Resource Sharing via Capability-Based Multiparty Session Types ⋆

A. Laura Voinea[0000−0003−4482−205X], Ornela Dardha[0000−0001−9927−7875], and Simon J. Gay[0000−0003−3033−9091]

School of Computing Science, University of Glasgow, United Kingdom
a.voinea.1@research.gla.ac.uk
{Ornela.Dardha, Simon.Gay}@glasgow.ac.uk

Abstract. *Multiparty Session Types (MPST)* are a type formalism used to model communication protocols among components in distributed systems, by specifying *type* and *direction* of data transmitted. It is standard for multiparty session type systems to use access control based on *linear* or *affine* types. While useful in offering strong guarantees of communication safety and session fidelity, linearity and affinity run into the well-known problem of inflexible programming, excluding scenarios that make use of shared channels or need to store channels in shared data structures.
In this paper, we develop *capability-based resource sharing* for multiparty session types. In this setting, channels are split into two entities, the channel itself and the capability of using it. This gives rise to a more flexible session type system, which allows channel references to be shared and stored in persistent data structures. We illustrate our type system through a producer-consumer case study. Finally, we prove that the resulting language satisfies type safety.

Keywords: session types · sharing · concurrent programming

1 Introduction

In the present era of communication-centric software systems, it is increasingly recognised that the structure of communication is an essential aspect of system design. *(Multiparty) session types* [18,19,31] allow communication structures to be codified as type definitions in programming languages, which can be exploited by compilers, development environments and runtime systems, for compile-time analysis or runtime monitoring. A substantial and ever-growing literature on session types and, more generally, behavioural types [20] provides a rich theoretical foundation, now being applied to a range of programming languages [1,16].

⋆ Supported by the UK EPSRC grant EP/K034413/1, "From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD)", by the EU HORIZON 2020 MSCA RISE project 778233 "BehAPI: Behavioural Application Program Interfaces", and by an EPSRC PhD studentship.

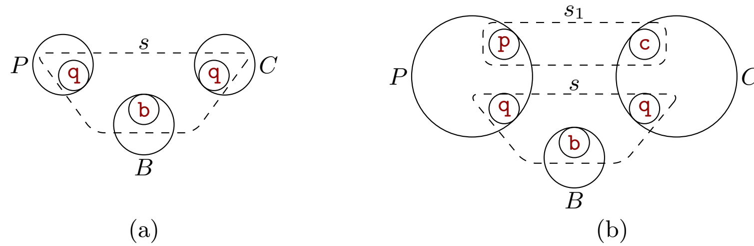

Fig. 1. Producer-consumer system: Producer—P and Consumer—C sharing access to Buffer—B by implementing the same role q. (a) P and C communicate with B in session s; (b) P and C exchange the capability to use channel $\mathbf{s}[\mathtt{q}]$ in session s_1 and then use it to communicate with B in session s

Session type systems must control *aliasing* of the endpoints of communication channels, in order to avoid race conditions. If agents A and B both think they are running the client side of a protocol with the same server S, then a message sent by A advances the session state without B's knowledge, which interferes with B's attempt to run the protocol.

In order to guarantee unique ownership of channel endpoints and eliminate aliasing, most session type systems use strict *linear typing*. For more flexibility, some others use *affine typing*, which allows channels to be discarded, but they still forbid aliasing. It is possible to allow a session-typed channel to become sharable in the special case in which the session type reaches a point which is essentially stateless. However, in such systems, channels are *linearly typed* for the most interesting parts of their lifetimes—we discuss these possibilities in § 6.

This leads us to our research questions:

Q1 *Are session types intrinsically related to linearity or affinity?*
Q2 *Can we define session type systems without linear types?*
Q3 *How can we check resource (channel) sharing and aliasing, to guarantee communication safety and session fidelity, i.e., type safety?*

The goal of this paper is to investigate questions **Q1**–**Q3**. To give a more flexible approach to resource sharing and access control, we propose a system of multiparty session types (MPST) that includes techniques from the Capability Calculus [11], and from Walker et al.'s work on alias types [35]. The key idea is to split a communication channel into two entities: (1) the channel itself, and (2) its usage *capability*. Both entities are first-class and can be referred to separately. Channels can now be shared, or stored in shared data structures, and aliasing is allowed. However, in order to guarantee communication safety and session fidelity, i.e., type safety, capabilities are used linearly so that only one alias can be used at a time.

This approach has several benefits, and improves on the state of the art: (i) for the first time, it is now possible for a system to have a communication structure defined by shared channels, with the capabilities being transferred from process to process as required; (ii) a capability can be implemented as a simple token,

whereas delegation of channels requires a relatively complex implementation, thus making linearity of capabilities more lightweight than linearity of channels.

Example 1 (Producer-Consumer Fig. 1). Producer P and consumer C communicate via buffer B in session $\mathtt{s}$, given in Fig. 1 (a). P and C implement *role* $\mathtt{q}$, and B implements role $\mathtt{b}$. *Shared access* to buffer B is captured by the fact that both P and C implement the same role $\mathtt{q}$ and use the same channel $\mathtt{s}[\mathtt{q}]$ to communicate with B.

Following MPST theory, we start by defining a *global type*, describing communications among *all* participants:

$$G_0 = \mathtt{q}{\to}\mathtt{b}{:}\mathsf{add}(\mathtt{Int}).\ \mathtt{q}{\to}\mathtt{b}{:}\mathsf{request}().\ \mathtt{b}{\to}\mathtt{q}{:}\mathsf{send}(\mathtt{Int}).\ G_0$$

In G_0, protocol proceeds as follows: P (playing $\mathtt{q}$) sends an add message to B (playing $\mathtt{b}$), to add data. In sequence, C (playing $\mathtt{q}$) sends a $\mathsf{request}$ message to B, to ask for data. B replies with a send message, sending data to C (playing $\mathtt{q}$), and the protocol repeats as G_0. *Projecting* the global protocol to each role gives us a local session type. In particular, for B, implementing role $\mathtt{b}$, we obtain:

$$S_{\mathtt{b}} = \mathtt{q}?\mathsf{add}(\mathtt{Int}).\ \mathtt{q}?\mathsf{request}().\ \mathtt{q}!\mathsf{send}(\mathtt{Int}).\ S_{\mathtt{b}}$$

where the $\mathtt{q}$ annotations show the other role participating in each interaction. For the shared access by P and C (role $\mathtt{q}$), we obtain:

$$S_{\mathtt{q}} = \mathtt{b}!\mathsf{add}(\mathtt{Int}).\ S'_{\mathtt{q}} \qquad\qquad S'_{\mathtt{q}} = \mathtt{b}!\mathsf{request}().\ \mathtt{b}?\mathsf{send}(\mathtt{Int}).\ S_{\mathtt{q}}$$

Finally, the definitions of processes are as follows—we will detail the syntax in § 2.

$$\begin{aligned}
\mathsf{P}\langle v\rangle &= \mathsf{s}[\mathtt{q}][\mathtt{b}]\oplus\langle\mathsf{add}(v)\rangle.\mathsf{P}\langle v+1\rangle\\
\mathsf{C}\langle\rangle &= \mathsf{s}[\mathtt{q}][\mathtt{b}]\oplus\langle\mathsf{request}()\rangle.\mathsf{s}[\mathtt{b}][\mathtt{q}]\&\{\mathsf{send}(\mathsf{i})\}.\mathsf{C}\langle\rangle\\
\mathsf{B}\langle\rangle &= \mathsf{s}[\mathtt{q}][\mathtt{b}]\&\{\mathsf{add}(x)\}.\mathsf{s}[\mathtt{q}][\mathtt{b}]\&\{\mathsf{request}()\}.\mathsf{s}[\mathtt{b}][\mathtt{q}]\oplus\langle\mathsf{send}(x)\rangle.\mathsf{B}\langle\rangle
\end{aligned}$$

Unfortunately, the system of processes above is not typable using standard multiparty session type systems because role $\mathtt{q}$ is shared by P and C, thus violating linearity of channel $\mathsf{s}[\mathtt{q}]$. To solve this issue and still allow sharing and aliasing, in our work, instead of associating a channel c with a session type S, we separately associate c with a *tracked* type $\mathtt{tr}(\rho)$, and S with *capability* ρ, $\{\rho\mapsto S\}$. The capability can be passed between P and C as they take turns in using the channel, illustrated in Fig. 1 (b). As a first attempt, we now define the following global type, getting us closer to our framework.

$$\begin{aligned}
G_1 = {}&\mathtt{q}{\to}\mathtt{b}{:}\mathsf{add}(\mathtt{Int}).\ \mathtt{p}{\to}\mathtt{c}{:}\mathsf{turn}(\mathtt{tr}(\rho_{\mathtt{q}})).\ \mathtt{q}{\to}\mathtt{b}{:}\mathsf{request}().\\
&\mathtt{b}{\to}\mathtt{q}{:}\mathsf{send}(\mathtt{Int}).\ \mathtt{c}{\to}\mathtt{p}{:}\mathsf{turn}(\mathtt{tr}(\rho_{\mathtt{q}})).G_1
\end{aligned}$$

However, a type such as $\mathtt{tr}(\rho_{\mathtt{q}})$ is usually too specific because it refers to the capability of a particular channel. It is preferable to be able to give definitions that abstract away from specific channels. We therefore introduce *existential types*, in the style of [35], which package a channel with its capability, in the form $\exists[\rho|\{\rho\mapsto S\}].\mathtt{tr}(\rho)$.

With the existential types in place, we can define our global type G in the following way. It now includes an extra initial message from P to C containing

$P ::= \mathbf{0} \mid P \mid Q \mid (\nu\, s)P$	inaction, parallel composition, restriction
$\mid \mathsf{c}[\mathsf{p}]\oplus\langle l(v)\rangle.P \mid \mathsf{c}[\mathsf{p}]\&_{i\in I}\{l_i(x_i).\,P_i\}$	select, branch
$\mid \mathsf{c}[\mathsf{p}]\oplus\langle l(\mathsf{pack}(\rho, \mathsf{s}[\mathsf{q}]))\rangle.P$	select pack
$\mid \mathsf{c}[\mathsf{p}]\&_{i\in I}\{l_i(\mathsf{pack}(\rho_i, \mathtt{s_i}[\mathsf{q}])).\,P_i\}$	branch pack
$\mid \mathsf{def}\, D \,\mathsf{in}\, P \mid X\langle\tilde{x}\rangle$	recursion, process call
$D ::= X\langle\tilde{x}\rangle = P$	process declaration
$\mathsf{c} ::= x \mid \mathsf{s}[\mathsf{p}]$	variable, channel with role p
$v ::= \mathsf{c} \mid \rho \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots$	channel, capability, base value

Fig. 2. Multiparty session π-calculus

the channel used with the buffer. The session types $S_{\mathtt{q}}$ and $S'_{\mathtt{q}}$ are the same as before.

$$G = \mathsf{p}\to\mathsf{c}{:}\mathsf{buffer}(\exists[\rho_{\mathsf{q}}|\{\rho_{\mathsf{q}}\mapsto S'_{\mathsf{q}}\}].\mathtt{tr}(\rho_{\mathsf{q}})).\mu\,\mathsf{t}.\mathsf{q}\to\mathsf{b}{:}\mathsf{add}(\mathtt{Int}).\mathsf{p}\to\mathsf{c}{:}\mathsf{turn}(\{\rho_{\mathsf{q}}\mapsto S'_{\mathsf{q}}\}).$$
$$\mathsf{q}\to\mathsf{b}{:}\mathsf{request}().\mathsf{b}\to\mathsf{q}{:}\mathsf{send}(\mathtt{Int}).\mathsf{c}\to\mathsf{p}{:}\mathsf{turn}(\{\rho_{\mathsf{q}}\mapsto S_{\mathsf{q}}\}).\mathsf{t}$$

In § 4 we complete this example by showing the projections to a local type for each role, and the definitions of processes that implement each role. □

Contributions of the paper

(i) **MPST with capabilities**: we present a new version of MPST theory without linear typing for channels, but with linearly-typed capabilities. In § 2 we define a multiparty session π-calculus with capabilities, and its operational semantics, and in § 3 we define a MPST system for it.

(ii) **Producer-Consumer Case Study**: in § 4 we present a detailed account of the producer-consumer case study, capturing the core of resource sharing and use of capabilities.

(iii) **Type Safety**: in § 5 we state the type safety property, and outline its proof.

2 Multiparty Session π-Calculus with Capabilities

Our π-calculus with multiparty session types is based on the language defined by Scalas *et al.* [28]. The syntax is defined in Fig. 2. We assume infinite sets of identifiers for variables (x), sessions ($\mathtt{s}$), capabilities (ρ) and roles ($\mathtt{p}$).

The calculus combines branch (resp., select) with input (resp., output), and a message $l(v)$ consists of a label l and a payload v, which is a value. A message in session $\mathtt{s}$ from role $\mathtt{p}$ to role $\mathtt{q}$ has the prefix $\mathtt{s}[\mathsf{p}][\mathsf{q}]$, where $\mathtt{s}[\mathsf{p}]$ is represented by $\mathtt{c}$ in the grammar. The select and branch operations come in two forms. The first form is standard, and the second form handles *packages*, which are the novel feature of our type system. A package consists of a capability ρ and a channel of type $\mathtt{tr}(\rho)$. We will see in § 3 in the typing rules, the capability is existentially

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

$$(\nu\, s)\mathbf{0} \equiv \mathbf{0} \qquad (\nu\, s)(\nu\, s')P \equiv (\nu\, s')(\nu\, s)P \qquad (\nu\, s)P \mid Q \equiv (\nu\, s)(P \mid Q) \;\; \text{if } s \notin fc(Q)$$

$$\mathsf{def}\, D \,\mathsf{in}\, \mathbf{0} \equiv \mathbf{0} \qquad \mathsf{def}\, D \,\mathsf{in}\, (\nu\, s)P \equiv (\nu\, s)(\mathsf{def}\, D \,\mathsf{in}\, P) \;\; \text{if } s \notin fc(P)$$

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

$$\mathsf{def}\, D \,\mathsf{in}\, \mathsf{def}\, D' \,\mathsf{in}\, P \equiv \mathsf{def}\, D' \,\mathsf{in}\, \mathsf{def}\, D \,\mathsf{in}\, P$$
$$\text{if } (dpv(D) \cup fpv(D)) \cap dpv(D') = (dpv(D') \cup fpv(D')) \cap dpv(D) = \emptyset$$

Fig. 3. Structural congruence (processes)

$$\frac{j \in I \text{ and } fv(v) = \emptyset}{\mathsf{s}[\mathtt{p}][\mathtt{q}]\&_{i\in I}\{l_i(x_i).\, P_i\} \mid \mathsf{s}[\mathtt{q}][\mathtt{p}]\oplus\langle l_j(v)\rangle.P \;\longrightarrow\; P_j\{v/x_j\} \mid P} \;\textsc{RCom}$$

$$\frac{j \in I}{\mathsf{s}[\mathtt{p}][\mathtt{q}]\&_{i\in I}\{l_i(\mathsf{pack}(\rho_i, \mathtt{v}_i)).\, P_i\} \mid \mathsf{s}[\mathtt{q}][\mathtt{p}]\oplus\langle l_j(\mathsf{pack}(\rho, \mathtt{v}))\rangle.P \;\longrightarrow\; P_j\{\mathtt{v}/\mathtt{v}_i\} \mid P} \;\textsc{RComP}$$

$$\frac{\widetilde{x} = x_1, \ldots, x_n \qquad \widetilde{v} = v_1, \ldots, v_n \qquad fv(\widetilde{v}) = \emptyset}{\mathsf{def}\, X\langle\tilde{x}\rangle = P \,\mathsf{in}\, (X\langle\tilde{x}\rangle \mid Q) \;\longrightarrow\; \mathsf{def}\, X\langle\tilde{x}\rangle = P \,\mathsf{in}\, (P\{\tilde{v}/\tilde{x}\} \mid Q)} \;\textsc{RCall}$$

$$\frac{P \;\longrightarrow\; Q}{(\nu\, s)P \;\longrightarrow\; (\nu\, s)Q} \;\textsc{RRes} \qquad \frac{P \;\longrightarrow\; Q}{P \mid R \;\longrightarrow\; Q \mid R} \;\textsc{RPar} \qquad \frac{P \;\longrightarrow\; Q}{\mathsf{def}\, D \,\mathsf{in}\, P \;\longrightarrow\; \mathsf{def}\, D \,\mathsf{in}\, Q} \;\textsc{RDef}$$

Fig. 4. Reduction (processes)

quantified. This enables a channel to be delegated, with the information that it is linked to *some* capability, which will be transmitted in a second message.

As usual, we define structural congruence to compensate for the limitations of textual syntax. It is the smallest congruence satisfying the axioms in Fig. 3. The definition uses the concepts of *free channels* of a process, $fc(P)$; *free process variables* of a process, $fpv(P)$; and *defined process variables* of a process declaration, $dpv(D)$. We omit the definitions of these concepts, which are standard and can be found in [28].

We define a reduction-based operational semantics by the rules in Fig. 4. Rule RCom is a standard communication between roles $\mathtt{p}$ and $\mathtt{q}$. Rule RComP is communication of an existential package. Rule RCall defines a standard approach to handling process definitions. The rest are standard contextual rules.

3 Multiparty Session Types with Capabilities

We now introduce a type system for the multiparty session π-calculus. The general methodology of multiparty session types is that system design begins with a *global type*, which specifies all of the communication among various *roles*. Given a global type G and a role $\mathtt{p}$, *projection* yields a *session type* or *local type* $G \upharpoonright \mathtt{p}$ that describes all of the communication involving $\mathtt{p}$. This local type can be further projected for another role $\mathtt{q}$, to give a *partial session type* that describes communication between $\mathtt{p}$ and $\mathtt{q}$.

S ::=		*local session type*
	$\mathtt{end}$	terminated session
|	$\mathtt{p}\oplus_{i\in I}!l_i(U_i).S_i$	selection towards role p
|	$\mathtt{p}\&_{i\in I}?l_i(U_i).S_i$	branching from role p
|	$\mathtt{p}\oplus_{i\in I}!l_i(\exists[\rho_i\|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i$	pack selection towards role p
|	$\mathtt{p}\&_{i\in I}?l_i(\exists[\rho_i\|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i$	pack branching from role p
|	$\mathtt{t}$	type variable
|	$\mu\,\mathtt{t}.S$	recursive session type
G ::=		*global type*
	$\mathtt{end}$	termination
|	$\mathtt{p}\to\mathtt{q}:\{l_i(U_i).G_i\}_{i\in I}$	interaction
|	$\mathtt{p}\to\mathtt{q}:\{l_i(\exists[\rho_i\|\{\rho_i\mapsto U_i\}].tr(\rho_i)).G_i\}_{i\in I}$	pack interaction
|	$\mathtt{t}$	type variable
|	$\mu\,\mathtt{t}.G$	recursive type
H ::=		*partial session type*
	$\mathtt{end}$	terminated session
|	$\oplus_{i\in I}!l_i(U_i).H_i$	selection
|	$\&_{i\in I}?l_i(U_i).H_i$	branching
|	$\oplus_{i\in I}!l_i(\exists[\rho_i\|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).H_i$	pack selection
|	$\&_{i\in I}?l_i(\exists[\rho_i\|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).H_i$	pack branching
|	$\mathtt{t}$	type variable
|	$\mu\,\mathtt{t}.H$	recursive type
C ::=	$\emptyset \mid C\otimes\{\rho\mapsto S\}$	*capabilities*
B ::=	$\mathtt{Int} \mid \mathtt{Bool}$	*ground type*
U ::=		*payload type*
	B	ground type
|	$\mathtt{tr}(\rho)$	tracked type
|	$\{\rho\mapsto S\}$	capability type
|	S closed	session type
Γ ::=	$\emptyset \mid \Gamma, x:U \mid \Gamma, \mathtt{s}[\mathtt{p}]:\mathtt{tr}(\rho)$	*environment*
Δ ::=	$\emptyset \mid \Delta, X:\tilde{U}$	*process names*

All branch and select types have the conditions $I \neq \emptyset$ and U_i closed.

Fig. 5. Types, capabilities, environments

Global types Fig. 5. Each interaction has a source role $\mathtt{p}$ and a target role $\mathtt{q}$. We combine branching and message transmission, so an interaction has a label l_i, a payload of type U_i, or of type $\exists[\rho_i|\{\rho_i \mapsto U_i\}].tr(\rho_i)$, and a continuation type G_i. If there is only one branch then we usually abbreviate the syntax to $\mathtt{p}{\to}\mathtt{q}{:}l(U).\ G$, respectively $\mathtt{p}{\to}\mathtt{q}{:}l(\exists[\rho_i|\{\rho_i \mapsto U_i\}].tr(\rho_i)).\ G$. Recursive types are allowed, with the assumption that they are guarded. **Base types** $B, B', \ldots$ can be types like $\mathtt{Bool}$, $\mathtt{Int}$, etc. **Payload types** $U, U_i, \ldots$ are either base types, tracked types, capability types or *closed* session types.

Local (session) types Fig. 5. The single form of interaction from global types splits into *select* (internal choice) and *branch* (external choice). The *branching type* $\mathtt{p}\&_{i\in I}?l_i\,(U_i)\,.S_i$ describes a channel that can receive a label l_i from role $\mathtt{p}$ (for some $i \in I$, chosen by $\mathtt{p}$), together with a *payload* of type U_i; then, the channel must be used as the continuation type S_i. The *selection type* $\mathtt{p}\oplus_{i\in I}!l_i(U_i).S_i$, describes a channel that can choose a label l_i (for any $i \in I$), and send it to $\mathtt{p}$ together with a payload of type U_i; then, the channel must be used as S_i. The types for *pack select* and *pack branch* act in a similar manner, and bind the capability ρ_i for the continuation type S_i. Session types also allow guarded recursion.

The relationship between global types and session types is formalised by the notion of projection.

Definition 1. *The projection of G onto a role $\mathtt{q}$, written $G \upharpoonright \mathtt{q}$, is:*

$$\mathtt{end} \upharpoonright \mathtt{q} \triangleq \mathtt{end} \qquad \mathtt{t} \upharpoonright \mathtt{q} \triangleq \mathtt{t} \qquad (\mu\,\mathtt{t}.G) \upharpoonright \mathtt{q} \triangleq \begin{cases} \mu\,\mathtt{t}.(G \upharpoonright \mathtt{q}) & \text{if } G \upharpoonright \mathtt{q} \neq \mathtt{t}'\ (\forall \mathtt{t}') \\ \mathtt{end} & \text{otherwise} \end{cases}$$

$$(\mathtt{p}{\to}\mathtt{p}'{:}\{l_i(U_i).G_i\}_{i\in I}) \upharpoonright \mathtt{q} \triangleq \begin{cases} \mathtt{p}' \oplus_{i\in I} !l_i(U_i).(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ \mathtt{p}\ \&_{i\in I}\ ?l_i(U_i).(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}', \\ \prod_{i\in I}(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{p} \neq \mathtt{q} \neq \mathtt{p}' \end{cases}$$

$$(\mathtt{p}{\to}\mathtt{p}'{:}\{l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).G_i\}_{i\in I}) \upharpoonright \mathtt{q} \triangleq$$
$$\begin{cases} \mathtt{p}' \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ \mathtt{p}\ \&_{i\in I}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}', \\ \prod_{i\in I}(G_i \upharpoonright \mathtt{q}) & \text{if } \mathtt{p} \neq \mathtt{q} \neq \mathtt{p}' \end{cases}$$

Where the merge operator for session types, $\prod$, *is defined by:*

$$\mathtt{end} \sqcap \mathtt{end} \triangleq \mathtt{end} \qquad \mathtt{t} \sqcap \mathtt{t} \triangleq \mathtt{t} \qquad \mu\,t.S \sqcap \mu\,t.S' \triangleq \mu\,t.(S \sqcap S')$$

$$\mathtt{p}\ \&_{i\in I}\ ?l_i(U_i).S_i \sqcap \mathtt{p}\ \&_{j\in J}\ ?l_j(U_j).S'_j \triangleq$$
$$\mathtt{p}\ \&_{k\in I\cap J}\ ?l_k(U_k).(S_k \sqcap S'_k)\ \&\ \mathtt{p}\ \&_{j\in I\setminus J}\ ?l_i(U_i).S_i\ \&\ \mathtt{p}\ \&_{j\in J\setminus I}\ ?l_j(U_j).S'_j$$

$$\mathtt{p} \oplus_{i\in I} !l_i(U_i).S_i \sqcap \mathtt{p} \oplus_{i\in I} !l_i(U_i).S_i \triangleq \mathtt{p} \oplus_{i\in I} !l_i(U_i).S_i$$

$$\mathtt{p}\ \&_{i\in I}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i \sqcap \mathtt{p}\ \&_{j\in J}\ ?l_j(\exists[\rho_j|\{\rho_j \mapsto U_j\}].\mathtt{tr}(\rho_j)).S'_j \triangleq$$
$$\mathtt{p}\ \&_{k\in I\cap J}\ ?l_k(\exists[\rho_k|\{\rho_k \mapsto U_k\}].\mathtt{tr}(\rho_k)).(S_k \sqcap S'_k)\ \&\ \mathtt{p}\ \&_{j\in I\setminus J}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i$$
$$\&\ \mathtt{p}\ \&_{j\in J\setminus I}\ ?l_j(\exists[\rho_j|\{\rho_j \mapsto U_j\}].\mathtt{tr}(\rho_j)).S'_j$$

$$\mathtt{p} \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i \sqcap \mathtt{p} \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i \triangleq$$
$$\mathtt{p} \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i$$

Projecting $\mathtt{end}$ or a type variable $\mathtt{t}$ onto any role does not change it. Projecting a recursive type $\mu\,\mathtt{t}.G$ onto $\mathtt{q}$ means projecting G onto $\mathtt{q}$. However, if G does

not involve $\mathtt{q}$ then $G\upharpoonright\mathtt{q}$ is a type variable, $\mathtt{t}'$, and it must be replaced by $\mathtt{end}$ to avoid introducing an unguarded recursive type. Projecting an interaction between $\mathtt{p}$ and $\mathtt{p}'$ onto either $\mathtt{p}$ or $\mathtt{p}'$ produces a select or a branch. Projecting onto a different role $\mathtt{q}$ ignores the interaction and combines the projections of the continuations using the merge operator.

The merge operator, $\sqcap$, introduced in [13,36], allows more global types to have defined projections, which in turn allows more processes to be typed. Different external choices from the same role $\mathtt{p}$ are integrated by merging the continuation types following a common message label, and including the branches with different labels. Merging for internal choices is undefined unless the interactions are identical. This excludes meaningless types that result when a sender $\mathtt{p}$ is unaware of which branch has been chosen by other roles in a previous interaction.

Definition 2. *For a session type S, $\mathtt{roles}(S)$ denotes the set of roles occurring in S. We write $\mathtt{p}\in S$ for $\mathtt{p}\in\mathtt{roles}(S)$, and $\mathtt{p}\in S\backslash\mathtt{q}$ for $\mathtt{p}\in\mathtt{roles}(S)\setminus\{\mathtt{q}\}$.*

Partial session types Fig. 5 have the same cases as local types, without role annotations. Partial types have a notion of *duality* which exchanges branch and select but preserves payload types.

Definition 3. *$\overline{H}$ is the* dual of *H, defined by:*

$$\overline{\oplus_{i\in I}!l_i(U_i).H_i} \triangleq \&_{i\in I}\,?l_i(U_i).\overline{H_i} \qquad \overline{\&_{i\in I}?l_i\,(U_i)\,.H_i} \triangleq \oplus_{i\in I}\,!l_i(U_i).\overline{H_i}$$

$$\overline{\oplus_{i\in I}\,!l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).H_i} \triangleq \&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).\overline{H_i}$$

$$\overline{\&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).H_i} \triangleq \oplus_{i\in I}\,!l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).\overline{H_i}$$

$$\overline{\mathtt{end}} \triangleq \mathtt{end} \qquad \overline{\mathtt{t}} \triangleq \mathtt{t} \qquad \overline{\mu\,t.H} \triangleq \mu\,t.\overline{H}$$

Similarly to the projection of global types to local types, a local type can be projected onto a role $\mathtt{q}$ to give a partial type. This yields a partial type that only describes the communications in S that involve $\mathtt{q}$. The definition follows the same principles as the previous definition (cf. Definition 1).

Definition 4. *$S\upharpoonright\mathtt{q}$ is the* partial projection of *S* onto *$\mathtt{q}$:*

$$\mathtt{end}\upharpoonright\mathtt{q} \triangleq \mathtt{end} \qquad \mathtt{t}\upharpoonright\mathtt{q} \triangleq \mathtt{t} \qquad (\mu\,\mathtt{t}.S)\upharpoonright\mathtt{q} \triangleq \begin{cases}\mu\,\mathtt{t}.(S\upharpoonright\mathtt{q}) & \textit{if } S\upharpoonright\mathtt{q}\neq\mathtt{t}'\ (\forall\mathtt{t}')\\ \mathtt{end} & \textit{otherwise}\end{cases}$$

$$(\mathtt{p}\oplus_{i\in I}!l_i(U_i).S_i)\upharpoonright\mathtt{q} \triangleq \begin{cases}\oplus_{i\in I}\,!l_i(U_i).(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{q}=\mathtt{p},\\ \bigsqcap_{i\in I}(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{p}\neq\mathtt{q}\end{cases}$$

$$(\mathtt{p}\&_{i\in I}?l_i\,(U_i)\,.S_i)\upharpoonright\mathtt{q} \triangleq \begin{cases}\&_{i\in I}\,?l_i(U_i).S_i\upharpoonright\mathtt{q} & \textit{if } \mathtt{q}=\mathtt{p},\\ \bigsqcap_{i\in I}(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{p}\neq\mathtt{q}\end{cases}$$

$$(\mathtt{p}\oplus_{i\in I}\,!l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i)\upharpoonright\mathtt{q} \triangleq \begin{cases}\oplus_{i\in I}\,!l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{q}=\mathtt{p},\\ \bigsqcap_{i\in I}(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{p}\neq\mathtt{q}\end{cases}$$

$$(\mathtt{p}\,\&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i)\upharpoonright\mathtt{q} \triangleq \begin{cases}\&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i\upharpoonright\mathtt{q} & \textit{if } \mathtt{q}=\mathtt{p},\\ \bigsqcap_{i\in I}(S_i\upharpoonright\mathtt{q}) & \textit{if } \mathtt{p}\neq\mathtt{q}\end{cases}$$

Where the merge operator for partial session types, $\sqcap$, *is defined by:*

$$\texttt{end} \sqcap \texttt{end} \triangleq \texttt{end} \qquad \texttt{t} \sqcap \texttt{t} \triangleq \texttt{t} \qquad \mu\,t.H \sqcap \mu\,t.H' \triangleq \mu\,t.(H \sqcap H')$$

$$\&_{i\in I}\,?l_i(U_i).H_i \;\sqcap\; \&_{i\in I}\,?l_i(U_i).H'_i \;\triangleq\; \&_{i\in I}\,?l_i(U_i).(H_i \sqcap H'_i)$$

$$\oplus_{i\in I}\,!l_i(U_i).H_i \;\sqcap\; \oplus_{j\in J}\,!l_j(U_j).H'_j \;\triangleq\; \left(\oplus_{k\in I\cap J}\,!l_k(U_k).(H_k \sqcap H'_k)\right) \oplus \left(\oplus_{i\in I\setminus J}\,!l_i(U_i).H_i\right) \oplus \left(\oplus_{j\in J\setminus I}\,!l_j(U_j).H'_j\right)$$

$$\&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\texttt{tr}(\rho_i)).H_i \;\sqcap\; \&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\texttt{tr}(\rho_i)).H'_i \;\triangleq\; \&_{i\in I}\,?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\texttt{tr}(\rho_i)).(H_i \sqcap H'_i)$$

$$\oplus_{i\in I}\,!l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\texttt{tr}(\rho_i)).H_i \;\sqcap\; \oplus_{j\in J}\,!l_j(\exists[\rho_j|\{\rho_j \mapsto U_j\}].\texttt{tr}(\rho_j)).H'_j \;\triangleq\; \left(\oplus_{k\in I\cap J}\,!l_k(\exists[\rho_k|\{\rho_k \mapsto U_k\}].\texttt{tr}(\rho_k)).(H_k \sqcap H'_k)\right) \oplus \left(\oplus_{i\in I\setminus J}\,!l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\texttt{tr}(\rho_i)).H_i\right) \oplus \left(\oplus_{j\in J\setminus I}\,!l_j(\exists[\rho_j|\{\rho_j \mapsto U_j\}].\texttt{tr}(\rho_j)).H'_j\right)$$

Unlike session type merging, $\sqcap$ can combine different *internal* choices, but *not* external choices because that could violate type safety. Different internal choices can depend on the outcome of previous interactions with other roles, since this dependency can be safely approximated as an internal choice. Different external choices, however cannot capture this dependency.

Example 2 (Projections of Global and Local Types). Consider the global type G of the producer-consumer example from the introduction.

$$G = \mathsf{p}\to\mathsf{c}{:}\mathsf{buffer}(\exists[\rho_\mathsf{q}|\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}].\texttt{tr}(\rho_\mathsf{q})).\mu\,\mathsf{t}.\mathsf{q}\to\mathsf{b}{:}\mathsf{add}(\texttt{Int}).\mathsf{p}\to\mathsf{c}{:}\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}).\mathsf{q}\to\mathsf{b}{:}\mathsf{request}(\texttt{Str}).\mathsf{b}\to\mathsf{q}{:}\mathsf{send}(\texttt{Int}).\mathsf{c}\to\mathsf{p}{:}\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S_\mathsf{q}\}).\mathsf{t}$$

It captures the interaction between the producer and consumer entities through roles $\texttt{p}$, $\texttt{c}$, and between producer, consumer and buffer through roles $\texttt{q}$ (shared between producer and consumer) and $\texttt{b}$. Projecting onto $\texttt{p}$ gives the session type

$$S = G\restriction\mathsf{p} = \mathsf{c}\oplus!\mathsf{buffer}(\exists[\rho_\mathsf{q}|\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}].\texttt{tr}(\rho_\mathsf{q})).\mu\,\mathsf{t}.\mathsf{c}\oplus!\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}).\mathsf{c}\&?\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S_\mathsf{q}\}).\mathsf{t}$$

and further projecting onto $\texttt{c}$ gives the partial session type:

$$H = S\restriction\mathsf{c} = \oplus!\mathsf{buffer}(\exists[\rho_\mathsf{q}|\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}].\texttt{tr}(\rho_\mathsf{q})).!\mu\,\mathsf{t}.\oplus!\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S'_\mathsf{q}\}).\&?\mathsf{turn}(\{\rho_\mathsf{q} \mapsto S_\mathsf{q}\}).\mathsf{t}$$

Definition 5 (Subtyping). Subtyping on session types $\leqslant_\mathsf{S}$ *is the largest relation such that* (i) *if* $S \leqslant_\mathsf{S} S'$, *then* $\forall \mathsf{p} \in (\texttt{roles}(S) \cup \texttt{roles}(S'))\ S\restriction\mathsf{p} \leqslant_\mathsf{P} S'\restriction\mathsf{p}$, *and* (ii) *is closed backwards under the coinductive rules in Fig. 6.* Subtyping on partial session types $\leqslant_\mathsf{P}$ *is defined coinductively by the rules in Fig. 7.*

Intuitively, the *subtyping relation* says that a session type S is "smaller" than S' when S is "less demanding" than S' i.e., when S allows more internal choices, and imposes fewer external choices, than S'. Clause *(i)* links local and partial subtyping, and ensures that if two types are related, then their partial projections exist. This clause is used later in defining consistency in Definition 8. In the second clause *(ii)* rules SBr, SSel define subtyping on branch/select types, and SBrP, SSelP define subtyping on branch pack/select pack types. SBr and SBrP are covariant in their continuation types as well as in the number of branches offered, whereas SSel, and SSelP are contravariant in both. SB relates base types,

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p} \&_{i\in I} ?l_i\,(U_i)\,.S_i \leqslant_{\mathsf{S}} \mathtt{p}\ \&_{i\in I\cup J}\ ?l_i(U_i').S_i'}\ \textsc{SBr} \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p} \oplus_{i\in I\cup J} !l_i(U_i).S_i \leqslant_{\mathsf{S}} \mathtt{p} \oplus_{i\in I} !l_i(U_i').S_i'}\ \textsc{SSel}$$

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p}\ \&_{i\in I}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i \leqslant_{\mathsf{S}} \mathtt{p}\ \&_{i\in I\cup J}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i'\}].\mathtt{tr}(\rho_i)).S_i'}\ \textsc{SBrP}$$

$$\frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p} \oplus_{i\in I\cup J} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i \leqslant_{\mathsf{S}} \mathtt{p} \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i'\}].\mathtt{tr}(\rho_i)).S_i'}\ \textsc{SSel}$$

$$\frac{B \leqslant_{\mathsf{B}} B'}{B \leqslant_{\mathsf{S}} B'}\ \textsc{SB} \qquad \frac{}{\mathtt{end} \leqslant_{\mathsf{S}} \mathtt{end}}\ \textsc{SEnd} \qquad \frac{S\{\mu\,\mathtt{t}.S/\mathtt{t}\} \leqslant_{\mathsf{S}} S'}{\mu\,\mathtt{t}.S \leqslant_{\mathsf{S}} S'}\ \textsc{S}\mu\textsc{L} \qquad \frac{S \leqslant_{\mathsf{S}} S'\{\mu\,\mathtt{t}.S'/\mathtt{t}\}}{S \leqslant_{\mathsf{S}} \mu\,\mathtt{t}.S'}\ \textsc{S}\mu\textsc{R}$$

Fig. 6. Subtyping for local session types.

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad H_i \leqslant_{\mathsf{P}} H_i'}{\&_{i\in I} ?l_i\,(U_i)\,.H_i \leqslant_{\mathsf{P}} \&_{i\in I\cup J}\ ?l_i(U_i').H_i'}\ \textsc{SParBr} \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad H_i \leqslant_{\mathsf{P}} H_i'}{\oplus_{i\in I\cup J} !l_i(U_i).H_i \leqslant_{\mathsf{P}} \oplus_{i\in I} !l_i(U_i').H_i'}\ \textsc{SParSel}$$

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad H_i \leqslant_{\mathsf{P}} H_i'}{\&_{i\in I}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).H \leqslant_{\mathsf{P}} \&_{i\in I\cup J}\ ?l_i(\exists[\rho_i|\{\rho_i \mapsto U_i'\}].\mathtt{tr}(\rho_i)).H_i'}\ \textsc{SParBrP}$$

$$\frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad H_i \leqslant_{\mathsf{P}} H_i'}{\oplus_{i\in I\cup J} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).H_i \leqslant_{\mathsf{P}} \oplus_{i\in I} !l_i(\exists[\rho_i|\{\rho_i \mapsto U_i'\}].\mathtt{tr}(\rho_i)).H_i'}\ \textsc{SParSelP}$$

$$\frac{}{\mathtt{end} \leqslant_{\mathsf{P}} \mathtt{end}}\ \textsc{SParEnd} \qquad \frac{H\{\mu\,\mathtt{t}.H/\mathtt{t}\} \leqslant_{\mathsf{P}} H'}{\mu\,\mathtt{t}.H \leqslant_{\mathsf{P}} H'}\ \textsc{SPar}\mu\textsc{L} \qquad \frac{H \leqslant_{\mathsf{P}} H'\{\mu\,\mathtt{t}.H'/\mathtt{t}\}}{H \leqslant_{\mathsf{P}} \mu\,\mathtt{t}.H'}\ \textsc{SPar}\mu\textsc{R}$$

Fig. 7. Subtyping for partial session types.

if they are related by $\leqslant_{\mathsf{B}}$. SEnd relates terminated channel types. SμL and SμR are standard under coinduction [26, § 21], relating types up-to their unfolding.

Capabilities In our type system linearity is enforced via capabilities, rather than via environment splitting as in most session type systems. Each process has a capability set C associated with it, allowing it to communicate on the associated channels. The tracked type $\mathtt{tr}(\rho)$ is a singleton type associating a channel to capability ρ and to no other, which in turn maps to the channel's session type $\{\rho \mapsto S\}$. Hence two variables with the same capability ρ are aliases for the same channel. Individual capabilities are joined together using the $\otimes$ operator: $C = \{\rho_1 \mapsto S_1\} \otimes \ldots \otimes \{\rho_n \mapsto S_n\}$. The ordering is insignificant. The type system maintains the invariant that $\rho_1, \ldots, \rho_n$ are distinct.

Definition 6 (Terminated capabilities). *A capability set C is* terminated *if for every $\rho \in \mathrm{dom}\,(C)$, $C(\rho) = \mathtt{end}$.*

Definition 7 (Substitution of capabilities).

$$\{\rho \mapsto S\}[\rho'/\rho_2] = \{\rho \mapsto S\} \qquad \{\rho \mapsto S\}[\rho'/\rho] = \{\rho' \mapsto S\}$$

$$\emptyset[\rho'/\rho] = \emptyset \qquad (C_1 \otimes C_2)[\rho'/\rho] = C_1[\rho'/\rho] \otimes C_2[\rho'/\rho]$$

There are two important concepts relating the environment Γ and the capability set C: *completeness* and *consistency*, used in our type system.

Completeness means that if a channel is in Γ and its capability is in C, then Γ also contains the other endpoints of the channel and C contains the corresponding capability. In this case, there is a self-contained collection of channels that can communicate. Consistency means that the opposite endpoints of every channel have dual partial types.

Definition 8 (Completeness and consistency).
(Γ, C) is complete *iff for all* $\mathtt{s}[\mathtt{p}]:\mathtt{tr}(\rho_\mathtt{p})$ *with* $\rho_\mathtt{p}:\{\rho_\mathtt{p}\mapsto S_\mathtt{p}\} \in \Gamma$ *and* $\{\rho_\mathtt{p}\mapsto S_\mathtt{p}\} \in C$, $\mathtt{q} \in S_\mathtt{p}$ *implies* $\mathtt{s}[\mathtt{q}] : \mathtt{tr}(\rho_\mathtt{q}), \rho_\mathtt{q} : \{\rho_\mathtt{q}\mapsto S_\mathtt{q}\} \in \Gamma$ *and* $\{\rho_\mathtt{q}\mapsto S_\mathtt{q}\} \in C$.

(Γ, C) is consistent *iff for all* $\mathtt{s}[\mathtt{p}] : \mathtt{tr}(\rho_\mathtt{p}), \mathtt{s}[\mathtt{q}] : \mathtt{tr}(\rho_\mathtt{q}), \rho_\mathtt{p} : \{\rho_\mathtt{p}\mapsto S_\mathtt{p}\}, \rho_\mathtt{q} : \{\rho_\mathtt{q}\mapsto S_\mathtt{q}\} \in \Gamma$ *we have* $\overline{S_\mathtt{p} \upharpoonright \mathtt{q}} \leqslant_\mathsf{P} S_\mathtt{q} \upharpoonright \mathtt{p}$.

Definition 9. *Typing judgements are inductively defined by the rules in Fig. 8, and have the form:* $\Gamma \vdash v : T; C$ *for values, or* $\Delta; \Gamma \vdash P; C$ *for processes (with* (Γ, C) *consistent, and* $\forall(\mathtt{c} : \mathtt{tr}(\rho) \in \Gamma; \{\rho\mapsto S\} \in C), S \upharpoonright \mathtt{p}$ *is defined* $\forall \mathtt{p} \in S$*).*

Γ is an environment of typed variables and channels together with their capability typing. Δ, defined in Fig. 5 is an environment of typed process names, used in rules TDef and TCall for recursive process definitions and calls. If a channel $\mathtt{s}[\mathtt{p}]$ is in Γ, with type $\mathtt{tr}(\rho)$, then Γ also contains $\rho : \{\rho\mapsto S\}$ for some session type S. The capability ρ might, or might not, be in C, to show whether or not the channel can be used. If ρ is in C, then it occurs with the same session type: $\{\rho\mapsto S\}$.

Rule TCap takes the type for a capability ρ from the capability set. TVar and TVal are standard. TInact has a standard condition that all session types have reached $\mathtt{end}$, expressed as the capability set being terminated. TPar combines the capability sets in a parallel composition. TSub is a standard subsumption rule using $\leqslant_\mathsf{S}$ (Definition 5), the difference being the type in the capability set. TSel (resp. TBr) states that the selection (resp. branching) on channel $\mathtt{c}[\mathtt{p}]$ is well typed if the capability associated with it is of compatible selection (resp. branching) type and the continuations $P_i, \forall i \in I$ are well-typed with the continuation session types. TSelP is similar to TSel, with the notable difference that an existential package is created for the channel being sent, containing the channel and its abstracted capability. Note that the actual capability to use the endpoint remains with process P. TBrP is similar to TBr, with the difference that it unpackages the channel received and binds its capability type in the continuation session type (used to identify the correct capability when received later). TRes requires the restricted environment Γ' and the associated capability set C' to be *complete* (Definition 8). TDef takes account of capability sets as well as parameters, and TCall similarly requires capability sets. The parameters of a defined process include any necessary capabilities, which then also appear in the corresponding C_i, because not all capabilities associated with the channel parameters need to be present when the call is made.

$$\textsc{TCap}\ \frac{}{\Gamma \vdash \rho : \{\rho \mapsto S\}; \{\rho \mapsto S\}} \qquad \textsc{TVar}\ \frac{\mathsf{c} : \mathtt{tr}(\rho), \rho : \{\rho \mapsto S\} \in \Gamma}{\Gamma \vdash \mathsf{c} : \mathtt{tr}(\rho); \emptyset} \qquad \textsc{TVal}\ \frac{v \in B}{\Gamma \vdash v : B; \emptyset} \qquad \textsc{TInact}\ \frac{C \text{ terminated}}{\Delta; \Gamma \vdash \mathbf{0}; C}$$

$$\textsc{TPar}\ \frac{\Delta; \Gamma \vdash P; C_1 \quad \Delta; \Gamma \vdash Q; C_2}{\Delta; \Gamma \vdash P \mid Q; C_1 \otimes C_2} \qquad \textsc{TSub}\ \frac{\Delta; \Gamma \vdash P; C \otimes \{\rho \mapsto U\} \qquad U' \leqslant_{\mathsf{S}} U}{\Delta; \Gamma \vdash P; C \otimes \{\rho \mapsto U'\}}$$

$$\textsc{TSel}\ \frac{\Gamma \vdash v : U; C \quad \Delta; \Gamma \vdash P; C' \otimes \{\rho \mapsto S_j\} \quad \mathsf{c} : \mathtt{tr}(\rho), \rho : \{\rho \mapsto S_j\} \in \Gamma \quad j \in I}{\Delta; \Gamma \vdash \mathsf{c}[\mathsf{p}] \oplus \langle l_j(v) \rangle . P; C \otimes C' \otimes \{\rho \mapsto \mathsf{p} \oplus_{i \in I} ! l_i(U_i).S_i\}}$$

$$\textsc{TBr}\ \frac{\Delta; \Gamma, x_i : U_i \vdash P_i; C \otimes C_i \otimes \{\rho \mapsto S_i\} \quad \mathsf{c} : \mathtt{tr}(\rho), \rho : \{\rho \mapsto S_i\} \in \Gamma \quad \forall i \in I}{\Delta; \Gamma \vdash \mathsf{c}[\mathsf{p}] \&_{i \in I} \{l_i(x_i).\, P_i\}; C \otimes \{\rho \mapsto \mathsf{p} \&_{i \in I} ? l_i\,(U_i)\,.S_i\}}$$

$$\textsc{TSelP}\ \frac{\Gamma \vdash v : \mathtt{tr}(\rho'); \emptyset \quad \Delta; \Gamma \vdash P; C \otimes \{\rho \mapsto S_j, \rho' \mapsto U\} \quad \mathsf{c} : \mathtt{tr}(\rho), \rho : \{\rho \mapsto S_j\} \in \Gamma \quad j \in I}{\Delta; \Gamma \vdash \mathsf{c}[\mathsf{p}] \oplus \langle l_j(\mathsf{pack}(\rho', v)) \rangle . P; C \otimes \{\rho \mapsto \mathsf{p} \oplus_{i \in I} ! l_i(\exists[\rho' | \{\rho' \mapsto U\}].\mathtt{tr}(\rho')).S_i, \rho' \mapsto U\}}$$

$$\textsc{TBrP}\ \frac{\Delta; \Gamma, \mathtt{v_i} : \mathtt{tr}(\rho_i), \rho_i : \{\rho_i \mapsto U_i\} \vdash P_i; C \otimes \{\rho \mapsto S_i\} \quad \forall i \in I \quad \mathsf{c} : \mathtt{tr}(\rho), \rho : \{\rho \mapsto S_i\} \in \Gamma}{\Delta; \Gamma \vdash \mathsf{c}[\mathsf{p}] \&_{i \in I} \{l_i(\mathsf{pack}(\rho_i, \mathtt{v_i})).\, P_i\}; C \otimes \{\rho \mapsto \mathsf{p} \&_{i \in I} ? l_i(\exists[\rho_i | \{\rho_i \mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i\}}$$

$$\textsc{TRes}\ \frac{\begin{array}{c}\Delta; \Gamma, \Gamma' \vdash P; C \otimes C' \\ (\Gamma' = \{\mathsf{s}[\mathsf{p}] : \mathtt{tr}(\rho_{\mathsf{p}}), \rho_{\mathsf{p}} : \{\rho_{\mathsf{p}} \mapsto S_{\mathsf{p}}\}\}_{\mathsf{p} \in I}, C' = \otimes_{\mathsf{p} \in I} \{\rho_{\mathsf{p}} \mapsto S_{\mathsf{p}}\}) \text{ complete}\end{array}}{\Delta; \Gamma \vdash (\nu\, s : \Gamma') P; C}$$

$$\textsc{TDef}\ \frac{\begin{array}{c}\Delta, X : \tilde{U}; \tilde{x} : \tilde{U} \vdash P; \tilde{C} \\ \Delta, X : \tilde{U}; \Gamma \vdash Q; C\end{array}}{\Delta; \Gamma \vdash \mathsf{def}\, X \langle \tilde{x} : \tilde{U} \rangle = P; \tilde{C}\ \mathsf{in}\ Q; C} \qquad \textsc{TCall}\ \frac{\forall i \in \{1..n\} \quad \Gamma \vdash v_i : U_i; C_i}{\Delta, X : U_1, \ldots, U_n; \Gamma \vdash X \langle v_1, \ldots, v_n \rangle; C_1 \otimes \ldots \otimes C_n}$$

Fig. 8. Typing rules

4 Case study: Producer-Consumer

We now expand on the producer-consumer scenario from § 1 by discussing the process definitions and showing part of the typing derivation. To lighten the notation, we present a set of mutually recursive definitions, instead of using the formal syntax of def ... in.

Recall that the example consists of three processes: the producer, the consumer, and a one-place buffer (Fig. 1). The producer and the consumer communicate with the buffer on a single shared channel. Each of the two must wait to receive the *capability* to communicate on the channel before doing so.

The buffer B is parameterised by channel x and by the capability for it, $\rho_{\mathtt{x}}$, and alternately responds to add and request messages. At the end of the definition,

$\{\rho_x \mapsto S_b\}$ shows the held capability and its session type.

$$\mathsf{B}\langle \mathtt{x}{:}\mathtt{tr}(\rho_x), \rho_x{:}\{\rho_x \mapsto S_b\}\rangle = \mathtt{x}[\mathtt{q}]\&\mathsf{add}(\mathsf{i}).\mathtt{x}[\mathtt{q}]\&\mathsf{request}(\mathtt{r}).\mathtt{x}[\mathtt{p}]\oplus\mathsf{send}(\mathsf{i}).\mathsf{B}\langle \mathtt{x}, \rho_x\rangle; \{\rho_x \mapsto S_b\}$$

The producer is represented by two process definitions: Produce and P. Produce is a recursive process with several parameters. Channels $\mathtt{x}$ and $\mathtt{y}$ are used to communicate with the consumer and the buffer, respectively. Their capabilities are ρ_x and ρ_y. Finally, i is the value to be sent to the buffer. The process sends a value to the buffer (add(i)), transfers the capability for the shared channel $\mathtt{y}$ (turn(ρ_y)) and receives it back from the consumer. Process P is the entry point for the producer. It has the same parameters as Produce, except for i. The only action of P is to send the consumer a shared reference to the channel used for communication with the buffer —$\mathtt{x}[\mathtt{c}]\oplus\mathsf{buffer}(\mathsf{pack}(\rho_y, \mathtt{y}[\mathtt{b}]))$.

$$\begin{aligned}
&\mathsf{Produce}\langle \mathtt{x} : \mathtt{tr}(\rho_x), \mathtt{y} : \mathtt{tr}(\rho_y), \mathsf{i} : \mathtt{Int}, \rho_x : \{\rho_x \mapsto S'_p\}, \rho_y : \{\rho_y \mapsto S_q\}\rangle = \mathtt{y}[\mathtt{b}]\oplus\mathsf{add}(\mathsf{i}).\\
&\quad \mathtt{x}[\mathtt{c}]\oplus\mathsf{turn}(\rho_y).\ \mathtt{x}[\mathtt{c}]\&\mathsf{turn}(\rho_y).\ \mathsf{Produce}\langle \mathtt{x}, \mathtt{y}, \mathsf{i}{+}1, \rho_x, \rho_y\rangle; \{\rho_x \mapsto S'_p\} \otimes \{\rho_y \mapsto S_q\}\\
&\mathsf{P}\langle \mathtt{x} : \mathtt{tr}(\rho_x), \mathtt{y} : \mathtt{tr}(\rho_y), \rho_x : \{\rho_x \mapsto S_p\}, \rho_y : \{\rho_y \mapsto S_q\}\rangle =\\
&\quad \mathtt{x}[\mathtt{c}]\oplus\mathsf{buffer}(\mathsf{pack}(\rho_y, \mathtt{y}[\mathtt{b}])).\ \mathsf{Produce}\langle \mathtt{x}, \mathtt{y}, 0, \rho_x, \rho_y\rangle; \{\rho_x \mapsto S_p\} \otimes \{\rho_y \mapsto S_q\}
\end{aligned}$$

In a similar way, the consumer is represented by Consume and C. The parameters, however, are different. C has $\mathtt{x}$ and its capability ρ_x, for communication with the producer, but it does not have $\mathtt{y}$ or ρ_y for communication with the buffer. It receives $\mathtt{y}$ from the producer, as part of $\mathsf{pack}(\rho_y, \mathtt{y}[\mathtt{b}])$, and $\mathtt{y}$ is passed as a parameter to Consume. The capability ρ_y is not a parameter of Consume, but it is received in a turn message from the producer.

$$\begin{aligned}
&\mathsf{Consume}\langle \mathtt{x} : \mathtt{tr}(\rho_x), \mathtt{y} : \mathtt{tr}(\rho_y), \rho_x : \{\rho_x \mapsto S'_c\}\rangle = \mathtt{x}[\mathtt{p}]\&\mathsf{turn}(\rho_y).\ \mathtt{y}[\mathtt{b}]\oplus\mathsf{request}(r).\\
&\qquad\qquad \mathtt{y}[\mathtt{b}]\&\mathsf{send}(\mathsf{i}).\ \mathtt{x}[\mathtt{p}]\oplus\mathsf{turn}(\rho_y).\ \mathsf{Consume}\langle \mathtt{x}, \mathtt{y}, \rho_x\rangle; \{\rho_x \mapsto S'_c\}\\
&\mathsf{C}\langle \mathtt{x} : \mathtt{tr}(\rho_x), \rho_x : \{\rho_x \mapsto S_c\}\rangle =\\
&\qquad\qquad \mathtt{x}[\mathtt{p}]\&\mathsf{buffer}(\mathsf{pack}(\rho_y, \mathtt{y}[\mathtt{b}])).\ \mathsf{Consume}\langle \mathtt{x}, \mathtt{y}, \rho_x\rangle; \{\rho_x \mapsto S_c\}
\end{aligned}$$

The complete system consists of the producer, the consumer and the buffer in parallel, with sessions s_1 (roles $\mathtt{p}$ and $\mathtt{c}$) and s_2 (roles $\mathtt{q}$ and $\mathtt{b}$) scoped to construct a closed process.

$$(\nu\, s_1)((\nu\, s_2)(\mathsf{P}\langle \mathtt{s}_1[\mathtt{p}], \mathtt{s}_2[\mathtt{q}], \rho_p, \rho_q\rangle \mid \mathsf{B}\langle \mathtt{s}_2[\mathtt{b}], \rho_b\rangle) \mid \mathsf{C}\langle \mathtt{s}_1[\mathtt{c}], \rho_c\rangle$$

The session types involved in these processes are projections of the global type G (§3). They specify how each role is expected to use its channel endpoint. The roles are $\mathtt{b}$ for the buffer, $\mathtt{q}$ for the combined role of the producer and the consumer as they interact with the buffer, $\mathtt{p}$ for the producer, and $\mathtt{c}$ for the consumer.

$$\begin{aligned}
S_b &= G \upharpoonright \mathtt{b} = \mu\, \mathtt{t}.\mathtt{q}\&?\mathsf{add}(\mathtt{Int}).\mathtt{q}\&?\mathsf{request}(\mathtt{Str}).\mathtt{q}\oplus!\mathsf{send}(\mathtt{Int}).t\\
S_q &= G \upharpoonright \mathtt{q} = \mu\, \mathtt{t}.\mathtt{b}\oplus!\mathsf{add}(\mathtt{Int}).\mathtt{b}\oplus!\mathsf{request}(\mathtt{Str}).\mathtt{b}\&?\mathsf{send}(\mathtt{Int}).t\\
S_p &= G \upharpoonright \mathtt{p} = \mathtt{c}\oplus!\mathsf{buffer}(\exists[\rho_q|\{\rho_q \mapsto S'_q\}].\mathtt{tr}(\rho_q)).\mu\, \mathtt{t}.\\
&\qquad \mathtt{c}\oplus!\mathsf{turn}(\{\rho_q \mapsto S'_q\}).\mathtt{c}\&?\mathsf{turn}(\{\rho_q \mapsto S_q\}).t\\
S_c &= G \upharpoonright \mathtt{c} = \mathtt{p}\&?\mathsf{buffer}(\exists[\rho_q|\{\rho_q \mapsto S'_q\}].\mathtt{tr}(\rho_q)).\mu\, \mathtt{t}.\\
&\qquad \mathtt{p}\&?\mathsf{turn}(\{\rho_q \mapsto S'_q\}).\mathtt{p}\oplus!\mathsf{turn}(\{\rho_q \mapsto S_q\}).t
\end{aligned}$$

$$\dfrac{\Gamma \vdash \mathtt{x}:\mathtt{tr}(\rho_\mathtt{x}), \mathtt{y}:\mathtt{tr}(\rho_\mathtt{y}), \mathsf{i}:\mathtt{Int}, \rho_\mathtt{x}:\{\rho_\mathtt{x} \mapsto S'_\mathtt{p}\}, \rho_\mathtt{y}:\{\rho_\mathtt{y} \mapsto S_\mathtt{q}\}; \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}}{\Delta; \Gamma \vdash \mathsf{Produce}\langle x, y, \mathsf{i}{+}1, \rho_\mathtt{x}, \rho_\mathtt{y}\rangle; \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}} \text{ TCALL}$$

$$\dfrac{\vdots}{\Delta; \Gamma \vdash \mathtt{x}[\mathtt{c}]\&\{\mathsf{turn}(\rho_\mathtt{y}).\mathsf{Produce}\langle x, y, \mathsf{i}{+}1, \rho_\mathtt{x}, \rho_\mathtt{y}\rangle\}; \{\rho_\mathtt{x} \mapsto \mathtt{c}\&?\mathsf{turn}(\{\rho_\mathtt{q} \mapsto S_\mathtt{q}\}).S'_\mathtt{p}\}} \text{ TBR}$$

$$\dfrac{\vdots \quad \dfrac{}{\Gamma \vdash \rho_\mathtt{q}:\{\rho_\mathtt{q} \mapsto S'_\mathtt{q}\}; \{\rho_\mathtt{q} \mapsto S'_\mathtt{q}\}}\text{ TCAP} \quad \mathtt{x}:\mathtt{tr}(\rho_\mathtt{x}), \rho_\mathtt{x}:\{\rho_\mathtt{x} \mapsto \mathtt{c}\&?\mathsf{turn}(\{\rho_\mathtt{q} \mapsto S_\mathtt{q}\}).S'_\mathtt{p}\} \in \Gamma}{\Delta; \Gamma \vdash \mathtt{x}[\mathtt{c}]\oplus\langle \mathsf{turn}(\rho_\mathtt{q})\rangle.\mathtt{x}[\mathtt{c}]\&\{\mathsf{turn}(\rho_\mathtt{y}).\mathsf{Produce}\langle x, y, \mathsf{i}{+}1, \rho_\mathtt{x}, \rho_\mathtt{y}\rangle\}; \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S'_\mathtt{q}\}} \text{ TSEL}$$

$$\dfrac{\vdots \quad \dfrac{\mathsf{i} \in \mathtt{Int}}{\Gamma \vdash \mathsf{i}:\mathtt{Int}; \emptyset}\text{ TVAL} \quad \mathtt{y}:\mathtt{tr}(\rho_\mathtt{y}), \rho_\mathtt{y}:\{\rho_\mathtt{y} \mapsto S'_\mathtt{q}\} \in \Gamma}{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{y}[\mathtt{b}]\oplus\langle \mathsf{add}(\mathsf{i})\rangle.\mathtt{x}[\mathtt{c}]\oplus\langle \mathsf{turn}(\rho_\mathtt{q})\rangle.\mathtt{x}[\mathtt{c}]\&\{\mathsf{turn}(\rho_\mathtt{q}).\mathsf{Produce}\langle \mathtt{x}, \mathtt{y}, \mathsf{i}{+}1, \rho_\mathtt{x}, \rho_\mathtt{y}\rangle\};\\ \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}\end{array}} \text{ TSEL}$$

Fig. 9. Typing derivation for Produce.

$$\dfrac{\Gamma \vdash \mathtt{x}:\mathtt{tr}(\rho_\mathtt{x}), \mathtt{y}:\mathtt{tr}(\rho_\mathtt{y}), \rho_\mathtt{x}:\{\rho_\mathtt{x} \mapsto S'_\mathtt{p}\}, \rho_\mathtt{y}:\{\rho_\mathtt{y} \mapsto S_\mathtt{q}\}; \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}}{\Delta; \Gamma \vdash \mathsf{Produce}\langle x, y, \mathsf{i}, \rho_\mathtt{p}, \rho_\mathtt{q}\rangle; \{\rho_\mathtt{x} \mapsto S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}} \text{ TCALL}$$

$$\dfrac{\vdots \quad \dfrac{\mathtt{y}:\mathtt{tr}(\rho_\mathtt{y}), \rho_\mathtt{y}:\{\rho_\mathtt{y} \mapsto S_\mathtt{q}\}}{\Gamma \vdash \mathtt{y}:\mathtt{tr}(\rho_\mathtt{y}); \emptyset}\text{ TVAR} \quad \mathtt{x}, \rho_\mathtt{x}:\{\rho_\mathtt{x} \mapsto S_\mathtt{p}\} \in \Gamma}{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{x}[\mathtt{c}]\oplus\langle \mathsf{buffer}(\mathsf{pack}(\rho_\mathtt{q}, \mathtt{y}[\mathtt{b}]))\rangle.\mathsf{Produce}\langle \mathtt{x}, \mathtt{y}, \mathsf{i}, \rho_\mathtt{x}, \rho_\mathtt{y}\rangle;\\ \{\rho_\mathtt{y} \mapsto \mathtt{p}\oplus!l(\exists[\rho_\mathtt{y}|\{\rho_\mathtt{y} \mapsto S_\mathtt{q}\}].\mathtt{tr}(\rho_\mathtt{y})).S'_\mathtt{p}, \rho_\mathtt{y} \mapsto S_\mathtt{q}\}\end{array}} \text{ TSELP}$$

Fig. 10. Typing derivation for P.

These types occur in the capabilities associated with each process. For example process $\mathsf{P}\langle \mathtt{s}_1[\mathtt{p}], \mathtt{s}_2[\mathtt{q}], \rho_\mathtt{p}, \rho_\mathtt{q}\rangle$ has $\{\rho_\mathtt{q} \mapsto S_\mathtt{q}\} \otimes \{\rho_\mathtt{p} \mapsto S_\mathtt{p}\}$, process $\mathsf{B}\langle \mathtt{s}_2[\mathtt{b}], \rho_\mathtt{b}\rangle$ has $\{\rho_\mathtt{b} \mapsto S_\mathtt{b}\}$, and process $\mathsf{C}\langle \mathtt{s}_1[\mathtt{c}], \rho_\mathtt{c}\rangle$ has $\{\rho_\mathtt{c} \mapsto S_\mathtt{c}\}$.

To illustrate the typing rules, we show the typing derivation for the producer, i.e. processes Produce (Fig. 9) and P (Fig. 10). Full derivations for all of the processes are in the technical report. The derivations use the following definitions.

$$\begin{aligned}
S'_\mathtt{p} &= \mu\,\mathtt{t}.\mathtt{c}\oplus!\mathsf{turn}(\{\rho_\mathtt{q} \mapsto S'_\mathtt{q}\}).\mathtt{c}\&?\mathsf{turn}(\{\rho_\mathtt{q} \mapsto S_\mathtt{q}\}).t\\
S'_\mathtt{q} &= \mathtt{b}\oplus!\mathsf{request}(\mathtt{Str}).\mathtt{b}\&?\mathsf{send}(\mathtt{Int}).S_\mathtt{q}\\
\Delta &= \mathsf{Produce} : (\mathtt{tr}(\rho_\mathtt{p}), \mathtt{tr}(\rho_\mathtt{q}), \mathtt{Int}, \{\rho_\mathtt{p} \mapsto S_\mathtt{p}\}, \{\rho_\mathtt{q} \mapsto S_\mathtt{q}\})\\
\Gamma &= \mathtt{x}:\mathtt{tr}(\rho_\mathtt{p}), \mathtt{y}:\mathtt{tr}(\rho_\mathtt{q}), \mathsf{i}:\mathtt{Int}, \rho_\mathtt{p}:\{\rho_\mathtt{p} \mapsto S_\mathtt{p}\}, \rho_\mathtt{q}:\{\rho_\mathtt{q} \mapsto S_\mathtt{q}\}
\end{aligned}$$

Scenarios with multiple producers/consumers can be represented in a similar way, the capabilities acting as a form of lock for the resource being shared. The full typing derivation for producer consumer case study can be found in the extended version of this paper [33].

5 Technical Results

Following standard practice in the MPST literature, we show type safety and hence communication safety by proving a subject reduction theorem (Theorem 1).

In the usual way, session types evolve during reduction—in our system, this is seen in both the Γ environment and the capability set C.

Definition 10 (Typing context reduction). *The* reduction $(\Gamma; C) \longrightarrow (\Gamma'; C')$ *is:*

$$(\mathtt{s}[\mathtt{p}]:\mathtt{tr}(\rho_\mathtt{p}),\ \mathtt{s}[\mathtt{q}]:\mathtt{tr}(\rho_\mathtt{q}), \rho_\mathtt{p}:\{\rho_\mathtt{p}\mapsto S_\mathtt{p}\}, \rho_\mathtt{q}:\{\rho_\mathtt{q}\mapsto S_\mathtt{q}\}; \{\rho_\mathtt{p}\mapsto S_\mathtt{p}, \rho_\mathtt{q}\mapsto S_\mathtt{q}\}) \longrightarrow$$
$$(\mathtt{s}[\mathtt{p}]:\mathtt{tr}(\rho_\mathtt{p}),\ \mathtt{s}[\mathtt{q}]:\mathtt{tr}(\rho_\mathtt{q}), \rho_\mathtt{p}:\{\rho_\mathtt{p}\mapsto S_k\}, \rho_\mathtt{q}:\{\rho_\mathtt{q}\mapsto S'_k\}; \{\rho_\mathtt{p}\mapsto S_k, \rho_\mathtt{q}\mapsto S'_k\})$$
$$\textit{if}\begin{cases}\mathtt{unf}(S_\mathtt{p}) = \mathtt{q}\oplus_{i\in I}!l_i(U_i).S_i & k \in I\\ \mathtt{unf}(S_\mathtt{q}) = \mathtt{p}\ \&_{i\in I\cup J}\ ?l_i(U'_i).S'_i & U_k \leqslant_\mathsf{S} U'_k\end{cases}$$
$$\textit{or if}\ \begin{cases}\mathtt{unf}(S_\mathtt{p}) = \mathtt{q}\oplus_{i\in I}!l_i(\exists[\rho_i|\{\rho_i\mapsto U_i\}].\mathtt{tr}(\rho_i)).S_i & k \in I\\ \mathtt{unf}(S_\mathtt{q}) = \mathtt{p}\ \&_{i\in I\cup J}\ ?l_i(\exists[\rho_i|\{\rho_i\mapsto U'_i\}].\mathtt{tr}(\rho_i)).S'_i & U_k \leqslant_\mathsf{S} U'_k\end{cases}$$

$$(\Gamma,\mathtt{c}:\mathtt{tr}(\rho), \rho:\{\rho\mapsto U\}; C\otimes\{\rho\mapsto U\}) \longrightarrow (\Gamma',\mathtt{c}:\mathtt{tr}(\rho), \rho:\{\rho\mapsto U'\}; C'\otimes\{\rho\mapsto U'\})$$
$$\textit{if}\ \ (\Gamma; C) \longrightarrow (\Gamma'; C')\ \ \textit{and}\ \ U\leqslant_\mathsf{S} U'$$

Following [28] our Definition 10 also accommodates subtyping ($\leqslant_\mathsf{S}$) and our iso-recursive type equivalence (hence, unfolds types explicitly).

Theorem 1 (Subject reduction). *If* $\Delta; \Gamma \vdash P; C$ *and* $P \longrightarrow P'$*, then there exist* Γ' *and* C' *such that* $\Delta; \Gamma' \vdash P'; C'$ *and* $(\Gamma; C) \longrightarrow^* (\Gamma'; C')$*.*

The proof is by induction on the derivation of $P \longrightarrow P'$, with an analysis of the derivation of $\Delta; \Gamma \vdash P; C$. A key case is RRes, which requires preservation of the condition in TRes that (Γ, C) is consistent. This is because a communication reduction consumes matching prefixes from a pair of dual partial session types, which therefore remain dual. The full proof is in the extended version of this paper [33].

6 Related Work, Conclusion and Future Work

From the beginning of session types, channel endpoints were treated as linear resources so that each role in a protocol could be implemented by a unique agent. This approach is reinforced by several connections between session types and other linear type theories: the encodings of binary session types and multiparty session types into linear π-calculus types [12,28]; the Curry-Howard correspondence between binary session types and linear logic [6,34]; the connection between multiparty session types and linear logic [7,8].

Some session type systems generalise linearity. Vasconcelos [32] allows a session type to become non-linear, and sharable, when it reaches a state that is invariant with every subsequent message. Mostrous and Vasconcelos [24] define *affine* session types, in which each endpoint must be used at most once and can be discarded with an explicit operator. In Fowler *et al.*'s [15] implementation of session types for the Links web programming language, affine typing allows sessions to be cancelled when exceptions (including dropped connections) occur. Caires and Pérez [5] use monadic types to describe cancellation (i.e. affine sessions)

and non-determinism. Pruiksma and Pfenning [27] use adjoint logic to describe session cancellation and other behaviours including multicast and replication.

Usually linearity spreads, because a data structure containing linear values must also be linear. In the standard π-calculus, exceptions to this nature of linearity have been studied by Kobayashi in his work on deadlock-freedom Padovani [25] extends the linear π-calculus with composite regular types in such a way that data containing linear values can be shared among several processes. However, this sharing can occur only if there is no overlapping access to such values, which differs from our work where we have full sharing of values. On the other hand, we work directly with (multiparty) session types, whereas Padovani works with linear π-calculus and obtains his results via the encoding of session types into linear π-types [12].

Session types are related to the concept of *typestate* [29], especially in the work of Kouzapas *et al.* [21,22] which defines a typestate system for Java based on multiparty session types. Typestate systems require linear typing or some other form of alias control, to avoid conflicting state changes via multiple references. Approaches include the permission-based systems used in the Plural and Plaid languages [4,30] and the fine-grained approach of Militão *et al.* [23]. Crafa and Padovani [10] develop a "chemical" approach to concurrent typestate oriented programming, allowing objects to be accessed and modified concurrently by several processes, each potentially changing only part of their state. Our approach is partly inspired by Fähndrich and DeLine's "adoption and focus" system [14], in which a shared stateful resource (in our case, a session channel) is separated from the linear key (capability, in our system) that enables it to be used. In this way the state changes of channels follow the standard session operations, channels can be shared (for example, stored in shared data structures), and access can be controlled by passing the capability around the system.

Balzer *et al.* [2,3] support sharing of binary session channels by allowing locks to be acquired and released at points that are explicitly specified in the session type. Our approach with multiparty sessions is not based on locks, so it doesn't require runtime mechanisms for managing blocked processes and notifying them when locks are released.

We have presented a new system of multiparty session types with capabilities, which allows sharing of resources in a way that generalises the strictly linear or affine access control typical of session type systems. The key technical idea is to separate a channel from the capability of using the channel. This allows channels to be shared, while capabilities are linearly controlled. We use a form of existential typing to maintain the link between a channel and its capability, while both are transmitted in messages. We have proved communication safety, formulated as a subject reduction theorem (Theorem 1). An area of future work is to prove progress and deadlock-freedom properties along the lines of, for example, Coppo *et al.* [9]. Another possibility is to apply our techniques to functional languages with session types [17].

References

1. Ancona, D., et al.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2–3), 95–230 (2016). https://doi.org/10.1561/2500000031
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. PACMPL **1**(ICFP), 37:1–37:29 (2017). https://doi.org/10.1145/3110281
3. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: ESOP. LNCS, vol. 11423, pp. 611–639. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_22
4. Bierhoff, K., Aldrich, J.: PLURAL: checking protocol compliance under aliasing. In: ICSE Companion. pp. 971–972. ACM Press (2008). https://doi.org/http://doi.acm.org/10.1145/1370175.1370213
5. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: ESOP. LNCS, vol. 10201, pp. 229–259. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_9
6. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. LNCS, vol. 6269, pp. 222–236. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_16
7. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: CONCUR. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl — Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.33
8. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. In: CONCUR. LIPIcs, vol. 42. Schloss Dagstuhl — Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.CONCUR.2015.412
9. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science **26**(2), 238–302 (2016). https://doi.org/10.1017/S0960129514000188
10. Crafa, S., Padovani, L.: The chemical approach to typestate-oriented programming. ACM Transactions on Programming Languages and Systems **39**(3), 13:1–13:45 (2017). https://doi.org/10.1145/3064849
11. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: POPL. pp. 262–275. ACM (1999). https://doi.org/10.1145/292540.292564
12. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: PPDP. ACM (2012). https://doi.org/10.1145/2370776.2370794
13. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Logical Methods in Computer Science **8**(4) (2012). https://doi.org/10.2168/LMCS-8(4:6)2012
14. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: PLDI. pp. 13–24. ACM (2002). https://doi.org/10.1145/512529.512532
15. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. PACMPL **3**(POPL), 28:1–28:29 (2019). https://doi.org/10.1145/3290341
16. Gay, S.J., Ravara, A. (eds.): Behavioural Types: From Theory to Tools. River Publishers (2017). https://doi.org/10.13052/rp-9788793519817
17. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. Journal of Functional Programming **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268

18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM Press (2008). https://doi.org/10.1145/1328438.1328472
19. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
20. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Computing Surveys **49**(1) (2016). https://doi.org/10.1145/2873052
21. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: PPDP. pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
22. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. Science of Computer Programming **155**, 52–75 (2018). https://doi.org/10.1016/j.scico.2017.10.006
23. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: FTFJP. pp. 7:1–7:7. ACM (2010). https://doi.org/10.1145/1924520.1924527
24. Mostrous, D., Vasconcelos, V.T.: Affine sessions. Logical Methods in Computer Science **14**(4) (2018). https://doi.org/10.23638/LMCS-14(4:14)2018
25. Padovani, L.: Type reconstruction for the linear π-calculus with composite regular types. Logical Methods in Computer Science **11**(4) (2015). https://doi.org/10.2168/LMCS-11(4:13)2015
26. Pierce, B.C.: Types and programming languages. MIT Press (2002)
27. Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. In: PLACES. Electronic Proceedings in Theoretical Computer Science, vol. 291, pp. 60–79. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.291.6
28. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl — Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
29. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
30. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, É.: First-class state change in Plaid. In: OOPSLA. pp. 713–732. ACM (2011). https://doi.org/10.1145/2048066.2048122
31. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. Springer LNCS, vol. 817, pp. 398–413 (1994)
32. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
33. Voinea, A.L., Dardha, O., Gay, S.J.: Resource sharing via capability-based multiparty session types. Tech. rep., School of Computing Science, University of Glasgow (2019), http://www.dcs.gla.ac.uk/~ornela/publications/VDG19-Extended.pdf
34. Wadler, P.: Propositions as sessions. In: ICFP. pp. 273–286. ACM (2012). https://doi.org/10.1145/2364527.2364568
35. Walker, D., Morrisett, J.G.: Alias types for recursive data structures. In: TIC. pp. 177–206. LNCS, Springer (2000)
36. Yoshida, N., Deniélou, P., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FOSSACS (2010). https://doi.org/doi:10.1007/978-3-642-12032-9_10