mid s\!:\!\varepsilon) \quad (\forall r_i \in G, \mathtt{R}(r_i) = \varnothing) & & \lfloor\text{INIT}\rfloor \\
a[\mathtt{p}\!:\!r](y).P \mid a\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P}] \rightarrow P\{s[\mathtt{p}\!:\!r]/y\} \mid a\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P}\uplus\{\mathtt{p}\}] & & \lfloor\text{JOIN}\rfloor \\
\mathtt{quit}\langle s[\mathtt{p}\!:\!r]\rangle \mid a\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P}] \rightarrow a\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P}\setminus\{\mathtt{p}\}] & & \lfloor\text{QUIT}\rfloor \\
s[\mathtt{p}\!:\!r]\mathtt{lock} \mid a\langle s\rangle[\mathtt{R}] \rightarrow a^{\circ}\langle s\rangle[\mathtt{R},\{\mathtt{p}\!:\!r\}] & & \lfloor\text{LOCK}\rfloor \\
s[\mathtt{p}\!:\!r]\mathtt{lock} \mid a^{\circ}\langle s\rangle[\mathtt{R},\Lambda] \rightarrow \begin{cases} a^{\circ}\langle s\rangle[\mathtt{R},\Lambda\uplus\{\mathtt{p}\!:\!r\}] & (\mathtt{R}\not\approx\Lambda\uplus\{\mathtt{p}\!:\!r\}) \\ a^{\bullet}\langle s\rangle[\mathtt{R},\Lambda\uplus\{\mathtt{p}\!:\!r\}] & (\mathtt{R}\approx\Lambda\uplus\{\mathtt{p}\!:\!r\}) \end{cases} & & \begin{array}{l}\lfloor\text{UP}\rfloor \\ \lfloor\text{TOP}\rfloor\end{array} \\
s[\mathtt{p}\!:\!r]\mathtt{unlock} \mid a^{\bullet}\langle s\rangle[\mathtt{R},\Lambda\uplus\{\mathtt{p}\!:\!r\}] \rightarrow \begin{cases} a^{\bullet}\langle s\rangle[\mathtt{R},\Lambda] & (\Lambda\neq\varnothing) \\ a\langle s\rangle[\mathtt{R}] & (\Lambda=\varnothing) \end{cases} & & \begin{array}{l}\lfloor\text{DOWN}\rfloor \\ \lfloor\text{UNLOCK}\rfloor\end{array} \\
s[\mathtt{p}\!:\!r]\forall(x\!:\!r\setminus\vec{\mathtt{p}}).\{P\} \mid a^{\bullet}\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P},\Lambda] \rightarrow P\{\mathtt{p}_1/x\} \mid \ldots \mid P\{\mathtt{p}_k/x\} \mid a^{\bullet}\langle s\rangle[\mathtt{R}\cdot r\!:\!\mathtt{P},\Lambda] & & \\
(\mathtt{R}(r)\setminus\vec{\mathtt{p}} = \{\mathtt{p}_1,\ldots,\mathtt{p}_k\} \wedge \mathtt{p}\in\mathtt{R}(r')) & & \lfloor\text{POLL}\rfloor \\
s[\mathtt{p}\!:\!r]!\langle\mathtt{p}'\!:\!r', l\langle\vec{\mathtt{p}}\rangle\langle v\rangle\rangle \mid a^{\bullet}\langle s\rangle[\mathtt{R},\Lambda] \mid s\!:\!h \rightarrow a^{\bullet}\langle s\rangle[\mathtt{R},\Lambda] \mid s\!:\!h\cdot(\mathtt{p}\!:\!r,\ \mathtt{p}'\!:\!r',\ l\langle\vec{\mathtt{p}}\rangle\langle v\rangle) & & \\
(\mathtt{p}\in\mathtt{R}(r)\wedge\mathtt{p}'\in\mathtt{R}(r')) & & \lfloor\text{SEND}\rfloor \\
s[\mathtt{p}\!:\!r]?\langle\mathtt{p}'\!:\!r', \{l_i\langle\vec{\mathtt{p}}_i\rangle(x_i).P_i\}_{i\in I}\rangle & & \\
\mid a^{\bullet}\langle s\rangle[\mathtt{R}] \mid s\!:\!(\mathtt{p}'\!:\!r',\ \mathtt{p}\!:\!r,\ l_k\langle\vec{\mathtt{p}}_k\rangle\langle v\rangle)\cdot h \rightarrow P_k\{v/x_k\} \mid a^{\bullet}\langle s\rangle[\mathtt{R}] \mid s\!:\!h \quad (\mathtt{p}\in\mathtt{R}(r)\wedge k\in I) & & \lfloor\text{RECV}\rfloor
\end{array}
$$

Other rules are from 2.

Figure 7. Operational semantics with multiparty lock

Before moving to the main theorems, we define the condition for global environments. We say that a global type G is *fair* if there exists at least one finite path (whose leaf is ε) up to unfolding G. A fair type can be easily defined by a kinding system: ε is fair; $p\to p'\{l_i\langle\vec{p}_i\rangle(U_i).G_i\}_{i\in I}$ is fair if for some $k\in I$, G_k is fair; and others are defined homomorphically (see [3]).

DEFINITION 5.1 (Well-locked and persistently well-locked). We say that a closed global type G is *well-locked* if G is of the form $\mathtt{lock}\{G_0\};\mathsf{end}$ and G_0 does not include any $\mathtt{lock}$. We say that a closed global type G is *persistently well-locked* if G is of the form $\mu\mathbf{x}.\mathtt{lock}\{G_0\};\mathbf{x};\mathsf{end}$, with $\mathtt{lock}\{G_0\}$ well-locked and G_0 is fair. We call Γ *well-locked* if for all $\Gamma(u)=\langle G\rangle$, G is either well-locked or persistently well-locked. We call Γ *persistently well-locked* if for all $\Gamma(u)=\langle G\rangle$, G is persistently well-locked.

$\mathtt{lock}\{G_0\}$ means that a single multiparty session is locked. $\mu\mathbf{x}.\mathtt{lock}\{G_0\};\mathbf{x}$ states a multiparty session is *persistently* (repeatedly) locked. The persistent lock ensures if a new participant $\mathtt{p}$ wants to join, it can join to the beginning of the interaction G_0, and if one wishes to quit, it can quit at the end of the session. Thus it requires that the global type is of the form $\mu\mathbf{x}.G_0;\mathbf{x}$ with G_0 is well-locked and do not contain any possibly infinite loop to reach the end point ($\mathtt{unlock}$). The persistent condition is needed for the final strong join progress discussed later.

5.3 Communication safety and progress

We first state communication safety — every receiver has a corresponding sender with the right label. In effect, it states that, in a session execution, no receiver waits for a message that will never come. It can be extended to exclude messages sent but not received.

DEFINITION 5.2 (Communication-safety). We say P is ***communication safe*** if:

- $P\equiv\mathscr{E}[s[\mathtt{p}\!:\!r]?\langle\mathtt{p}'\!:\!r',\{l_i\langle\vec{\mathtt{p}}_i\rangle(x).P_i\}_{i\in I}\rangle]$ implies that there exists $\mathscr{E}'$ such that $\mathscr{E}[\mathbf{0}]\rightarrow^{*}\mathscr{E}'[s\!:\!(\mathtt{p}'\!:\!r',\ \mathtt{p}\!:\!r,\ l_k\langle\vec{\mathtt{p}}_k\rangle\langle v\rangle)\cdot h]$ with $k\in I$
- $P\equiv\mathscr{E}[s\!:\!(\mathtt{p}'\!:\!r',\ \mathtt{p}\!:\!r,\ l_k\langle\vec{\mathtt{p}}_k\rangle\langle v\rangle)\cdot h]$ implies that there exists $\mathscr{E}'$ such that $\mathscr{E}[\mathbf{0}]\rightarrow^{*}\mathscr{E}'[s[\mathtt{p}\!:\!r]?\langle\mathtt{p}'\!:\!r',\{l_i\langle\vec{\mathtt{p}}_i\rangle(x).P_i\}_{i\in I}\rangle]$ with $k\in I$.

The first statement means that branching processes can always find out a correct element in the message buffer; and the second one is its dual. Note that combining with Type safety, the receiver will input value v with expected type.

THEOREM 5.3 (Communication safety). *Suppose $\Gamma\vdash P\rhd\Delta$ and Γ is well-locked. If $P\rightarrow^{*}P'$, then P' is communication safe.*

The proof starts by a definition of coherent environments (a certain kind of duality relations over multiple participants [4, § 3]). Then, we prove a stronger subject reduction theorem that shows the reduction of well-locked processes preserves the coherency of the resulting environment. Then we get the above theorem as a corollary. We also note that session fidelity [19, Corollary 5.6] is a corollary.

Now we prove the *progress* property in a single multiparty session as in [19, Theorem 5.12], i.e. if a program P starts from one session, the reductions at session channels do not get stuck.

DEFINITION 5.4 (Single-session join). We write $\Gamma\vdash^{\star}P\rhd\Delta$ if P is typable and with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to be at most a singleton (more precisely, $\Delta=\varnothing$ in [JOIN,LEAVE,SEL,SELS,BAR] and Δ contains at most one element in $\Delta;\Delta'$ in [SEQ], $\Delta\circ\Delta'$ in [PAR,SEQ] and Δ in [IF,NEW,REC,RVAR] in figure 6). We say $Q=a[p](y).Q'$ is a *single-session* join if $a\!:\!\langle G\rangle\vdash^{\star}Q\rhd\varnothing$ and Q' does not contain shared name restriction and any join process.

$\Gamma\vdash^{\star}P\rhd\Delta$ ensures that P contains (several) join processes each of which holds a single session, while single-session join $a[p](y).Q'$ has only one active point a, and once the session initiated at a, Q' can only perform session communication at that initiated session.

DEFINITION 5.5 (Progress property). We say $\Gamma\vdash P\rhd\varnothing$ can progress, or satisfies the ***progress property***, if $P\rightarrow^{*}P'$, then either $P'\equiv\mathbf{0}$, $P'\rightarrow R$ or for some single-session join $a\!:\!\langle G\rangle\vdash Q$ with $a\!:\!\langle G\rangle\in\Gamma$ such that $P'\mid Q\rightarrow R$ and R can progress.

The above definition means that a process satisfies the progress property if it can never reach a deadlock state, i.e., if it never reduces to a process which contains active sessions (this amounts to containing waiting process at some session channel) and which is irreducible in any inactive context with single-session join Q running in parallel.

THEOREM 5.6 (Progress). *Suppose $\Gamma\vdash^{\star}P\rhd\varnothing$ and P is initial. Assume Γ is well-locked and P does not contain any shared name restriction. Then P can progress.*

5.4 Join progress

The above standard progress property is not strong enough, since all late joiners cannot participate to existing sessions. This subsection states a new progress property, not found in the literature.

Recall the (1) map-reduce example from § 1, and change the position of the recursion in the global type to

$$G_0 = \forall x\!:\!\text{client}.\text{server}\!\to\! x\langle\mathsf{Map}\rangle;\mu\mathbf{x}.x\!\to\!\text{server}\langle\mathsf{Reduce}\rangle;\mathbf{x}$$

From G_0, we have the following well-typed processes:

$$P_0(s,z\!:\!\text{client}) = s?\langle\text{server},\mathsf{Map}\rangle;\mu X.s!\,\langle\text{server},\mathsf{Reduce}\rangle;X$$
$$P_0(s,z\!:\!\text{server}) = s\forall(x\!:\!\text{client}).\{s!\,\langle x,\mathsf{Map}\rangle;\mu X.s?\langle x,\mathsf{Reduce}\rangle;X\}$$

While the interaction between them is communication safe, the problem is that a late client will never be listened to by *the existing server* because the server's polling operation is not repeated to include him. In other words, the late client cannot join an existing, already running session. Persistent locking ensures this situation does not happen.

Below we write $P\xrightarrow{s[\mathtt{p}:r]}Q$ if $P \to Q$ and $P \to Q$ is derived using $\lfloor$Quit$\rfloor$, $\lfloor$Send$\rfloor$, $\lfloor$Recv$\rfloor$ or $\lfloor$Poll$\rfloor$ at $s[\mathtt{p}:r]$ with $\lfloor$Par,Ctx,Cong$\rfloor$, i.e. P interacts with a queue or registry through $s[\mathtt{p}:r]$.

Definition 5.7 (Join progress property). We say that $a:\langle G\rangle \vdash P \triangleright \varnothing$ satisfies the ***join progress property***[4] if:

- P can progress; and
- if $P \to^* (\nu\ s)(P' \mid a\langle s\rangle[\mathtt{R}])$ then, for any single-session join $a:\langle G\rangle \vdash a[\mathtt{p}:r](y).Q\triangleright\varnothing$ with $\mathtt{p}:r$ fresh, and for any R such that $P' \mid a\langle s\rangle[\mathtt{R}] \mid a[\mathtt{p}:r](y).Q \to^* a^\bullet\langle s\rangle[\mathtt{R}'] \mid R$,
  - if $s[\mathtt{p}:r]\in R$, then there exists $a^\bullet\langle s\rangle[\mathtt{R}'] \mid R \to^* \xrightarrow{s[\mathtt{p}:r]} R'$; and
  - $(\nu\ s)R$ satisfies the join progress property.

The above definition says that a fresh joiner ($a[\mathtt{p}:r](y).Q'$) can always join the existing (unlocked) session s in P'. In addition, it can always progress at the created session channel s by interacting with P'. More intuitively, once some participants under any role start a session, the late joiner can still join that session and interact with earlier joiners, progressing further. Note that we can consider *any* single-session join $a[\mathtt{p}:r](y).Q'$ to make a process progress which contrasts with the definition of the progress property (Definition 5.5) where P is only composed with *some* single-session join.

Theorem 5.8 (Join progress). *Suppose $a:\langle G\rangle \vdash^\star P \triangleright \varnothing$ and P is initial. Assume $a:\langle G\rangle$ is persistently well-locked and P does not contain any shared name restriction. Then P satisfies the join progress property.*

We have for our examples:

Proposition 5.9 (Properties of the examples). *Assume each global type G in the protocols (1–3) of § 1 is replaced by* $\mathsf{lock}\{G\}$. *Then all examples are type/communication safe and can always progress. Moreover if each global type in the protocols (1,2) of § 1 inside the recursive type, i.e. $\mu\boldsymbol{x}.G;\boldsymbol{x}$ is replaced by $\mu\boldsymbol{x}.\mathsf{lock}\{G\};\boldsymbol{x}$, then they additionally satisfy the join progress property.*

5.5 Implementation

Prototype implementation The multirole calculus has been implemented as an extension of ML. Following the technique used in [5, 13], the global types that the programmer writes are compiled into an end-point function for each role. This choice allows to replace the implementation of the typing system by an automated generation of well-typed processes that can be used, through an API, by the programmer. The session semantics is thus entirely generated and implemented by communication libraries.

A distributed implementation The main issue for our compiler is to distribute as much as possible the centralised aspects of the semantics of figures 2 and 7. First concerned, the message buffers are completely distributed and implemented on the sender side: a thread is spawned to asynchronously make sure that the message gets across the TCP channel. Second, the registries can be partially distributed with one registry per role that deals with the corresponding joining, leaving and polling activities. These distributed registries however need to stay in contact to synchronise the global locking events (rules $\lfloor$ Lock, Top, Unlock$\rfloor$). Registries are attributed to participants by age: the first joiner for a role plays the registry as well, until he quits, in which case the registry is transmitted to the second older.

Extension Singly instantiated roles, like the server or broker from examples in § 1, are modelled through an inefficient implicit quantification. Our implementation gives a special status to these roles. We use the fact that they play their own registry. As a consequence, no separate polling is necessary to send them messages and the extra messages required by the quantification can be avoided.

Efficiency To gain performances, we propose an implementation with optimised messaging and improved asynchrony.

Since the slowest operation is communication, our implementation tries to minimise the number of messages that are exchanged. For example, if a session requires to poll several time for the same role inside the same locked section, the polling is done only once. In the same way, if two messages are specified to be sent in a row between the same participants, they are automatically concatenated.

The second way to increase the efficiency of our distributed implementation is to increase its asynchrony. This can be done (as suggested in § 5) by allowing communication in both the locking and interaction states. This removes the global synchronisation point of rule $\lfloor$Top$\rfloor$ and allows more flexibility for processes to asynchronously activate their lock.

6. Related Work

The first motivation for our work is the strong need to extend session type theory with *role*-based abstraction to support the class of communication protocols in practice.

Scribble [17, 28] is a specialised Java-like language for describing multiparty session types that features roles as a design mechanism, but which lacks the formal foundations, as presented in this paper, required for full role-based projection and type-checking. Scribble is currently being experimented with for several different applications in distributed systems [24, 27] and business and financial protocols [2, 32]. Their tool projects multiparty protocols into local types, generating a Java skeleton for protocols which includes roles, sending-receiving, branching and recursion. A user then implements session programs using a Java API following the generated skeleton. Its origin is WS-CDL [1], where a role is declared as a unit of abstract behaviour for participants. A protocol (a multiparty session type) is defined first by specifying the roles and which role each participant belongs to, and then initiated by assigning the role to each participant. Our auction example was extracted from a Scribble specification.

The need for roles in session programming is also substantiated by our experiences in implementing parallel algorithms for high-performance clusters using Session Java (SJ) [20, 21], see [25].

The second motivation is the incorporation of dynamic features most suited to and compatible with existing multiparty session types [4–6, 9, 14, 19, 22, 23, 29, 35, 36]. The Conversation Calculus [8, 33] models distributed behaviours among "places" using new primitives such as conversation contexts (i.e. shared interaction points) and up ($\uparrow$) communication (similar to [7, 12]). A conversation models the interactions between a client and various services, with dynamic joining into a conversation, for a possibly

[4] The property can be generalised to Γ from $\{a:\langle G\rangle\}$ if we compose a parallel compositon of single-session processes $Q_1 \mid \ldots \mid Q_n$ to R in the first item such as $a^\bullet\langle s\rangle[\mathtt{R}'] \mid R \mid Q_1 \mid \ldots \mid Q_n \to^* \xrightarrow{s[\mathtt{p}:r]} R'$.

unknown number of processes. While both their work and ours aim to support dynamic natures for sessions, the two join mechanisms are quite different. Their join is encoded by base primitives for late joining into a point of conversation, which more closely resembles the late asynchronous session initiation in [20, 35]. On the other hand, our join mechanism is *role-based*, and articulated at the level of *global types*, by declaring a single type construct which binds participants to a role. In contrast to [8], the process which controls joining might be a sender or listener, depending on the result of the projection (i.e. the position of polling). This flexibility enables direct modelling and clear articulation (i.e. without encoding) of different patterns of dynamic parallel protocols including symmetric P2P chats (§ 1 and Examples 3.1 and 3.2) by types. In [8], they proposed a sophisticated typing system that builds a well-founded order on events (similar to the line of [34]), to guarantee progress for processes under the assumption that all communications are matched with sufficient joiners. They do not, however, explore type inference for progress (decidability of a generation of well-formed ordering) [33]. Our progress can be, on the other hand, guaranteed by well-formedness of global types, with an automatic insertion of locks (which means a typing system with progress is decidable with Proposition 4.1). This leads to a simple but practical prototype implementation as discussed in § 5.5. A strong joining property has not been studied in [8].

Contracts [11] record the overall behaviour of a process, and typable processes themselves may not always satisfy the properties of session types such as progress: it is proved *later* by checking whether a whole contract satisfies a certain form. Proving properties with contracts requires an exploration of all possible interleaved or non-deterministic paths of a protocol. See also [35, § 5].

Our recent work [35] provides parameterised global types which can be instantiated by different numbers of participants based on indexed dependent types. Like other session works, the number of participants is fixed once a session is established.

A master's thesis [26] studies a Java-like core calculus for multiroles (based on a preliminary idea in [16]), where a role may be a collection of processes whose membership can vary. In contrast to our system, a new member of conversations can only join a conversation if it has not already started. Progress has not been proved either.

For further comparisons of session types with other service-oriented calculi and behaviour typing systems, see [15] for a wide ranging survey of the related literature.

7. Conclusion and Future Work

This work introduced a multirole session type discipline for validating dynamic behaviours among an unspecified number of participants, answering a well-known open problem of multiparty session types [4, 6, 9, 14, 19, 22, 29, 35]. Dynamism is formalised through a powerful universal type construct which can represent many collective communications protocols, ranging over parallel computations, P2P networking, chat protocols and e-commerce auctions. Despite the greater expressiveness, projection and type checking are decidable. Global types offer a practical guideline for a correct multiparty synchronisation mechanism, by which the theorems (properties) are articulated as: (1) $\forall x.G$ (subject reduction and type safety with dynamic join and leave semantics), (2) well-locked $\mathtt{lock}\{G\}$ (communication safety and progress); and (3) persistently well-locked $\mu\mathbf{x}.\mathtt{lock}\{G\};\mathbf{x}$ (join progress). Our prototype implementation demonstrates the direct applicativity of the present theory.

To realise the full potential of the multirole session type theory, several challenges need to be addressed. First, the theory can be integrated with the multiparty session exceptions developed in [10] in order to handle system failure and fault-tolerance in a larger class of distributed protocols, preserving type safety. It is especially useful to directly express more complex and dynamic topologies, in combination with the parameterised type theory from [35].

One extension that comes immediately to mind is addition of an explicit existential $\exists x\!:\!r.G$. It however raises many semantic issues. Consider $G' = \forall x\!:\!\mathsf{client}.\{\exists y\!:\!\mathsf{server}.x \to y\{\mathsf{Msg}\}\}$. In that example, every client contacts a server (the intuition is that each x chooses a unique y). The question is: how can we ensure by local typing that servers will be listening to the right number of requests? The difficulty is that a server y can be potentially chosen by every client x or by none, and that this choice is distributed (and thus very hard to locally type check). Consequently, the global existential quantification rather abstracts complex distributed election algorithms. A different solution is to extend to subtyping between roles $r_1 <: r_2$ by which we can represent a protocol with memberships, e.g. a client sends a message to a subset of subscribers.

Second, type-based approaches for correct locking has been widely studied, including [30] in a framework of linear program analysis and types. Our aim in § 5.2 is to propose a simple way to realise synchronisation, articulated by global types, suggesting another use of global descriptions for different purposes. One such instance is studied in [14], where multiparty session types lead to an efficient buffer analysis, along with automatically guaranteed communication and buffer safety. A benefit of using global types (i.e. a choreography framework [1]) is that the analysis can be done solely based on global types, without directly analysing (possibly distributed) end-point types or processes since we can assume all processes agree with that global specification. An integration with global and local locking [30] is, however, an interesting future topic from the viewpoint of local refinements [22].

Third, we are currently collaborating with several industry partners working on open standardisations for financial protocols [32] and messaging middleware [2], governance architectures [27], and cyberinfrastructures [24], to attest the practical use and expressiveness of the session framework, for which an integration with multiparty logic [6] and security [5, 9] for monitoring, is our next task.

Acknowledgement We are grateful to Luis Caires, Marco Carbone, Mariangiola Dezani-Ciancaglini, Kohei Honda, Raymond Hu, Kohei Suenaga, Vasco Vasconcelos and Soren Debois for their helpful comments. We thank the anonymous reviewers for their precise questions and suggestions. This work was supported by EPSRC EP/F003757/01 and G015635/01.

References

[1] Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.

[2] Advanced Message Queueing Protocols. `http://www.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Pr%otocol`.

[3] On-line Appendix of this paper. `http://www.doc.ic.ac.uk/~pmalo/dynamic`.

[4] L. Bettini et al. Global Progress in Dynamically Interfered Multiparty Sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[5] K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.

[6] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.

[7] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *TOPLAS*, 26(1):57–124, 2004.

[8] L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009. A full version will appear in TCS.

[9] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session Types for Access and Information Flow Control. In *CONCUR'10*, volume 6269 of *LNCS*, pages 237–252. Springer, 2010.

[10] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *30th FSTTCS'10*, LIPICS, 2010. To appear.

[11] G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR*, number 5710 in LNCS, pages 211–228, 2009.

[12] G. Castagna, J. Vitek, and F. Z. Nardelli. The seal calculus. *Inf. Comput.*, 201(1):1–54, 2005.

[13] R. Corin and P. Deniélou. A protocol compiler for secure sessions in ML. In *TGC*, volume 4912 of *LNCS*, pages 276–293. Springer, 2008.

[14] P.-M. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010. Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/multianalysis`.

[15] M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and Session Types: an Overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.

[16] E. Giachino, M. Sackman, S. Drossopoulou, and S. Eisenbach. Softly safely spoken: role playing for session types. Preliminary on-line preproceeding, 64–69 pages, `http://gloss.di.fc.ul.pt/places09/preproceedings.pdf/view`.

[17] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, LNCS. Springer, 2011. To appear.

[18] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[19] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284, 2008.

[20] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-Safe Eventful Sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.

[21] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.

[22] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332, 2009.

[23] L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sum-types. Technical Report 8, Department of Computing, Imperial College London, 2009. To appear in Express'10. Apims Project at: `http://www.thelas.dk/index.php/apims`.

[24] Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observin%g/ooi/`.

[25] O. Pernet, N. Ng, R. Hu, N. Yoshida, and Y. Kryftis. Safe Parallel Programming with Session Java. Technical Report 14, Department of Computing, Imperial College London, 2010.

[26] A. Raad. *Smelling of Roses: ROles, Specification, Specification and Scrutiny*. DoC master's thesis, Imperial College London, 2010.

[27] Savara JBoss Project. `http://www.jboss.org/savara`.

[28] Scribble Project. `http://www.jboss.org/scribble`.

[29] K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *Coordination'10*, volume 6116 of *LNCS*, pages 152–167. Springer, 2010.

[30] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS*, volume 5356 of *LNCS*, pages 155–170, 2008.

[31] 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, 1994.

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

[33] H. Viera. *A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing*. PhD thesis, University Nova de Lisboa, 2010.

[34] N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCS*, pages 371–386, 1996.

[35] N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCs*, volume 6014 of *LNCS*, pages 128–145, 2010.

[36] N. Yoshida, V. T. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *FMCO'08*, volume 5751 of *LNCS*, pages 226–246. Springer, 2009.