Partial and Complete Processes in Multiparty Sessions [1]

Mario Coppo Mariangiola Dezani-Ciancaglini
Ines Margaria Maddalena Zacchi [2]

Dipartimento di Informatica Università di Torino, corso Svizzera 185, 10149 Torino, Italy

Abstract

Multiparty sessions describe the interactions among multiple agents in a distributed environment and require essentially two steps: the specification of the communication protocols and the implementation of such protocols as processes. Multiparty session types address this methodology: global and session types provide the communication protocols, whereas the processes describe the behaviour of the peers involved in the sessions. Because of the close relationships between types and processes, some information, such as the names of senders and receivers, are replicated both in types and in processes.
In multiparty conversations it is quite natural that participants with essentially the same role are implemented by processes that follow the same pattern, differing only in the senders and receivers of communication actions. In order to allow for a lighter and less rigid development of processes, we propose a translation tool which allows one to write processes in a simplified syntax, called partial syntax, where the names of senders/receivers for input/output operations are omitted. By adding the missing information, *partial processes* can be automatically translated in *complete processes*, for which an operational semantics is defined. The partial syntax, in particular, allows one to use the same process template to implement similar participants.
In this paper we present a translation and type checking algorithm from partial to complete processes, which, if successful, also assures that the target process is well typed. The algorithm is synthesised from a rule-based description of the translation in natural semantics and it is proved sound and complete with respect to the translation rules.

1 Introduction

Session types are one of the most successful formalisms introduced to describe communicating processes and to study their behaviour. The basic idea, appeared first in [11] and [6], is to introduce a new form of polymorphism which permits the typing of channel names by structured sequences of types, abstractly representing the trace of the usage of the channels. In modelling distributed systems, where processes interact by means of message passing, it is appropriate to allow many interactions to occur within the scope of private channels following disciplined protocols. As usual, we call sessions such private interactions and session types the protocols describing them. In its simplest form, a session is established between two peers, such as a client connecting with a server. In general, a session may involve any (usually fixed) number of peers. In these cases, we speak of multiparty sessions

[1] This work was partially supported by ICT COST Action IC1201 BETTY, MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

[2] Email: {coppo,dezani,ines,zacchi}@di.unito.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and of multiparty session types [7] for their protocol descriptions. A multiparty session type theory consists of three parts: global types, processes, and local types, called also session types. *Global types* describe communication protocols in terms of the interactions between peers, of the order of these interactions, and of the kind of exchanged messages. The description given by a global type is neutral, independent from the peers and their viewpoints. *Processes* describe, by means of a formal language, the behaviour of the peers involved in the session. For each peer a *session type* describes the same communication protocol as the global type, but from the viewpoint of the peer. Local and global types are related by a projection operation that extracts local types from the global ones, and a type system makes sure that a process uses the communication channels it owns according to their local types. Among the more interesting features of interaction between peers, session delegation is a key operation which permits to rely on other parties for completing specific tasks transparently in a type safe manner. A typical scenario is given by the protocol of Remote Procedure Call for server/client distributed systems. In such a case the server, after receiving a request from a client, delegates remaining interactions with the client to an application process. The client and the application process are initially unknown to each other but later communicate directly, transparently to the client, through dynamic mobility of the communication channel. Such protocol is usually synchronous, because the processes involved in the communications remain tightly coupled.

Regarding the syntax of processes, the need to specify sender and receivers for each input/output operation makes cumbersome the code writing; moreover, in distributed applications often many processes perform exactly the same pattern of input/output operations, differing only for the involved participants. For example in the Remote Procedure Call protocol, the application processes, that provide the services, in many interesting cases differ only in the process names involved in the communications. In this paper we present a translation algorithm that allows one to code processes in a simplified syntax, called *partial syntax*, which does not require to specify the names of senders/receivers for input/output operations. Partial code is simpler to write and can be shared by different participants in a conversation, but it is incomplete and cannot be directly executed. The executable code of the processes, written in *complete syntax* (i.e. including the names of the processes involved in the communications) can be obtained automatically from the partial code by exploiting the information given by the global types. The translation is successful only if the target process is well typed according to standard typing rules for multiparty sessions, so the translation algorithm also includes type checking. The translation from *partial processes* to *complete processes* is formalised in natural semantics by a set of rules. These rules implicitly define a recursive translation and type checking function, whose code is given in a ML-like language. The soundness and completeness of both the natural semantics rules and the corresponding algorithm can be easily proved.

In the present paper process execution is defined by a synchronous operational semantics. However the translation does not depend on the way in which communications are performed, and could be applied as well to an asynchronous calculus like that defined in [1].

Multiparty sessions arise motivated by the applications to financial protocols in the context of the development of the language Scribble [12]. The first theoretical works on multiparty session types are [2] and [7]. The paper [2] uses a distributed calculus where each channel connects a master endpoint to one or more slave endpoints. Both processes

and types in [7] share a vector of channels and each communication uses one of these channels. The process syntaxes in the present paper are similar to those in [1]; the main difference is that communications in [1] are asynchronous. The paper [4] takes advantage of partial syntax flexibility for dealing with self-adapting protocol participants. Bisimulation of a synchronous multiparty session calculus is studied in [9], where the calculus syntax differs from our complete syntax only for the lack of multiple receivers. We refer to [5,3,8] for comprehensive overviews of related works.

Outline. Section 2 introduces global types and partial syntax by means of examples. The whole calculus and the type system are given in Sections 3 and 4, respectively. Section 5 concludes with the natural semantics rules, mapping processes written in partial syntax to processes written in complete syntax. Lastly a translation and type checking algorithm is provided.

2 Motivating Examples

We present the syntax of the calculus with the aid of examples, in order to give a basic idea of functionalities and linguistic features.

The overall scenario, involving a Manager (M), an Italian factory (IF), an American factory (AF) and two transport companies (IC) and (AC), proceeds as follows.

(i) The manager sends to the factories an identifier of items to supply.

(ii) The factories communicate the manager the numbers of items they can provide.

(iii) If the total number is satisfactory, the manager notifies the factories and the transport companies to arrange the details and waits for the expedition date from both transport companies. Otherwise the manager notifies all participants to quit the protocol.

Multiparty session programming consists of two steps: specifying the intended communication protocols via global types, and implementing these protocols as processes. The specifications of the manager-factories protocol are given by the global type G_a. The participants are actually coded by numbers: in G_a we have $\mathtt{M}=5$, $\mathtt{IF}=1$, $\mathtt{AF}=2$, $\mathtt{IC}=3$, $\mathtt{AC}=4$.

$$
\begin{array}{lll}
G_a = & & \\
\quad 1.\ 5 \longrightarrow \{1,2\}: & \langle \mathsf{string} \rangle. & \\
\quad 2.\ 1 \longrightarrow \{5\}: & \langle \mathsf{nat} \rangle. & \\
\quad 3.\ 2 \longrightarrow \{5\}: & \langle \mathsf{nat} \rangle. & \\
\quad 4.\ 5 \longrightarrow \{1,2,3,4\}: & \{\mathsf{ok}: 1 \longrightarrow \{3\} : \langle \mathsf{nat} \rangle. & \\
\quad 5. & \qquad 2 \longrightarrow \{4\} : \langle \mathsf{nat} \rangle. & \\
\quad 6. & \qquad 3 \longrightarrow \{5\} : \langle \mathsf{date} \rangle. & \\
\quad 7. & \qquad 4 \longrightarrow \{5\} : \langle \mathsf{date} \rangle.\mathsf{end} & \\
\quad 8. & \quad \mathsf{quit} : \mathsf{end}\} &
\end{array}
$$

In G_a, line 1 denotes M multicasts the same string to IF and AF. Lines 2 and 3 say both IF and AF send a natural value to M. In lines 4-8 M sends either ok or quit to all other participants. In the first case both IF and AF send to IC and AC, respectively, the number of items to deliver (lines 4-5) and the transport companies communicate delivery date to M directly (lines 6-7); in the second case there are no further communications.

Figure 1 gives the code, associated with G_a, for participants in the partial syntax, formally defined in the following section. We only need three partial processes because the

$$\begin{array}{lcl}
\mathtt{M} & = & \overline{a}[5](y).y!\langle\mathtt{"identifier"}\rangle.y?(x1).y?(x2).\ \mathbf{if}\ \mathbf{good}(x1,\,x2)\ \mathbf{then}\ \ y \oplus \mathsf{ok}.y?(d1).y?(d2).\mathbf{0} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ y \oplus \mathsf{quit}.\mathbf{0} \\
\mathtt{F}[\mathtt{j}] & = & a[\mathtt{j}](y).y?(id).y!\langle\mathtt{number}\rangle.y\&\{\mathsf{ok} : y!\langle\mathtt{number}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\} \\
\mathtt{C}[\mathtt{k}] & = & a[\mathtt{k}](y).y\&\{\mathsf{ok} : y?(n).y!\langle\mathtt{date}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}
\end{array}$$

Fig. 1. The manager-factories example.

two factories IF and AF are realised by two instances of the same partial process F[j] and similarly the two transport companies are realised by two instances of the partial process C[k].
Session name a establishes the session corresponding to G_a. Process M initiates a session involving five participants as fifth one by $\overline{a}[5](y)$; the first and second participants are obtained by instantiating j by 1 and 2, respectively, in F[j], the third and forth participants are obtained as two instances, for k = 3 and k = 4, of C[k]. Each participant uses a bound variable as a placeholder for the channel that will be used in the communications. Sender and receivers of the exchanged data are specified by the global type.

The first line of G_a is implemented by the output and input actions $y!\langle\mathtt{"identifier"}\rangle$ in the code of M and $y?(id)$ in the code of F[j]. Lines 2 and 3 are implemented by $y?(x1).y?(x2)$ in the code of M and $y!\langle\mathtt{number}\rangle$ in the code of F[j] ($\mathtt{j} = 1,2$), respectively. The next lines of G_a are implemented by the selection and branching: actions $y \oplus \mathsf{ok}.y?(d1).y?(d2).\mathbf{0}$ and $y \oplus \mathsf{quit}.\mathbf{0}$ in the code of M, $y\&\{\mathsf{ok} : y!\langle\mathtt{number}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$ in the code of F[j] and $y\&\{\mathsf{ok} : y?(n).y!\langle\mathtt{date}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$ in the code of C[k].

We now enrich this protocol introducing *delegation*. In the initial session a the participants are only the manager and the two factories, which also send the delivery date to the manager, according to the new global type G_a. Actually, after receiving ok, both factories accept a new session b offered by a unique transport company. In this session, described by the global type G_b, each factory sends to the transport company not only the number of items, but also the channel used to communicate with the manager. In this way each factory delegates the transport company to send the delivery date to the manager. This delegation is *transparent* to the manager. The new global types G_a and G_b are then:

$G_a =$
1. $3 \longrightarrow \{1,2\} : \langle\mathsf{string}\rangle.$
2. $1 \longrightarrow \{3\} : \langle\mathsf{nat}\rangle.$
3. $2 \longrightarrow \{3\} : \langle\mathsf{nat}\rangle.$
4. $3 \longrightarrow \{1,2\} : \{\mathsf{ok} : 1 \longrightarrow \{3\} : \langle\mathsf{date}\rangle.$
5. $\qquad\qquad\qquad\qquad 2 \longrightarrow \{3\} : \langle\mathsf{date}\rangle.\mathsf{end}$
6. $\qquad\qquad\quad \mathsf{quit} : \mathsf{end}\}$

$G_b =$
1. $1 \longrightarrow \{3\} : \langle\mathsf{nat}\rangle.$
2. $1 \longrightarrow \{3\} : \langle T\rangle.$
3. $2 \longrightarrow \{3\} : \langle\mathsf{nat}\rangle.$
4. $2 \longrightarrow \{3\} : \langle T\rangle.\mathsf{end}$

$T = !\langle\{3\},\mathsf{date}\rangle.\mathsf{end}$

where the session type T says that the exchanged channel must be used to send a date to participant 3 in session a, i.e. to the manager. The new code for the factories and the transport company is given in Figure 2. Notice that the code for the manager does not change, but for the replacement of $\overline{a}[5]$ with $\overline{a}[3]$. This example then shows also how the partial syntax allows one to write processes which can fit different global types. In this example, unlike the previous one, if the manager decides to quit the protocol, session b is not open, so transport company is not involved in the conversation.

In the last example we add recursive-branching behaviour to the protocol with delegation. In particular we allow M repeatedly negotiates with the factories IF and AF the number

$$\begin{array}{rcl} \mathtt{F[j]} & = & a[\mathtt{j}](y).y?(id).y!\langle \mathtt{number}\rangle.y\&\{\mathsf{ok} : b[\mathtt{j}](z).z!\langle \mathtt{number}\rangle.z!\langle\langle y\rangle\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\} \\ \mathtt{C} & = & \overline{b}[3](z).z?(n).z?((t)).t!\langle \mathtt{date}\rangle.z?(n).z?((t)).t!\langle \mathtt{date}\rangle.\mathbf{0} \end{array}$$

Fig. 2. The manager-factories example with delegation.

$$\begin{array}{rcl} \mathtt{M} & = & \overline{a}[3](y).y!\langle \mathtt{"identifier"}\rangle.\ \mathsf{def}\ X(t) = t?(x1).t?(x2).\ \mathsf{if}\ \mathbf{good}(x1,x2)\ \mathsf{then}\ t \oplus \mathsf{ok}.t?(d1).t?(d2).\mathbf{0} \\ & & \qquad\qquad \mathsf{else\ if}\ \mathbf{negotiable}(x1,x2)\ \mathsf{then}\ t \oplus \mathsf{more}.X\langle t\rangle \\ & & \qquad\qquad\qquad \mathsf{else}\ t \oplus \mathsf{quit}.\mathbf{0} \\ & & \qquad\qquad \mathsf{in}\ X\langle y\rangle \\ \mathtt{F[j]} & = & a[\mathtt{j}](y).y?(id).\mathsf{def}\ X'(x,t') = t'!\langle x\rangle.t'\&\{\mathsf{ok} : b[\mathtt{j}](z).z!\langle x\rangle.z!\langle\langle t'\rangle\rangle.\mathbf{0}, \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{more} : X'\langle \mathbf{newnumber}(x), t'\rangle, \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{quit} : \mathbf{0}\} \\ & & \qquad\qquad\qquad\qquad \mathsf{in}\ X'\langle \mathtt{number}, y\rangle \end{array}$$

Fig. 3. The manager-factories example with delegation and recursion.

of items to supply. The scenario is basically the same, the only part that changes is that, if the number of items the factories can provide is too small, the manager initiates a negotiation with the factories to increase that number. The manager starts asking the factories for a first number of items. At each step each factory answers with a new offer. The manager can accept the offer, ask for a new proposal or give up. When the manager decides to end the negotiation (accepting the last offers or giving up) he communicates the decision to factories; in the first case, as before, the transport company is contacted.

The communication protocol is described by the following global type, that differs from the one of the previous example for the recursive part, which represents the (possibly) recursive negotiation between the manager and the factories.

$G_a =$

$$\begin{array}{lll} 1. & 3 \longrightarrow \{1,2\} : \langle \mathsf{string}\rangle. & \\ 2. & \mu \mathbf{t}.\ 1 \longrightarrow \{3\} : \langle \mathsf{nat}\rangle. & \\ 3. & 2 \longrightarrow \{3\} : \langle \mathsf{nat}\rangle. & \\ 4. & 3 \longrightarrow \{1,2\} : \{\mathsf{ok} : 1 \longrightarrow \{3\} : \langle \mathsf{date}\rangle. & \\ 5. & \qquad\qquad 2 \longrightarrow \{3\} : \langle \mathsf{date}\rangle.\mathsf{end} & \\ 6. & \qquad \mathsf{more} : \mathbf{t}\} & \\ 6. & \qquad \mathsf{quit} : \mathsf{end}\} & \end{array}$$

$G_b =$

$$\begin{array}{ll} 1. & 1 \longrightarrow \{3\} : \langle \mathsf{nat}\rangle. \\ 2. & 1 \longrightarrow \{3\} : \langle T\rangle. \\ 3. & 2 \longrightarrow \{3\} : \langle \mathsf{nat}\rangle. \\ 4. & 2 \longrightarrow \{3\} : \langle T\rangle.\mathsf{end} \end{array}$$

$$T = !\langle \{3\}, \mathsf{date}\rangle.\mathsf{end}$$

The code of participants, shown in Figure 3, is similar to the previous one, but for the recursive definitions in the processes M and F[j].

3 Calculus for Multiparty Sessions

As explained in the Introduction, we consider two different syntaxes, the first one, called *partial syntax*, to write flexible code, that can be used to implement different peers involved in sessions, the other one to describe the final code. We call the latter *complete syntax* since it specifies sender and receivers for any input/output operation.

3.1 Partial syntax

The *partial processes*, ranged over by $P, Q \ldots$, and *expressions*, ranged over by $e, e', \ldots$, are given by the grammar in Table 1. To simplify the formal treatment we assume each recursively defined process having one data parameter and one channel parameter. The generalisation to multiple data and channel parameters however is straightforward.

P	::=	$\overline{u}[\mathtt{p}](y).P$	Multicast Request
	$\mid$	$u[\mathtt{p}](y).P$	Accept
	$\mid$	$y!\langle e\rangle.P$	Value sending
	$\mid$	$y?(x).P$	Value reception
	$\mid$	$y!\langle\langle z\rangle\rangle.P$	Session delegation
	$\mid$	$y?((z)).P$	Session reception
	$\mid$	$y \oplus l.P$	Selection
	$\mid$	$y\&\{l_i : P_i\}_{i\in I}$	Branching
	$\mid$	if e then P else Q	Conditional
	$\mid$	$P \mid Q$	Parallel
	$\mid$	$\mathbf{0}$	Inaction
	$\mid$	$(\nu a : G)\ P$	Hiding
	$\mid$	$X\langle e, c\rangle$	Process call
	$\mid$	def D in P	Recursion

a, b			Service name
x			Value variable
$y, z, \mathtt{t}$			Channel Variable
$\mathtt{p}, \mathtt{q}$			Participant number
$X, \mathtt{Y}$			Process variable
D	::=	$X(x,y) = P$	Declaration
l			Label
u	::=	$x \mid a$	Identifier
v	::=	$a \mid \mathsf{true}$	Value
	$\mid$	false	
e	::=	$v \mid x$	
	$\mid$	e and e'	Expression
	$\mid$	not e ...	

Table 1
Partial syntax for processes and naming conventions.

P	::=	$c!\langle\Pi, e\rangle.P$	Value sending
	$\mid$	$c?(\mathtt{p}, x).P$	Value reception
	$\mid$	$c!\langle\langle\mathtt{p}, c'\rangle\rangle.P$	Session delegation
	$\mid$	$c?((\mathtt{q}, y)).P$	Session reception
	$\mid$	$c \oplus \langle\Pi, l\rangle.P$	Selection
	$\mid$	$c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branching
	$\mid$	$(\nu s)\ P$	Hiding session
		s	Session name
		$\mathtt{c} ::= y \mid s[\mathtt{p}]$	Channel

Table 2
Complete syntax: the other syntactic forms are as in Table 1.

A partial process of the form $\overline{u}[\mathtt{p}](y).P$ initiates a new session through a service name identified by u with the other participants, each of shape $u[\mathtt{q}](y).P_{\mathtt{q}}$, where $1 \leq \mathtt{q} \leq \mathtt{p}-1$. The (bound) variable y represents the channel that will be used for the communications. We call $\mathtt{p}$, $\mathtt{q}$,... (ranging over natural numbers) the *participants* of a session. Session communications, i.e. communications that take place inside an established session, are performed using the next three pairs of primitives: the sending and receiving of a value, the session delegation and reception, where the former delegates to the latter the capability to participate in a session by passing a channel associated with the session; and the selection and branching, where the former chooses one of the branches offered by the latter. Notice that the hiding of service names specifies their global types. The rest of the syntax is standard from [6].

3.2 Complete Syntax

To define the operational semantics, partial processes must be put in an executable form (see Section 5) by adding the information about senders an receivers of messages provided by the global types. The complete syntax is defined in Table 2. It differs from the syntax of Table 1 since the input/output operations (including delegation/reception and branching/selection) specify the sender and the receivers, respectively. We use Π to range over non-empty sets of participants. Thus, $c!\langle\Pi, e\rangle$ means to send a value to all the participants in Π; accordingly, $c?(\mathtt{p}, x)$ denotes the reception of a value from participant $\mathtt{p}$. The same holds for selection/branching and delegation/reception, but in this last case the receiver is only one.

As usual, we call $s[\mathtt{p}]$ a *channel with role*: it represents the input/output port of the participant $\mathtt{p}$ in the session s.

Figure 4 gives the processes implementing the manager-factories protocol of the first

$$
\begin{array}{lcl}
\mathtt{M} & = & \overline{a}[5](y).y!\langle\{1,2\},\ \mathtt{"identifier"}\rangle.y?(1,\ x1).y?(2,\ x2). \\
 & & \qquad \textsf{if } \mathbf{good}(x1,\ x2) \textsf{ then } y \oplus \langle\{1,2,3,4\},\mathsf{ok}\rangle.y?(3,\ d1).y?(4,\ d2).\mathbf{0} \\
 & & \qquad\qquad \textsf{else } y \oplus \langle\{1,2,3,4\},\mathsf{quit}\rangle.\mathbf{0} \\
\mathtt{IF} & = & a[1](y).y?(5,id).y!\langle\{5\},\mathtt{number}\rangle.y\&(5,\{\mathsf{ok} : y!\langle\{3\},\mathtt{number}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}) \\
\mathtt{AF} & = & a[2](y).y?(5,id).y!\langle\{5\},\mathtt{number}\rangle.y\&(5,\{\mathsf{ok} : y!\langle\{4\},\mathtt{number}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}) \\
\mathtt{IC} & = & a[3](y).y\&(5,\{\mathsf{ok} : y?(1,n).y!\langle 5,\mathtt{date}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}) \\
\mathtt{AC} & = & a[4](y).y\&(5,\{\mathsf{ok} : y?(2,n).y!\langle 5,\mathtt{date}\rangle.\mathbf{0},\ \mathsf{quit} : \mathbf{0}\})
\end{array}
$$

Fig. 4. The manager-factories example in complete syntax.

$$a[1](y).P_1 \mid \ldots \mid a[\mathtt{n}-1](y).P_{\mathtt{n}-1} \mid \overline{a}[\mathtt{n}](y).P_{\mathtt{n}} \longrightarrow (\nu s)(P_1\{s[1]/y\} \mid \ldots \mid P_{\mathtt{n}-1}\{s[\mathtt{n}-1]/y\} \mid P_{\mathtt{n}}\{s[\mathtt{n}]/y\}) \qquad \text{[Link]}$$

$$s[\mathtt{p}]!\langle\Pi\cup\{\mathtt{q}\},e\rangle.P \mid s[\mathtt{q}]?(\mathtt{p},x).Q \longrightarrow s[\mathtt{p}]!\langle\Pi,e\rangle.P \mid Q\{v/x\} \quad (e\downarrow v) \quad \text{if } \Pi\neq\emptyset \text{ and } \mathtt{q}\notin\Pi \qquad \text{[Comm]}$$

$$s[\mathtt{p}]!\langle\{\mathtt{q}\},e\rangle.P \mid s[\mathtt{q}]?(\mathtt{p},x).Q \longrightarrow P \mid Q\{v/x\} \quad (e\downarrow v) \qquad \text{[Comm1]}$$

$$s[\mathtt{p}]!\langle\langle\mathtt{q},s'[\mathtt{p}']\rangle\rangle.P \mid s[\mathtt{q}]?((\mathtt{p},y)).Q \longrightarrow P \mid Q\{s'[\mathtt{p}']/y\} \qquad \text{[Deleg]}$$

$$s[\mathtt{p}] \oplus \langle\Pi\cup\{\mathtt{q}\},l_{i_0}\rangle.P \mid s[\mathtt{q}]\&(\mathtt{p},\{l_i : P_i\}_{i\in I}) \longrightarrow s[\mathtt{p}] \oplus \langle\Pi,l_{i_0}P\rangle. \mid P_{i_0} \quad (i_0\in I) \quad \text{if } \Pi\neq\emptyset \text{ and } \mathtt{q}\notin\Pi \qquad \text{[Label]}$$

$$s[\mathtt{p}] \oplus \langle\{\mathtt{q}\},l_{i_0}\rangle.P \mid s[\mathtt{q}]\&(\mathtt{p},\{l_i : P_i\}_{i\in I}) \longrightarrow P \mid P_{i_0} \quad (i_0\in I) \qquad \text{[Label1]}$$

$$\textsf{if } e \textsf{ then } P \textsf{ else } Q \longrightarrow P \quad (e\downarrow \textsf{true}) \qquad \textsf{if } e \textsf{ then } P \textsf{ else } Q \longrightarrow Q \quad (e\downarrow \textsf{false}) \qquad \text{[If-T], [If-F]}$$

$$\textsf{def } X(x,y)=P \textsf{ in } (X\langle e,s[\mathtt{p}]\rangle \mid Q) \longrightarrow \textsf{def } X(x,y)=P \textsf{ in } (P\{v/x\}\{s[\mathtt{p}]/y\} \mid Q) \quad (e\downarrow v) \qquad \text{[ProcCall]}$$

$$P\longrightarrow P' \quad\Rightarrow\quad (\nu r)\,P\longrightarrow(\nu r)\,P' \qquad P\longrightarrow P' \quad\Rightarrow\quad P\mid Q\longrightarrow P'\mid Q \qquad \text{[Scop], [Par]}$$

$$P\longrightarrow P' \quad\Rightarrow\quad \textsf{def } D \textsf{ in } P\longrightarrow \textsf{def } D \textsf{ in } P' \qquad \text{[Defin]}$$

$$P\equiv P' \text{ and } P'\longrightarrow Q' \text{ and } Q\equiv Q' \quad\Rightarrow\quad P\longrightarrow Q \qquad \text{[Str]}$$

Table 3
Reduction rules.

$$P\mid\mathbf{0}\equiv P \quad P\mid Q\equiv Q\mid P \quad (P\mid Q)\mid R\equiv P\mid(Q\mid R) \quad (\nu r)P\mid Q\equiv(\nu r)(P\mid Q)$$

$$(\nu rr')P\equiv(\nu r'r)P \quad (\nu r)\mathbf{0}\equiv\mathbf{0} \quad \textsf{def } D \textsf{ in } \mathbf{0}\equiv\mathbf{0} \quad \textsf{def } D \textsf{ in } (\nu r)P\equiv(\nu r)\textsf{def } D \textsf{ in } P$$

$$(\textsf{def } D \textsf{ in } P)\mid Q\equiv \textsf{def } D \textsf{ in } (P\mid Q) \quad \textsf{def } D \textsf{ in } (\textsf{def } D' \textsf{ in } P)\equiv \textsf{def } D' \textsf{ in } (\textsf{def } D \textsf{ in } P)$$

Table 4
Structural equivalence.

example in complete syntax. According to the global type G_a all input/output operations specify also the sender and the receivers, respectively.

3.3 Operational Semantics

As usual the operational semantics consists of reduction rules (Table 3) and structural equivalence rules (Table 4) that allow to rearrange the terms in order to apply the reduction rules. In writing processes we use "Barendregt's convention", i.e. we assume that all bound names/variables are different from one another and from the free ones.

Rule [Link] describes the initiation of a new session among n participants that synchronise over the service name a. The last participant $\overline{a}[\mathtt{n}](y).P_{\mathtt{n}}$, distinguished by the overbar on the service name, specifies the number $\mathtt{n}$ of participants. For this reason we call it the *initiator* of the session. Obviously each session must have a unique initiator. After the connection, the participants will share the private session name s. The variables y in each participant $\mathtt{p}$ will then be replaced with the corresponding channel with role $s[\mathtt{p}]$.

Session	T	::=	$!\langle\Pi, U\rangle.T$	*send*	|	$\mu\mathbf{t}.T$	*recursive*
		|	$?(\mathtt{p}, U).T$	*receive*	|	$\mathbf{t}$	*variable*
		|	$\oplus\langle\Pi, \{l_i : T_i\}_{i\in I}\rangle$	*selection*	|	end	*end*
		|	$\&(\mathtt{p}, \{l_i : T_i\}_{i\in I})$	*branching*			

Table 5
Session types.

The communication rules [Comm], [Comm1], [Deleg], [Label] and [Label1] formalise a communication between two processes inside a session. We need to remove the receiving participant from the set of the receivers in order to avoid exchanging the same message more than once. If there is only one receiver (rules [Comm1], [Deleg] and [Label1]) the whole sending action is removed. Rule [Link] is synchronous and atomic, involving all participants at once. Message sender and receiver must synchronise, but multicast communication is not atomic, since the receivers get the same message in different reduction steps. These choices are debatable, but they are quite common in the literature [5].

In the following we denote with $e{\downarrow}\upsilon$ the evaluation of the expression e to the value υ and we use $\longrightarrow^*$ with the expected meaning.

4 Type System

This section introduces the type system, which assures soundness of communications.

4.1 Global Types

Global types, ranged over by $G, G', ..$ describe the whole conversation scenarios of multiparty sessions. The grammar is:

Global	G	::=	$\mathtt{p} \to \Pi : \langle U\rangle.G'$	Exchange	U	::=	$S \mid T$
		|	$\mathtt{p} \to \Pi : \{l_i : G_i\}_{i\in I}$	Sorts	S	::=	$\mathsf{bool} \mid \ldots \mid G$
		|	$\mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$				

The global type $\mathtt{p} \to \Pi : \langle U\rangle.G'$ says that participant $\mathtt{p}$ multicasts a message of type U to the set of participants Π and then the interactions described in G' take place. *Exchange types* $U, U', ...$ consist of *sorts* $S, S', \ldots$ for values (either base types or global types), and *session types* $T, T', \ldots$ for channels (discussed in § 4.2). The global type $\mathtt{p} \to \Pi : \{l_i : G_i\}_{i\in I}$ says participant $\mathtt{p}$ multicasts one of the labels l_i to the set of participants Π. If l_j is sent, interactions described in the global type G_j take place. In both cases we assume $\mathtt{p} \notin \Pi$. The global type $\mu\mathbf{t}.G$ is a recursive type, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e., type variables only appear under some prefix. We assume that G in the grammar of sorts is closed, i.e., without free type variables. Type end represents the termination of the session.

4.2 Session Types

Local types of processes, called *session types* are defined in Table 5. While global types represent whole protocols, session types correspond to the view-points of single participants in sessions. The *send type* $!\langle\Pi, U\rangle.T$ expresses the sending to all participants in Π of a value or of a channel of type U, followed by the communications of T. The *selection type* $\oplus\langle\Pi, \{l_i : T_i\}_{i\in I}\rangle$ represents the transmission to all participants in Π of a label l_i

$$(\mathtt{p} \to \Pi : \langle U \rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !\langle \Pi, U \rangle.(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, U).(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} \in \Pi, \\ G' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\mathtt{p} \to \Pi : \{l_i : G_i\}_{i \in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus \langle \Pi, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I} \rangle & \text{if } \mathtt{q} = \mathtt{p} \\ \& (\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} \in \Pi \\ G_{i_0} \upharpoonright \mathtt{q} & \text{where } i_0 \in I \text{ if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \notin \Pi \text{ and } G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} \text{ for all } i, j \in I. \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright \mathtt{q} = \begin{cases} \mu \mathbf{t}.(G \upharpoonright \mathtt{q}) & \text{if } G \upharpoonright \mathtt{q} \neq \mathbf{t}, \\ \mathsf{end} & \text{otherwise.} \end{cases} \qquad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \qquad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}.$$

Table 6
Projection of global types onto participants.

$$\Gamma, u : S \vdash u : S \quad \lfloor \text{Name} \rfloor \qquad \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{bool} \quad \lfloor \text{Bool} \rfloor \qquad \frac{\Gamma \vdash e_i : \mathsf{bool}}{\Gamma \vdash e_1 \ \mathsf{and} \ e_2 : \mathsf{bool}} \ \lfloor \text{And} \rfloor$$

Table 7
Some typing rules for expressions.

$$\frac{\Gamma \vdash u : G \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p} \quad \mathrm{pn}(G) = \mathtt{p}}{\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta} \ \lfloor \text{MCast} \rfloor \qquad \frac{\Gamma \vdash u : G \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p}}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta} \ \lfloor \text{MAcc} \rfloor$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \Pi, e \rangle.P \triangleright \Delta, c : !\langle \Pi, S \rangle.T} \ \lfloor \text{Send} \rfloor \qquad \frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathtt{q}, x).P \triangleright \Delta, c : ?(\mathtt{q}, S).T} \ \lfloor \text{Rcv} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\langle \mathtt{p}, c' \rangle\rangle.P \triangleright \Delta, c : !\langle \{\mathtt{p}\}, T' \rangle.T, c' : T'} \ \lfloor \text{Deleg} \rfloor \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : T'}{\Gamma \vdash c?((\mathtt{q}, y)).P \triangleright \Delta, c : ?(\mathtt{q}, T').T} \ \lfloor \text{Srec} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash c \oplus \langle \Pi, l_j \rangle.P \triangleright \Delta, c : \oplus \langle \Pi, \{l_i : T_i\}_{i \in I} \rangle} \ \lfloor \text{Sel} \rfloor \qquad \frac{\Gamma \vdash P_i \triangleright \Delta, c : T_i \quad \forall i \in I}{\Gamma \vdash c \& (\mathtt{p}, \{l_i : P_i\}_{i \in I}) \triangleright \Delta, c : \& (\mathtt{p}, \{l_i : T_i\}_{i \in I})} \ \lfloor \text{Branch} \rfloor$$

Table 8
Typing rules for processes I.

chosen in the set $\{l_i \mid i \in I\}$ followed by the communications described by T_i. The *receive* and *branching* are the dual types, respectively, and only need one sender. Other types are standard. Following [10, §20.2] we take an *equi-recursive* view of recursive types, equating types with the same (possibly infinite) regular tree.

The relation between session and global types is formalised by the notion of projection as in [7]. The *projection of G onto* $\mathtt{q}$ ($G \upharpoonright \mathtt{q}$) is defined by induction on G, see Table 6.

4.3 Typing Rules for Complete Processes

The typing judgements for expressions and processes are of the shape:

$$\Gamma \vdash e : S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

where Γ is the *standard environment* which associates variables with sorts, service and session names to global types and process variables to pairs of sorts and session types; Δ is the *session environment* which associates channels with session types. Formally we define:

$$\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, s : G \mid \Gamma, X : S\,T \quad \text{and} \quad \Delta ::= \emptyset \mid \Delta, c : T$$

assuming that we can write $\Gamma, u : S$ only if u does not occur in Γ, briefly $u \notin dom(\Gamma)$ ($dom(\Gamma)$ denotes the domain of Γ, i.e., the set of identifiers which occur in Γ). We use the same convention for $X : S\,T$ and Δ (thus we can write Δ, Δ' only if $dom(\Delta) \cap dom(\Delta') = \emptyset$).

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} \lfloor \text{Par} \rfloor \qquad \frac{\Gamma, a : G \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta} \lfloor \text{NRes} \rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta} \lfloor \text{If} \rfloor$$

$$\frac{\Delta\ \text{end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \lfloor \text{Inact} \rfloor \qquad \frac{\Gamma \vdash e : S \quad \Delta\ \text{end only}}{\Gamma, X : S\,T \vdash X\langle e, c \rangle \triangleright \Delta, c : T} \lfloor \text{Var} \rfloor$$

$$\frac{\Gamma, X : S\,T,\ x : S \vdash P \triangleright y : T \qquad \Gamma, X : S\,T \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(x,y) = P\ \mathsf{in}\ Q \triangleright \Delta} \lfloor \text{Def} \rfloor$$

$$\frac{\Gamma, s : G \vdash P \triangleright \Delta, \{s[\mathtt{p}] : G \upharpoonright \mathtt{p} \mid 1 \leq \mathtt{p} \leq \mathtt{n}\} \quad s \notin Dom(\Delta) \quad \mathtt{n} = \mathrm{pn}(G)}{\Gamma \vdash (\nu s) P \triangleright \Delta} \lfloor \text{SRes} \rfloor$$

Table 9
Typing rules for processes II.

Table 7 gives some typing rules for expressions. Tables 8 and 9 present the typing rules for processes. Rule $\lfloor$MCast$\rfloor$ permits to type a service initiator identified by u, if the type of y is the p-th projection of the global type G of u and the number of participants in G (denoted by $\mathrm{pn}(G)$) is p. Rule $\lfloor$MAcc$\rfloor$ permits to type the p-th participant in the service identified by u, which uses the channel y, if the type of y is the p-th projection of the global type G of u. The successive six rules associate the input/output processes with the input/output types in the expected way. Note that, according to our notational convention on environments, in rule $\lfloor$Deleg$\rfloor$ the channel which is sent cannot appear in the session environment of the premise, i.e., $c' \notin dom(\Delta) \cup \{c\}$. Rule $\lfloor$Par$\rfloor$ permits to put in parallel two processes only if their sessions environments have disjoint domains. A session environment is "end only" when all channels are typed by end. Rule $\lfloor$SRes$\rfloor$ restricts a session s only if the session environment contains the session channels of all participants in the global type G, which is the type of s in the standard environment. Moreover these session channels must be typed with the projections of G on the respective participants.

4.4 Subject Reduction

Since session environments represent the forthcoming communications, by reducing processes session environments can change. This can be formalised as in [7] by introducing the notion of reduction of session environments, whose rules are:

- $\{s[\mathtt{p}] : !\langle \Pi \cup \{\mathtt{q}\}, U\rangle.T,\ s[\mathtt{q}] : ?(\mathtt{p}, U).T'\} \Rightarrow \{s[\mathtt{p}] : !\langle \Pi, U\rangle.T,\ s[\mathtt{q}] : T'\}$ $\quad \Pi \neq \emptyset$ and $\mathtt{q} \notin \Pi$
- $\{s[\mathtt{p}] : !\langle \{\mathtt{q}\}, U\rangle.T,\ s[\mathtt{q}] : ?(\mathtt{p}, U).T'\} \Rightarrow \{s[\mathtt{p}] : T,\ s[\mathtt{q}] : T'\}$
- $\{s[\mathtt{p}] : \oplus\langle \Pi \cup \{\mathtt{q}\}, \{l_i : T_i\}_{i\in I}\rangle,\ s[\mathtt{q}] : \&(\mathtt{p}, \{l_i : T'_i\}_{i\in I})\} \Rightarrow \{s[\mathtt{p}] : \oplus\langle \Pi, \{l_i : T_i\}_{i\in I}\rangle, s[\mathtt{q}] : T'_i\}$ $\quad \Pi \neq \emptyset$ and $\mathtt{q} \notin \Pi$
- $\{s[\mathtt{p}] : \oplus\langle \{\mathtt{q}\}, \{l_i : T_i\}_{i\in I}\rangle,\ s[\mathtt{q}] : \&(\mathtt{p}, \{l_i : T'_i\}_{i\in I})\} \Rightarrow \{s[\mathtt{p}] : T_i,\ s[\mathtt{q}] : T'_i\}$
- $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta''$ if $\Delta \Rightarrow \Delta'$

The first two rules correspond to the exchange of a value or channel from the participant p to the participant q, the third and fourth rules correspond to the choice of the label l_i by the participant p.

Subject reduction only holds if the types of the channels with role are the projections of the global type of the corresponding session. More formally:

$$\frac{\Theta, u : G, y : G \upharpoonright \mathtt{p} \vdash P \Rightarrow P^* \quad \mathrm{pn}(G) = \mathtt{p}}{\Theta, u : G \vdash \overline{u}[\mathtt{p}](y).P \Rightarrow \overline{u}[\mathtt{p}](y).P^*}\ (MCast)$$

$$\frac{\Theta, u : G, y : G \upharpoonright \mathtt{p} \vdash P \Rightarrow P^*}{\Theta, u : G \vdash u[\mathtt{p}](y).P \Rightarrow u[\mathtt{p}](y).P^*}\ (MAcc)$$

$$\frac{\Theta, y : T \vdash P \Rightarrow P^* \quad \Theta \vdash e : S}{\Theta, y : !\langle \Pi, S\rangle.T \vdash y!\langle e\rangle.P \Rightarrow y!\langle \Pi, e\rangle.P^*}\ (Send)$$

$$\frac{\Theta, x : S, y : T \vdash P \Rightarrow P^*}{\Theta, y : ?(\mathtt{q}, S).T \vdash y?(x).P \Rightarrow y?(\mathtt{q}, x).P^*}\ (Rcv)$$

$$\frac{\Theta, y : T \vdash P \Rightarrow P^*}{\Theta, y : !\langle\{\mathtt{p}\}, T'\rangle.T, z : T' \vdash y!\langle\langle z\rangle\rangle.P \Rightarrow y!\langle\langle \mathtt{p}, z\rangle\rangle.P^*}\ (Deleg)$$

$$\frac{\Theta, y : T, z : T' \vdash P \Rightarrow P^*}{\Theta, y : ?(\{\mathtt{p}\}, T').T \vdash y?((z)).P \Rightarrow y?((\mathtt{p}, z)).P^*}\ (Srec)$$

$$\frac{\Theta, y : T_j \vdash P \Rightarrow P^* \quad j \in I}{\Theta, y : \oplus\langle \Pi, \{l_i : T_i\}_{i\in I}\rangle \vdash y \oplus \langle \Pi, l_j\rangle.P \Rightarrow y \oplus \langle \Pi, l_j\rangle.P^*}\ (Sel)$$

$$\frac{\Theta, y : T_i \vdash P_i \Rightarrow P_i^* \quad \forall i \in I}{\Theta, y : \&(\mathtt{p}, \{l_i : T_i\}_{i\in I}) \vdash y\&\{l_i : P_i\}_{i\in I}\mathtt{p} \Rightarrow y\&(\mathtt{p}, \{l_i : P_i^*\}_{i\in I})}\ (Branch)$$

Table 10
Translation rules I

Definition 4.1 A session environment Δ is *coherent* for the standard environment Γ if $s[\mathtt{p}] : T \in \Delta$ implies $s : G \in \Gamma$ and $G \upharpoonright \mathtt{p} = T$.

We can state type preservation under reduction as follows:

Theorem 4.2 (Type Preservation) *If $\Gamma \vdash P \triangleright \Delta$ with Δ coherent for Γ and $P \longrightarrow^* P'$, then $\Gamma \vdash P' \triangleright \Delta'$ for some coherent Δ' such that $\Delta \Rightarrow^* \Delta'$.*

The condition of coherence is needed to show subject reduction when rule [Comm] or [Comm1] is applied. Type preservation is proved in [3], where communications are asynchronous. In the synchronous case the proof is easier.

5 Natural Semantics Translation

The translation for completing processes is defined via a set of formal rules in natural semantics. The rules are defined following the syntax of the processes and so they provide a basis to define a recursive deterministic procedure implementing the translation.

A *translation environment* Θ is defined by:

$$\Theta ::= \emptyset \mid \Theta, u : S \mid \Theta, X : S\ T \mid \Theta, y : T$$

i.e. it contains assumptions of both standard and session environments. As for session environments, a translation environment is "end only" if all channel variables are typed by end. Two translation environments Θ and Θ' are *compatible* (notation $\Theta \asymp \Theta'$) when their domains restricted to channel variables are disjoint.

A translation environment can clearly be split into a standard environment and a session

$$\frac{\Theta \vdash P \Rightarrow P^* \quad \Theta' \vdash Q \Rightarrow Q^* \quad \Theta \asymp \Theta'}{\Theta \cup \Theta' \vdash P \mid Q \Rightarrow P^* \mid Q^*}\;(Par)$$

$$\frac{\Theta \vdash P \Rightarrow P^* \quad \Theta \vdash Q \Rightarrow Q^* \quad \Theta \vdash e : \mathsf{bool}}{\Theta \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \Rightarrow \mathsf{if}\ e\ \mathsf{then}\ P^*\ \mathsf{else}\ Q^*}\;(If)$$

$$\frac{\Theta\ \mathsf{end\ only}}{\Theta \vdash \mathbf{0} \Rightarrow \mathbf{0}}\;(Inact) \qquad \frac{\Theta, a : G \vdash P \Rightarrow P^*}{\Theta \vdash (\nu a : G)P \Rightarrow (\nu a : G)P^*}\;(NRes)$$

$$\frac{\Theta \vdash e : S \quad \Theta\ \mathsf{end\ only}}{\Theta, y : T, X : ST \vdash X\langle e, y\rangle \Rightarrow X\langle e, y\rangle}\;(Var) \qquad \frac{\Theta \vdash Q \Rightarrow Q^* \quad X \notin dom(\Theta)}{\Theta \vdash \mathsf{def}\ X(x,y) = P\ \mathsf{in}\ Q \Rightarrow Q^*}\;(DefS)$$

$$\frac{\Gamma_\Theta, X : ST, x : S, y : T \vdash P \Rightarrow P^* \quad \Theta, X : ST \vdash Q \Rightarrow Q^*}{\Theta \vdash \mathsf{def}\ X(x,y) = P\ \mathsf{in}\ Q \Rightarrow \mathsf{def}\ X(x,y) = P^*\ \mathsf{in}\ Q^*}\;(Def)$$

Table 11
Translation rules II

$$\mathscr{D} \quad \frac{\Theta' \vdash \mathbf{nn}(x) : \mathsf{nat} \quad \Theta'\ \mathsf{end\ only}}{\Theta'', t' : T \vdash X'\langle \mathbf{nn}(x), t'\rangle \Rightarrow X'\langle \mathbf{nn}(x), t'\rangle} \qquad \frac{\Theta', t' : \mathsf{end} \quad \mathsf{end\ only}}{\Theta'', t' : \mathsf{end} \vdash \mathbf{0} \Rightarrow \mathbf{0}}$$

$$\frac{\frac{\frac{\frac{\cdots}{\Theta'', t' : \&(3, \{\mathsf{ok}\ldots\}) \vdash t'\&\{\ldots\} \Rightarrow t'\&(3, \{\mathsf{ok}\ldots\})}}{\Theta'', t' : T \vdash t'!\langle x\rangle \ldots \Rightarrow t'!\langle \{3\}, x\rangle \ldots} \qquad \frac{\Theta \vdash \mathtt{num} : \mathsf{nat} \quad \Theta\ \mathsf{end\ only}}{\Theta, y : T, X' : \mathsf{nat}\ T \vdash X'\langle \mathtt{num}, y\rangle \Rightarrow X'\langle \mathtt{num}, y\rangle}}{\Theta, y : T \vdash \mathsf{def}\ X'(x,t') = \ldots \mathsf{in}\ X'\langle \mathtt{num}, y\rangle \Rightarrow \mathsf{def}\ X'(x,t') = \ldots \mathsf{in}\ X'\langle \mathtt{num}, y\rangle}}{a : G_a, b : G_b, y :?(3, \mathsf{string}).T \vdash y?(id)\ldots \Rightarrow y?(3, id)\ldots}}{a : G_a, b : G_b \vdash \mathtt{F}[1] \Rightarrow a[1](y)\ldots}$$

where $\Theta = a : G_a, b : G_b, id : \mathsf{string}$, $T = \mu\mathbf{t}.!\langle \{3\}, \mathsf{nat}\rangle.\&(3, \{\mathsf{ok} :!\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}, \mathsf{more} : \mathbf{t}, \mathsf{quit} :\mathsf{end}\})$
$\Theta' = \Theta, x : \mathsf{nat}$, $\Theta'' = \Theta', X' : \mathsf{nat}\ T$
`num` is short for `number`, **nn** is short for **newnumber** and $\mathscr{D}$ is the derivation

$$\frac{\frac{\frac{\Theta'', t' :!\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}, z : \mathsf{end} \vdash \mathbf{0} \Rightarrow \mathbf{0}}{\Theta'', t' :!\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}, z : \langle 3, !\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}\rangle.\mathsf{end} \vdash z!\langle\langle t'\rangle\rangle.\mathbf{0} \Rightarrow z!\langle\langle 3, t'\rangle\rangle.\mathbf{0}}}{\Theta'', t' :!\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}, z :!\langle \{3\}, \mathsf{nat}\rangle.\langle 3, !\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end}\rangle.\mathsf{end} \vdash z!\langle x\rangle.z!\langle\langle t'\rangle\rangle.\mathbf{0} \Rightarrow z!\langle \{3\}, x\rangle.z!\langle\langle 3, t'\rangle\rangle.\mathbf{0}}}{\Theta'', t' :!\langle \{3\}, \mathsf{nat}\rangle.\mathsf{end} \vdash b[1](z).z!\langle x\rangle.z!\langle\langle t'\rangle\rangle.\mathbf{0} \Rightarrow b[1](z).z!\langle \{3\}, x\rangle.z!\langle\langle 3, t'\rangle\rangle.\mathbf{0}}$$

Fig. 5. Translation of $\mathtt{F}[1]$ in Figure 3.

environment. We define

$$\Delta_\Theta = \{y : T \mid y : T \in \Theta\} \text{ and } \Gamma_\Theta = \Theta \setminus \Delta_\Theta$$

The typing rules for expressions can be easily modified to allow translation environments instead of standard environments. The translation judgments have the shape:

$$\Theta \vdash P \Rightarrow P^*$$

meaning that from the environment Θ the process P in partial syntax is translated to the process P^* in complete syntax. A translation judgment is *well formed* when the sets of process variables in Θ and in P^* coincide. We assume that all translation judgments are well formed. The translation rules are given in Tables 10 and 11. Rule $(DefS)$ is needed since, if there are no calls to X, then the global type does not contain a type for the channel y. In this case the translated process is simplified, since the recursive definition is erased.

As an example Figure 5 shows the application of this translation rules to the process $\mathtt{F}[1]$ of Figure 3.

The translation rules include type checking in the sense that, if a translation statement is provable, then the corresponding complete process is typable using the standard session environments obtained by splitting the translation environment. More precisely we have the following soundness result:

Theorem 5.1 (Soundness of the translation rules) *If* $\Theta \vdash P \Rightarrow P^*$, *then* $\Gamma_\Theta \vdash P^* \triangleright \Delta_\Theta$.

Notice that all recursive definitions in P^* are called at least once in their bodies.

Let $|P|$ be the partial process obtained from the complete process P by erasing senders and receivers. If a complete process P without channels with role is typable, then the judgment which states the translation of $|P|$ is provable. The process resulting from this translation coincides with the original P, but for erasing recursive definitions of process variables which are never called. More precisely:

Theorem 5.2 (Completeness of the translation rules) *Let P be a complete process without channels with role. If $\Gamma \vdash P \triangleright \Delta$, then $\Gamma' \cup \Delta \vdash |P| \Rightarrow \hat{P}$, where $\hat{P}$ is the process resulting from P by erasing unused definitions and Γ' is the restriction of Γ to the process variables which occur in $\hat{P}$.*

Translation rule $(DefS)$ requires to use Γ' instead of Γ. Notice that the restriction to complete processes without channels with role is needed, since otherwise the mapping $|\ |$ becomes meaningless.

The proofs of these theorems can be done by induction on the translation rules and the typing rules, respectively. These proofs are standard, thanks to the similarity between translation and typing rules.

Following the translation rules it is rather easy to design an algorithm to realise them. We split each translation environment Θ in two disjoint parts, Υ and Ξ where:

- Υ, the *definition environment*, contains only assumptions of the shape $X : S\ T$;
- Ξ, the *basic* environment, contains any kind of assumptions.

The algorithm, written as the recursive function `trans` in a ML-like language, is given in Table 12. This function takes as input a basic environment Ξ and a partial process P and returns a pair formed by a definition environment Υ, which collects the types of the process variables free in P, and a complete process P^*.

We introduce now the notation used in Table 12. The function $\mathsf{typeof}(e, \Xi)$ gives the type of the expression e in the standard environment Γ_Ξ, if any. We define

$$\Xi^P = \{y : T \mid y : T \in \Xi \text{ and } y \text{ has at least one occurrence in } P\} \cup$$
$$\{u : S \mid u : S \in \Xi\} \cup \{X : ST \mid X : ST \in \Xi\}$$

This is useful to split the typing of the session channels in translating the parallel composition of processes.

The union $\uplus$ between definition environments is defined only if the same process variables have the same sorts and session types (as trees). We assume unions between definition environments to be always defined when we write them in Table 12, otherwise the `trans` function fails. We can always assume that session types are written in forms different from

function $\texttt{trans}(\Xi\ ;\ R)$

case R of

- $\overline{u}[\mathtt{p}](y).P$:	if $\Xi(u) = G$ and $\mathtt{p} = \mathrm{pn}(G)$ then let $T = G \upharpoonright \mathtt{p}$ and $(\Upsilon,\ P^*) = \texttt{trans}(\Xi, y : T\ ;\ P)$ in return $(\Upsilon,\ \overline{u}[\mathtt{p}](y).P^*)$ else FAIL
- $u[\mathtt{p}](y).P$:	if $\Xi(u) = G$ and $\mathtt{p} < \mathrm{pn}(G)$ then let $T = G \upharpoonright \mathtt{p}$ and $(\Upsilon,\ P^*) = \texttt{trans}(\Xi, y : T\ ;\ P)$ in return $(\Upsilon,\ u[\mathtt{p}](y).P^*)$ else FAIL
- $y!\langle e\rangle.P$:	if $\Xi = \Xi', y : !\langle \Pi, S\rangle.T$ and $\mathrm{typeof}(e,\ \Xi) = S$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi', y : T\ ;\ P)$ in return $(\Upsilon, y!\langle \Pi, e\rangle.P^*)$ else FAIL
- $y?(x).P$:	if $\Xi = \Xi', y : ?(\mathtt{p}, S).T$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi', x : S, y : T\ ;\ P)$ in return $(\Upsilon, y?(\mathtt{p}, x).P^*)$ else FAIL
- $y!\langle\langle z\rangle\rangle.P$:	if $\Xi = \Xi', y : !\langle \mathtt{p}, T'\rangle.T, z : T'$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi', y : T\ ;\ P)$ in return $(\Upsilon, y!\langle\langle \mathtt{p}, z\rangle\rangle.P^*)$ else FAIL
- $y?((z)).P$:	if $\Xi = \Xi', y : ?(\mathtt{p}, T').T$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi', y : T, z : T'\ ;\ P)$ in return $(\Upsilon, y?((\mathtt{p}, z)).P^*)$ else FAIL
- $y \oplus l_j.P$:	if $\Xi = \Xi', y : \oplus\langle \Pi, \{l_i : T_i\}_{i\in I}\rangle$ and $j \in I$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi', y : T_j\ ;\ P)$ in return $(\Upsilon, y \oplus \langle \Pi, l_j\rangle.P^*)$ else FAIL
- $y\&\{l_i : P_i\}_{i\in I}$:	if $\Xi = \Xi', y : \&(\mathtt{p}, \{l_i : T_i\}_{i\in I})$ then for all $i \in I$ let $(\Upsilon_i,\ P_i^*) = \texttt{trans}(\Xi', y : T_i\ ;\ P_i)$ in return $(\uplus_{i\in I}\Upsilon_i, y\&(\mathtt{p}, \{l_i : P_i^*\}_{i\in I}))$ else FAIL
-if e then P else Q:	if $\mathrm{typeof}(e,\ \Xi) = \mathrm{bool}$ then let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi\ ;\ P)$ and $(\Upsilon',\ Q^*) = \texttt{trans}(\Xi\ ;\ Q)$ in return $(\Upsilon \uplus \Upsilon',$ if e then P^* else $Q^*)$ else FAIL
- $P \mid Q$:	let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi^P\ ;\ P)$ and $(\Upsilon',\ Q^*) = \texttt{trans}(\Xi^Q\ ;\ Q)$ in return $(\Upsilon \uplus \Upsilon',\ P^* \mid Q^*)$
- $\mathbf{0}$:	return $(\emptyset,\ \mathbf{0})$
- $(\nu a : G)\ P$:	let $(\Upsilon,\ P^*) = \texttt{trans}(\Xi, a : G\ ;\ P^*)$ in return $(\Upsilon,\ (\nu a : G)\ P^*)$
- $X\langle e, y\rangle$:	let $S = \mathrm{typeof}(e, \Xi)$ and $\Xi(y) = T$ in if $X \notin dom(\Xi)$ then return $(\{X : S\ T\},\ X\langle e, y\rangle)$ else if $\Xi(X) = S\ T$ then return $(\emptyset,\ X\langle e, y\rangle)$ else FAIL
- def $X(x, y) = P$ in Q :	let $(\Upsilon,\ Q^*) = \texttt{trans}(\Xi\ ;\ Q)$ in if $X \notin dom(\Upsilon)$ then return $(\Upsilon,\ Q^*)$ else let $\Upsilon = \Upsilon', X : S\ T$ and $(\Upsilon'', P^*) = \texttt{trans}(\Xi, y : T, x : S, X : S\ T\ ;\ P)$ in return $(\Upsilon \uplus \Upsilon'',$ def $X(x, y) = P^*$ in $Q^*)$

Table 12
The translation function.

$\mu\mathbf{t}.T$, by unfolding them when needed.

The more interesting cases in the definition of the function `trans` are the last two. To translate a recursion def $X(x, y) = P$ in Q we first compute the type of X inside Q by calling $\texttt{trans}(\Xi\ ;\ Q)$. When `trans` is applied to a call $X\langle e, z\rangle$ inside Q it adds the assumption $X : S\ T$ (where S is the sort of the expression e and T is the type of z) to the definition

environment. If Q does not contain calls of X we simply erase the declaration. Otherwise we check that P can be typed from the environment $\Xi, y : T, x : S, X : S\ T$. When `trans` is applied to a call $X\langle e', y\rangle$ inside P it only checks that the types of X and y agree, i.e. that the session types of X and y have the same tree.

The following Theorems state the correctness and completeness of the `trans` function. They can be shown by induction on the definition of `trans` and on the translation rules, respectively.

Theorem 5.3 (Soundness of `trans`) *If* $\texttt{trans}(\Xi\ ;\ P) = (\Upsilon, P^*)$, *then* $\Xi', \Upsilon \vdash P \Rightarrow P^*$, *where* Ξ' *is the restriction of* Ξ *to the process variables which occur in* P^*.

Theorem 5.4 (Completeness of `trans`) *If* $\Xi, \Upsilon \vdash P \Rightarrow P^*$, *then* $\texttt{trans}(\Xi\ ;\ P) = (\Upsilon, P^*)$.

We say that a process is *closed* if the only names occurring free in it are service names. Let Θ_0 be an environment containing only assumptions for service names and P a closed partial process. If the call $\texttt{trans}(\Theta_0\ ;\ P)$ terminates without rising errors, then the result is of the shape $(\emptyset, P^*)$, where P^* is the well-typed translation of P and the definition environment is empty.

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions and remarks.

References

[1] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[2] Eduardo Bonelli and Adriana Compagnoni. Multipoint Session Types for a Distributed Calculus. In Gilles Barthe and Cédric Fournet, editors, *TGC'07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.

[3] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*. Springer, 2015. To appear.

[4] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-Adaptive Monitors for Multiparty Sessions. In Alberto Alberto Lluch-Lafuente and Emilio Tuosto, editors, *PDP'14*, pages 688–696. IEEE, 2014.

[5] Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and Session Types: an Overview. In Cosimo Laneve and Jianwen Su, editors, *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.

[6] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In Chris Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[7] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In George C. Necula and Philip Wadler, editors, *POPL'08*, pages 273–284. ACM Press, 2008.

[8] Hans Hüttel, Ivan Lanese, Vasco Thudichum Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Behavioural Types. http://www.behavioural-types.eu/publications/WG1-State-of-the-Art.pdf, 2014.

[9] Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. *Logical Methods in Computer Science*, 10, 2014.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[12] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *TGC'13*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.