

# Dependent Session Types for Evolving Multiparty Communication Topologies

Nobuko Yoshida   Pierre-Malo Deniélou   Andi Bejleri   Raymond Hu

Department of Computing, Imperial College London

## Abstract

Many application-level distributed protocols and parallel algorithms are dynamic in nature: the number of participants, messages or repetitions is only known at run-time, and the communication topology may be altered during the execution. This paper proposes a dependent type theory for multiparty sessions which can statically guarantee type-safe, deadlock-free multiparty interactions among processes with dynamically evolving communication topologies. We use the primitive recursion operator from Gödel's System  $\mathcal{T}$  along with dependent product types to express a wide range of topologies where the structure of the multiparty communications depend on numerical parameters that are instantiated at run-time. To type individual distributed processes, a parameterised global type is projected onto a generic generator which represents a class of all possible end-point types. Termination of the type-checking algorithm is proved with the full multiparty session types including recursive types. The expressiveness of our type theory is demonstrated through non-trivial programming and verification examples with complex communication topologies taken from distributed parallel algorithms and Web services usecases.

## 1. Introduction

As the momentum around communications-based programming and software grows, the need for effective frameworks to coordinate the interactions among distributed peers is pressing. Such coordination typically *structures* the interactions, with respect to the sequences in which messages are transmitted and received, their compositions, choices and repetitions, to form a larger composite, meaningful multiparty protocol. An aspect that is particularly challenging is the fact that most actual communication protocols are *dynamic*, in the sense that the number of participants, repetitions within the interactions and the communication topologies cannot be fixed at design time; rather, the identities of some participants involved in a protocol and how they are connected may only be determined at run-time, and may even change as the protocol progresses.

This paper provides a robust type-based verification methodology to statically ensure communication-safe, deadlock-free interactions in dynamic multiparty protocols. Our motivation is to be able to “program” evolving global specifications as a minimal, tractable extension from multiparty session types [6, 7, 21, 25]. The diverse applications that feature such dynamic protocols include parallel algorithms for scientific computing (§ 2.5) to Web services (§ 5.1), as the examples in this paper shall illustrate.

Let us first consider a simple protocol where participant Alice sends a message of type  $\text{nat}$  to participant Bob. To develop the code for this protocol, we start by specifying the global type as

$$G_1 = \text{Alice} \rightarrow \text{Bob} : \langle \text{nat} \rangle . \text{end}$$

where  $\rightarrow$  signifies the flow of communication and end denotes protocol termination. With agreement on  $G_1$  as a specification for Alice and Bob, each program can be implemented separately. Then for type-checking,  $G_1$  is *projected* into local session types: one from Alice's point of view,  $!(\text{Bob}, \text{nat})$  (output to Bob with  $\text{nat}$ -type),

## Sequence topology

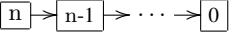
$$\begin{array}{c} \Pi n. (\mathbf{R} \text{ end} \\ \lambda i. \lambda x. W[i+1] \rightarrow W[i] : \langle \text{nat} \rangle . x \\ n) \end{array}$$


Figure 1. Simple multiparty network topology

and another from Bob's point of view,  $?(\text{Alice}, \text{nat})$  (input from Alice with  $\text{nat}$ -type), against which the respective Alice and Bob programs are checked to be correct.

The first motivation to extend the above approach is to allow modular specification of protocols, and programming for an arbitrary composition of global types. Consider the global type  $G_2 = \text{Bob} \rightarrow \text{Carol} : \langle \text{nat} \rangle . \text{end}$ . The designer may wish to compose  $G_1$  and  $G_2$  together to build a larger protocol:

$$\begin{aligned} G_3 &= \text{comp}(G_1, G_2) \\ &= \text{Alice} \rightarrow \text{Bob} : \langle \text{nat} \rangle . \text{Bob} \rightarrow \text{Carol} : \langle \text{nat} \rangle . \text{end} \end{aligned}$$

We may also need to iterate the composed protocols  $n + 1$ -times, as in  $\text{foreach}(i : [0..n]) \{ \text{comp}(G_1, G_2) \}$ . These kinds of operations and constructs for the composition and repetition of interaction units are standard requirements in e.g. multiparty contracts [30].

In order to support the type-based specification and programming of such idioms, we use the primitive recursion operator  $\mathbf{R}$  from Gödel's System  $\mathcal{T}$  [18]. The basic system of finite session types is thus extended to allow *primitive recursive formulations of session type sequences* from which dependent product types on natural numbers can be constructed. The following two reduction rules from System  $\mathcal{T}$  are added to global program specifications.

$$\begin{aligned} \mathbf{R} G \lambda i. \lambda x. G' 0 &\longrightarrow G \\ \mathbf{R} G \lambda i. \lambda x. G' n + 1 &\longrightarrow G' \{n/i\} \{ (\mathbf{R} G \lambda i. \lambda x. G' n) / x \} \end{aligned}$$

Now we can define the composition and repetition operators

$$\begin{aligned} \text{comp}(G_1, G_2) &= \mathbf{R} G_2 \lambda i. \lambda x. G_1 \{x/\text{end}\} 1 \\ \text{foreach}(i : [0..n]) \{ G \} &= \mathbf{R} \text{end} \lambda i. \lambda x. G \{x/\text{end}\} n + 1 \end{aligned}$$

where we assume that  $x$  does not occur in  $G_1$  and  $G$ , and that  $G_1$  and  $G$  terminate with end; the encodings substitute  $x$  for end. The composition operator executes  $G_1$  and  $G_2$  sequentially, while the repetition operator above repeats  $G$   $n + 1$ -times. These definitions are classic syntactic sugars in functional programming.

Further, we can bind the number of repetitions  $n$  by a dependent product type to build a *global specification procedure*, as in

$$\Pi j. \text{foreach}(i : [0..j]) \{ \text{comp}(G_1, G_2) \} \quad (1)$$

where  $\Pi j$  denotes the dependent binder.

Beyond a variable number of exchanges between a fixed number of principals, the ability to *parameterise participant identities* enables the representation of many communication topologies found in the literature. For example, the simple participant index in  $W[i]$  (the  $i$ -th worker) creates a parameterised family of workers, and we can formulate a *growing sequence of session types* as depicted in figure 1: neither the number of participants nor the total number of message exchanges are fixed before execution.

Here we face a couple of immediate questions:

- *How expressive is this simple approach?* Are we able to we program the main communication topologies found across the literature for e.g. parallel algorithms and Web services?
- *How can we check program correctness?* More concretely, how can we project a parameterised global type to a local type even when we do not know the exact shape of the topology yet.

For example, in the sequencing topology of figure 1, if  $n \geq 2$ , then there are three different *roles* inhabiting this specification: the initiator, the  $n - 2$  middle workers, and the last worker, each of which engages in a distinct communication pattern. The number of roles is also variable; when  $n = 1$ , there is only the initiator and the last worker; and when  $n = 0$ , the session does not involve any communication.

The integration of dependent types and multiparty session types enables not only the precise description of the communication topology of agents and the scale of their interaction patterns, it offers a powerful tool by which we can write down a generic generator of end-point local types and implementations that follows the specified structure and can be statically type-checked. This means that whatever concrete topologies are formed by the generated code at run-time, the participants are always guaranteed to conform to the stipulated global topology and interaction patterns. The ability to treat the typed interactions among the arbitrary number of participants as a single multiparty session also leads to clearer prospects regarding resource usage by these agents.

### Contributions of this work

- A *new expressive framework for programming global, distributed specifications*, which can elegantly and concisely describe a wide range of parametric dynamic communication topologies. This framework is based on the combination of multiparty session types and the recursion operator for dependent types originating from Gödel's System  $\mathcal{T}$ , and in which the number of participants, messages and/or repetitions can vary at run-time (§ 2).
- A projection method from a dependent global type onto a *generic end-point generator* which exactly captures the interaction structures of parameterised end-points and which can generate the class of all possible end-point types (§ 3.1).
- A *dependent typing system* that treats the full multiparty session types integrated with dependent product types with recursors. The resulting static typing system allows decidable type-checking and guarantees type-safety and deadlock-freedom for well-typed multiparty processes in dynamic communication topologies (§ 3).
- *Applications* featuring various communication topologies, including the complex butterfly network for the parallel Fast Fourier Transform algorithm (§ 2.5.5.2). We prove their typability, type-safety and deadlock-freedom. We have additionally applied our framework to public usecases for Web services [3] (§ 5.1). We also report preliminary benchmark results to demonstrate the potential benefits of using parameterised dependent types on efficiency and optimal resource usage (§ 6).

Detailed definitions and additional materials are found at [1].

## 2. Types and Processes for Multiparty Communication Topologies

### 2.1 Global types

The global types allow the description of the parameterised topologies and conversation scenarios of a multiparty session as a type signature. We start by defining their grammar. Our type syntax is based on the three different formulations: (1) the global types from [7]; (2) dependent types with primitive recursive combinators based on [26]; and (3) parameterised dependent types from a simplified Dependent ML (DML) [4, 31].

$S ::= \text{bool} \mid \text{nat} \mid \dots \mid \langle G \rangle$	Value type
$U ::= S \mid T$	Message type
$i ::= i \mid n \mid i + i' \mid i - i' \mid i * i'$	Indexes
$P ::= P \wedge P \mid i \leq i'$	Propositions
$I ::= \text{nat} \mid \{i : I \mid P\}$	Index sorts
$\mathcal{P} ::= \text{Alice} \mid \text{Bob} \mid \text{Worker} \mid \dots$	Participants
$p ::= p[i] \mid \mathcal{P}$	Principals
$G ::=$	Global types
$p \rightarrow p' : \langle U \rangle . G$	Message
$p \rightarrow p' : \{l_i : G_i\}_{i \in I}$	Branching
$\mu t . G$	Recursion
$R G \lambda i : I . \lambda x . G'$	Primitive Recursor
$t$	Recursive type variable
$x$	Recursor type variable
$\Pi i : I . G$	Dependent type
$G \ i$	Application
end	End

Figure 2. Global types

$R G \lambda i : I . \lambda x . G' \ 0$	$\rightarrow G$
$R G \lambda i : I . \lambda x . G' \ n$	$\rightarrow G' \{n - 1 / i\} \{ (R G \lambda i : I . \lambda x . G' \ n - 1) / x \}$
$(\Pi i : I . G) \ n$	$\rightarrow G \{n / i\}$

Figure 3. Type reduction

The grammar of global types  $(G, G', \dots)$  is given in figure 2. *Parameterised principals*  $p, p', q, \dots$  are participant constants (Alice, Bob, ...) that can be indexed by one or more parameters, e.g.  $\text{Worker}[5][i + 1]$ . Index  $i$  ranges over index variables  $i, j, n$ , natural numbers  $n$  or arithmetic operations. A global interaction can be a message exchange  $(p \rightarrow p' : \langle U \rangle . G)$ , where  $p, p'$  denote the sending and receiving principals,  $U$  the payload type of the message and  $G$  the subsequent interaction. Payload types  $U$  are either value types  $S$  (which contain base types  $\text{bool}, \text{nat}, \dots$  and session channel types  $\langle G \rangle$ ), or *local types*  $T$  (which correspond to the behaviour of one of the session participants and will be explained in § 3) for delegation. Branching  $(p \rightarrow p' : \{l_i : G_i\}_{i \in I})$  allows to follow the different  $G_i$  paths in the global interaction. Recursion  $(\mu t . G)$  models infinite interaction. Type variables ( $t$ ) are guarded in the standard way, i.e., type variables only appear under some prefix [28].

The interesting additions are the primitive recursion operator  $R G \lambda i : I . \lambda x . G'$  from Gödel's System  $\mathcal{T}$  [18] and the dependant global type  $\Pi i : I . G$  from [26]. Their semantics is given in figure 3. The primitive recursive operator takes as parameters a global type  $G$ , an index variable  $i$  with range  $I$ , a type variable for recursion  $x$  and a recursion body  $G'$ . When applied to an index  $i$ , its semantics corresponds to the repetition  $i$  times of the body  $G'$ , with the index variable  $i$  value going down one at each iteration, from  $n - 1$  to 0. The final behaviour is given by  $G$  when the index reaches 0. The dependent type  $\Pi i : I . G$  also has a standard semantics: when applied to an index  $i$ , the index is simply substituted. We assume a call-by value semantics so that the argument is evaluated first. The index sorts comprise the natural numbers and subset of index sorts, which are permitted only with respect of a restricted set of predicates  $(P, P', \dots)$ . In our case, these are conjunctions of inequalities. We often omit  $I$  and end.

### 2.2 Examples of multiparty communication topologies

We present some programming examples of global types that specify typical network topologies found in classical parallel algorithms textbooks [23].

**Ring topology - figure 4(a)** The ring topology features  $n + 1$  workers (named by  $W$ ) that each have exactly two neighbours: the worker  $W[i]$  communicates with the worker  $W[i - 1]$  and  $W[i + 1]$  ( $1 \leq i \leq n - 1$ ), with the exception of  $W[n]$  and  $W[0]$  who share

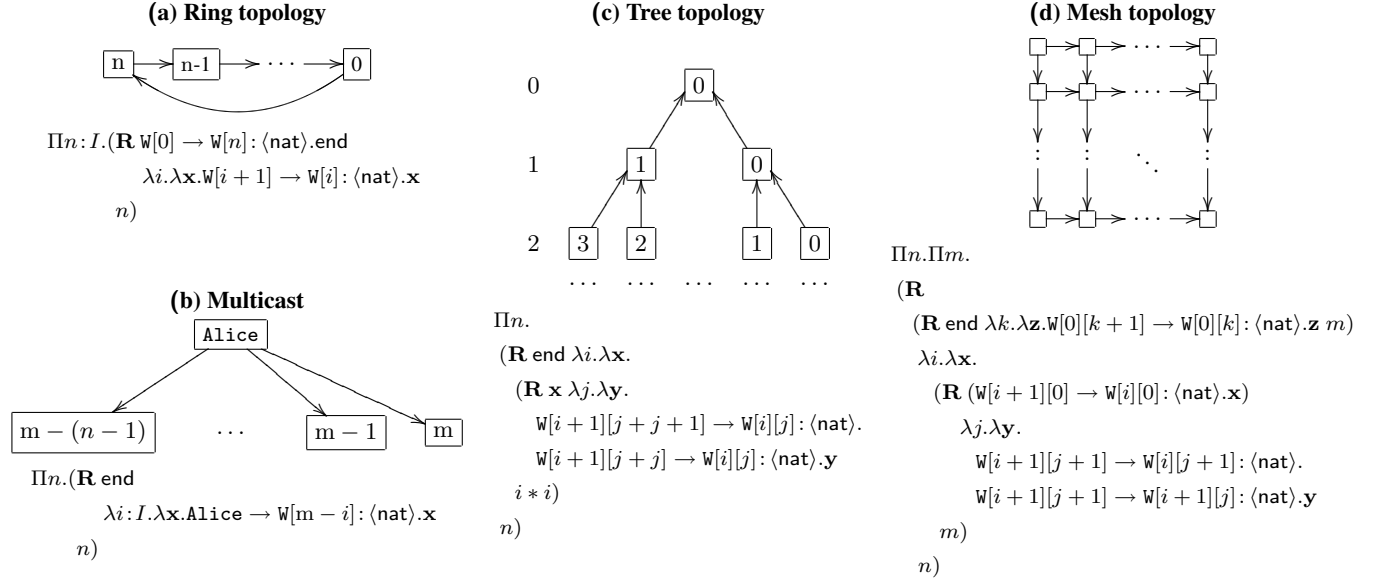


Figure 4. Multiparty network topologies

a direct link. This global type example is a slight variation from the sequence presented in figure 1. The type specifies that the first message is sent by  $W[n]$  to  $W[n-1]$ , the last being a message from  $W[0]$  back to  $W[n]$ . To ensure no self-ring between  $W[0]$  to  $W[0]$ , we set  $I = \{n : n \geq 1\}$ .

**Multicast - figure 4(b)** The multicast session consists of Alice sending a message to  $n$  workers  $W$ . The first message is thus sent from Alice to  $W[m - (n - 1)]$ , then to  $W[m - (n - 2)]$ , until  $W[m]$ . To ensure the indices are naturals, we set  $I = \{i : m - i \geq 0\}$ .

**Tree topology - figure 4(c)** The session from figure 4(c) represents the propagation of values over a network of  $2^n - 1$  workers organised in a binary tree topology. The workers are indexed by two parameters: the first parameter represents the depth, while the second reflects its width position. The global type specifies messages converging from the leaves towards the root  $W[0][0]$ .

**Mesh topology - figure 4(d)** The session from figure 4(c) describes communication over a mesh topology. In our two dimensional example, each worker has four neighbours, except for the ones located on the first and last rows and columns. Our session takes two parameters  $n$  and  $m$  which represent the number of rows and the number of columns. Then we have two iterators that repeat  $W[i+1][j+1] \rightarrow W[i][j+1] : \langle \text{nat} \rangle$  and  $W[i+1][j+1] \rightarrow W[i+1][j] : \langle \text{nat} \rangle$  for all  $i$  and  $j$ . These two messages specify that each worker not situated on the last row or last column sends a message to his neighbours situated below and on its right. The types  $W[i+1][0] \rightarrow W[i][0] : \langle \text{nat} \rangle$  and  $\mathbf{R} \, \text{end} \, \lambda k. \lambda z. W[0][k+1] \rightarrow W[0][k] : \langle \text{nat} \rangle. z \, m$  deal with, respectively, the last column and the last row. As in the tree example, the messages converge towards  $W[0][0]$ . Variants of this topology include toric meshes and hypercubes.

### 2.3 Syntax and semantics

**Syntax** The syntax of expressions and processes is given in figure 5, extended from [7]. Identifiers  $u$  can be variables  $x$  or channel names  $a$ . Values  $v$  are either channels  $a$ , natural number  $n$  or boolean  $\text{true}$ ,  $\text{false}$  constants. Expressions  $e$  are built out of indices  $i$ , values  $v$ , variables  $x$  and operations over expressions ( $e = e'$ ,  $e$  and  $e'$ , not  $e, \dots$ ). In processes, session are initiated by a multicast request  $\bar{u}[p_0..p_n](y).P$  that is accepted by the participants through  $u[p](y).P$ . Messages are sent by  $c!\langle p, e \rangle; P$  to the

$u ::= x \mid a$	Identifiers
$v ::= a \mid n \mid \text{true} \mid \text{false}$	Values
$e ::= i \mid v \mid x \mid e \, \text{op} \, e'$	Expressions
$P ::=$	Processes
$\bar{u}[p_0..p_n](y).P$	Multicast Request
$u[p](y).P$	Accept
$c!\langle p, e \rangle; P$	Value sending
$c?\langle p, x \rangle; P$	Value reception
$c!\langle\langle p, c' \rangle\rangle; P$	Session delegation
$c?\langle\langle p, y \rangle\rangle; P$	Session reception
$c \oplus \langle p, l \rangle; P$	Selection
$c \& \langle p, \{l_i : P_i\}_{i \in I} \rangle$	Branching
$(\nu a)P$	Hiding
$\text{def } \mathbb{X}(x, y) = P \text{ in } P$	Recursive definition
$\mathbb{X}(e, y)$	Process call
$0$	Inaction
$P \mid Q$	Parallel
$\text{if } e \text{ then } P \text{ else } Q$	Conditional
$\mathbf{R} \, P \, \lambda i. \lambda X. Q$	Primitive Recursion
$X$	Process Variable
$\lambda i. P$	Abstraction
$(P \, i)$	Application
$(\nu s)P$	Session restriction
$s : h$	Queues
$c ::= y \mid s[p]$	Channels
$\hat{p}, \hat{q} ::= \hat{p}[n] \mid \mathcal{P}$	Principal values
$m ::= (\hat{q}, \hat{p}, v) \mid (\hat{q}, \hat{p}, s[\hat{p}']) \mid (\hat{q}, \hat{p}, l)$	Messages in transit
$h ::= \epsilon \mid m \cdot h$	Queues

Figure 5. Syntax for user-defined and run-time processes

participant  $p$  and received by  $c?\langle p, x \rangle; P$  from the participant  $q$ . Delegation,  $c!\langle\langle p, c' \rangle\rangle; P$ , allows the end point of a running session to be sent to  $p$ . Delegated sessions are received by  $q$  using  $c?\langle\langle p, y \rangle\rangle; P$ . Selection,  $c?\langle\langle p, y \rangle\rangle; P$ , and branching,  $c \& \langle q, \{l_i : P_i\}_{i \in I} \rangle$ , allow a participant to choose a branch from those supported by another. Standard language constructs include restriction  $(\nu a)P$ , the local definition of recursive processes  $\text{def } \mathbb{X}(x, y) = P \text{ in } P$  and their use  $\mathbb{X}(e, y)$ , parallel composition  $P \mid Q$ , conditionals  $\text{if } e \text{ then } P \text{ else } Q$  and the null process  $0$ . The primitive recursion operator  $\mathbf{R} \, P \, \lambda i. \lambda X. Q$  takes as parameters a process  $P$ , a function

$(\lambda i.P) \mathbf{n} \longrightarrow P\{\mathbf{n}/i\}$	[Beta]
$\mathbf{R} P \lambda i. \lambda X. Q \mathbf{0} \longrightarrow P \quad \mathbf{R} P \lambda i. \lambda X. Q \mathbf{n} + 1 \longrightarrow P\{\mathbf{n}/i\} \{ \mathbf{R} P \lambda i. \lambda X. Q \mathbf{n}/X \}$	[ZeroR, SuccR]
$\text{if true then } P \text{ else } Q \longrightarrow P \quad \text{if false then } P \text{ else } Q \longrightarrow Q$	[If-T, If-F]
$\bar{a}[\hat{p}_0.. \hat{p}_n](y_0).P_0 \mid a[\hat{p}_1](y_1).P_1 \mid \dots a[\hat{p}_n](y_n).P_n \longrightarrow (\nu s)(P_0\{s[\hat{p}_0]/y_0\} \mid \dots \mid P_n\{s[\hat{p}_n]/y_n\} \mid s : \emptyset)$	[Link]
$s[\hat{p}]!\langle \hat{q}, v \rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{p}, \hat{q}, v)$	[Send]
$s[\hat{p}]!\langle \langle \hat{q}, s'[\hat{p}'] \rangle \rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{p}, \hat{q}, s'[\hat{p}'])$	[Deleg]
$s[\hat{p}] \oplus \langle \hat{q}, l \rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{p}, \hat{q}, l)$	[Label]
$s[\hat{p}]?(\hat{q}, x); P \mid s : (\hat{q}, \hat{p}, v) \cdot h \longrightarrow P\{v/x\} \mid s : h$	[Recv]
$s[\hat{p}]?(\langle \hat{q}, y \rangle); P \mid s : (\hat{q}, \hat{p}, s'[\hat{p}']) \cdot h \longrightarrow P\{s'[\hat{p}']/y\} \mid s : h$	[Srec]
$s[\hat{p}] \& \langle \hat{q}, \{l_i : P_i\}_{i \in I} \rangle \mid s : (\hat{q}, \hat{p}, l_{i_0}) \cdot h \longrightarrow P_{i_0} \mid s : h \quad (i_0 \in I)$	[Branch]
$\text{def } \mathbb{X}(x, y) = P \text{ in } (\mathbb{X}(v, c) \mid Q) \longrightarrow \text{def } \mathbb{X}(x, y) = P \text{ in } (P\{v/x\}\{c/y\} \mid Q)$	[Def]
$P \longrightarrow P' \Rightarrow P e \longrightarrow P' e \quad P \longrightarrow P' \Rightarrow (\nu r)P \longrightarrow (\nu r)P' \quad P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q$	[App, Scop, Par]
$P \longrightarrow P' \Rightarrow \text{def } D \text{ in } P \longrightarrow \text{def } D \text{ in } P' \quad P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q \equiv Q' \Rightarrow P \longrightarrow Q$	[Defin, Str]
$e_0 \longrightarrow e'_0 \Rightarrow \mathcal{E}[e_0, \dots, e_i] \longrightarrow \mathcal{E}[e'_0, \dots, e_i]$	[Context]

Figure 6. Reduction rules

taking an index parameter  $i$  and a recursion variable  $X$ . Note the distinction between  $\mathbb{X}\langle e, y \rangle$  (infinite repeat) and  $X$  (finite repeat). We often omit  $\mathbf{0}$  and the participant  $\mathbf{p}$  from the session primitives. Session hiding and the FIFO queue appear only at runtime, as explained below.

**Semantics** The semantics is defined by the reduction relation  $\longrightarrow$  presented in figure 6. Some of the run-time syntax is detailed in figure 5. The metavariables  $\hat{p}, \hat{q}, \dots$  range over principal values which are principals where all indices have been evaluated to integers (such as  $\mathbb{W}[3][5]$ ). The reduction rules feature  $\beta$ -reduction and recursors. Note that our  $\beta$ -reduction does not take higher-order values but *only numerals*. We omit integer and boolean operations.

The most important rule is [Link], which describes the initiation of a new session among  $\mathbf{n}$  participants that synchronise over a service name  $a$ . The first participant  $\bar{a}[\mathbf{p}_0.. \mathbf{p}_n](y_0).P_0$  is distinguished by the overbar on the service name. His role is to specify the set of participants, which would be dynamically determined during execution. We use contexts (whose definition we omit) to ensure the evaluation of indices before session reductions (such as  $a[\dots](y).P$ ) and the evaluation of expressions (e.g.  $\dots \text{op } \dots$ ). For example,  $\bar{a}[\mathbb{W}[0], \mathbb{W}[3+1], \mathbb{W}[2+2]](y_0).P_0$  becomes  $\bar{a}[\mathbb{W}[0], \mathbb{W}[4]](y_0).P_0$  before [Link] is applied. This facility, in combination with the [Beta] reduction, makes it possible to determine the participant number of a multiparty session at runtime (see e.g. the sequence example of figure 1). After the connection, the participants will share the private session name  $s$ , and the queue associated to  $s$ , which is initialised as empty. The variables  $y_p$  in each participant  $\mathbf{p}$  will then be replaced with the corresponding channel with its name,  $s[\mathbf{p}]$ .

The rest of the session reductions are standard [7, 21]. The output rules [Send], [Deleg] and [Label] push values, channels and labels, respectively, into the queue of the session  $s$ . The rules [Recv], [Srec] and [Branch] perform the complementary operations. Note that these operations check that the sender and receiver match. Processes are considered modulo structural equivalence, denoted by  $\equiv$ , whose definition we omit.

## 2.4 Processes for multiparty communication topologies

We give the process representation for the communication topologies from the Introduction (§ 1) and § 2.2. There are various ways to

implement end-point processes from a single global type. We show one instance for each topology below.

**Repetition** A concrete definition for the protocol (1) in § 1 is:

$\Pi n. (\mathbf{R} \text{ end } \lambda i. \lambda x. \text{Alice} \rightarrow \text{Bob} : \langle \text{nat} \rangle. \text{Bob} \rightarrow \text{Carol} : \langle \text{nat} \rangle. x \mathbf{n})$

Then Alice and Bob can be implemented with recursors as follows (we abbreviate Alice by  $\mathbf{a}$ , Bob by  $\mathbf{b}$  and Carol by  $\mathbf{c}$ ).

$\text{Alice}(n) = \bar{a}[\mathbf{a}, \mathbf{b}, \mathbf{c}](y). (\mathbf{R} \mathbf{0} \lambda i. \lambda X. y! \langle \mathbf{b}, e[i] \rangle; X \mathbf{n})$

$\text{Bob}(n) = a[\mathbf{b}](y). (\mathbf{R} \mathbf{0} \lambda i. \lambda X. y? \langle \mathbf{a}, z \rangle; y! \langle \mathbf{c}, z \rangle; X \mathbf{n})$

We omit Carol, which is also implemented using a recursor. Alice repeatedly sends a message  $e[i]$  to Bob  $n$ -times. Then  $n$  can be bound by  $\lambda$ -abstraction, allowing the user to dynamically assign the number of the repetitions.

$\lambda n. ((\nu a)(\text{Alice}(n) \mid \text{Bob}(n) \mid \text{Carol}(n))) \mathbf{1000}$

**Ring topology - figure 4(a)** This topology features three distinct participant roles: the worker  $\mathbb{W}[n]$  is the starter of the multiparty interaction, who sends a value and eventually receives a value from the last worker  $\mathbb{W}[0]$ . The rest of the workers first receive a value and then send a value to the next worker. We give a process that generates all the participants using a recursor.

$\mathbf{R} \quad \bar{a}[\mathbb{W}[n].. \mathbb{W}[0]](y). y! \langle \mathbb{W}[n-1], v \rangle; y? \langle \mathbb{W}[0], z \rangle; P$

$a[\mathbb{W}[0]](y). y? \langle \mathbb{W}[1], z \rangle; y! \langle \mathbb{W}[n], z \rangle; Q$

$\lambda i. \lambda X. (a[\mathbb{W}[i]](y). y? \langle \mathbb{W}[i-1], z \rangle; y! \langle \mathbb{W}[i+1], z \rangle; \mid X) \mathbf{n}$

Note that  $n \geq 1$  is ensured by typing.

**Tree topology - figure 4(c)** There are the three roles represented in this protocol – the root, the nodes and the leaves. We give a process for each of them:

$P_{\text{root}} = \bar{a}[\mathbb{W}[0][0].. \mathbb{W}[n][n]](y). y? \langle x_1 \rangle; y? \langle x_2 \rangle; P$

$P_{\text{node}[i][j]} = a[\mathbb{W}[i][j]](y). y? \langle x_1 \rangle; y? \langle x_2 \rangle; y! \langle x_1 + x_2 \rangle; \mathbf{0}$

$P_{\text{leaf}[j]} = a[\mathbb{W}[n][j]](y). y! \langle z_j \rangle; \mathbf{0}$

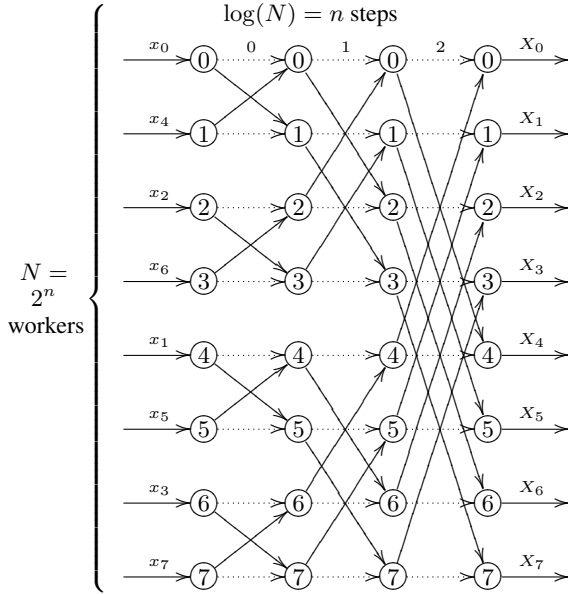
where we omit the participant annotations from sending and receiving. Here the processes compute the sum of a number of values that are known to the leaves. A process  $P$  is left abstract to represent the root's actions using this sum. We can generate all the processes using a recursor in a similar way to the previous example.

**Mesh - figure 4(d)** The mesh example is more complex: when  $n$  and  $m$  are bigger than 2, there are 9 distinct roles that each have

**(a) Butterfly pattern**

$$\begin{array}{lcl} x_{k-N/2} & \xrightarrow{\quad} & X_{k-N/2} = x_{k-N/2} + x_k * \omega_N^{k-N/2} \\ x_k & \xrightarrow{\quad} & X_k = x_{k-N/2} + x_k * \omega_N^k \end{array}$$

**(b) FFT diagram**



**(c) Global type**

$$G = \Pi n.$$

$$(\mathbf{R} \text{ end } \lambda l. \lambda x.$$

$$(\mathbf{R} \ x \ \lambda i. \lambda y.$$

$$(\mathbf{R} \ y \ \lambda j. \lambda z.$$

$$i * 2^{n-l} + 2^{n-l-1} + j \rightarrow i * 2^{n-l} + j : \langle \text{nat} \rangle.$$

$$i * 2^{n-l} + j \rightarrow i * 2^{n-l} + 2^{n-l-1} + j : \langle \text{nat} \rangle.$$

$$i * 2^{n-l} + 2^{n-l-1} + j \rightarrow i * 2^{n-l} + 2^{n-l-1} + j : \langle \text{nat} \rangle.$$

$$i * 2^{n-l} + j \rightarrow i * 2^{n-l} + j : \langle \text{nat} \rangle. z$$

$$) \ 2^{n-l-1}$$

$$) \ 2^l$$

$$) \ n)$$

$$\lambda k. \lambda u. k \rightarrow k : \langle \text{nat} \rangle. u)$$

$$2^n$$

**(d) Processes**  $P(n, p, x_{\overline{p}}, y) =$

$$y! \langle p, x_{\overline{p}} \rangle;$$

$$(\mathbf{R} \ y? \langle p, x \rangle; \mathbf{0} \ \lambda l. \lambda X.$$

$$\text{if } \text{bit}_{n-l}(p) = 0$$

$$\text{then } y? \langle p, x \rangle; y! \langle p + 2^{n-l-1}, x \rangle; y? \langle p + 2^{n-l-1}, z \rangle; y! \langle p, x + z \omega_N^{g(l,p)} \rangle; X$$

$$\text{else } y? \langle p, x \rangle; y? \langle p - 2^{n-l-1}, z \rangle; y! \langle p - 2^{n-l-1}, x \rangle; y! \langle p, z + x \omega_N^{g(l,p)} \rangle; X)$$

$$n$$

$$\text{where } g(l, p) = p \bmod 2^l$$

**Figure 7.** Fast Fourier Transform on a butterfly network topology

a different pattern of communication. We only list (1) the centre workers  $\mathbb{W}[i][j]$  ( $1 \leq i < n$ ,  $1 \leq j < m$ ) who are connected in all four directions, and (2) the initiator  $\mathbb{W}[n][m]$  from the top-left corner. Below  $f(i, j)$  represents the expression computed at the  $(i, j)$ -th element.

$$P_{\text{centre}}(i, j) = a[\mathbb{W}[i][j]](y).y?(\mathbb{W}[i+1][j], z_1); y?(\mathbb{W}[i][j+1], z_2);$$

$$y! \langle \mathbb{W}[i-1][j], f(i-1, j) \rangle;$$

$$y! \langle \mathbb{W}[i][j-1], f(i, j-1) \rangle; \mathbf{0}$$

$$P_{\text{start}}(n, m) = a[\mathbb{W}[0][0].. \mathbb{W}[n][m]](y).y! \langle \mathbb{W}[n-1][m], f(n-1, m) \rangle;$$

$$y! \langle \mathbb{W}[n][m-1], f(n, m-1) \rangle; \mathbf{0}$$

## 2.5 The butterfly network for Fast Fourier Transformation

This subsection attests the expressiveness of our global specification language by describing the Fast Fourier Transform (FFT) algorithm over a butterfly network. The FFT is one of the most widely used algorithms in computer science and engineering. Its application range from signal processing to big integer multiplication. It is also easily parallelisable and fast in the parallel domain.

We start by a quick reminder of the discrete fourier transform definition, followed by the description of an FFT algorithm that implements it over a butterfly network. We then give the corresponding global session type. From the diagram in (b) and the session type from (c), it is finally straightforward to implement the FFT as simple interacting processes.

**The Discrete Fourier Transform** The goal of the FFT is to compute the Discrete Fourier Transform (DFT) of a vector of complex numbers. Assume the input consists in  $N$  complex numbers  $\vec{x} = x_0, \dots, x_{N-1}$  that can be interpreted as the coefficients of a polynomial  $f(y) = \sum_{j=0}^{N-1} x_j y^j$ . The DFT transforms  $\vec{x}$  in a vector

$\vec{X} = X_0, \dots, X_{N-1}$  defined by

$$X_k = f(\omega_N^k)$$

with  $\omega_N^k = e^{i \frac{2k\pi}{N}}$  a primitive root of unity. The DFT can be seen as a polynomial interpolation on the primitive roots of unity or as the application of the square matrix  $(\omega_N^{ij})_{i,j}$  to the vector  $\vec{x}$ .

**FFT and the butterfly network** We present here the radix-2 variant of the Cooley-Tukey FFT algorithm [16]. Assuming that  $N$  is a power of 2, this FFT algorithm uses a divide-and-conquer strategy based on the following equation (we use the fact that  $\omega_N^{2k} = \omega_{N/2}^k$ ):

$$\begin{aligned} X_k &= \sum_{j=0}^{N/2-1} x_j \omega_N^{jk} \\ &= \sum_{j=0}^{N/2-1} x_{2j} \omega_{N/2}^{jk} + \omega_N^k \sum_{j=0}^{N/2-1} x_{2j+1} \omega_{N/2}^{jk} \end{aligned}$$

Each of the two separate sums are DFT of half of the original vector members, separated into even and odd. Recursive calls can then divide the input set further based on the value of the next binary bits. The good complexity of this FFT algorithm comes from the lower periodicity of  $\omega_{N/2}$ : we have  $\omega_{N/2}^{jk} = \omega_{N/2}^{j(k-N/2)}$  and thus computations of  $X_k$  and  $X_{k-N/2}$  only differ by the multiplicative factor affecting one of the two recursive calls. Figure 7(a) illustrates this recursive principle, called *butterfly*, where two different outputs can be computed in constant time from the results of the same two recursive calls.

The complete algorithm is illustrated by the diagram from figure 7(b). It features the application of the FFT on a network of  $N = 2^3 = 8$  machines computing the DFT of vector  $x_0, \dots, x_7$ . Each row represents a single machine at each step of the algorithm. Each edge represents a value sent to another machine. The dotted edges represent the particular messages that a machine sends to itself to remember a value for the next step. When reading the diagram

$T ::= !\langle p, U \rangle; T$	Output	$\mathbf{R} \ T \ \lambda i : I. \lambda x. T'$	Prim. Rec.
$ \ ?\langle p, U \rangle; T$	Input	$\mathbf{x}$	Type Var.
$ \ \oplus \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Select.	$\Pi i : I. T$	Dep. Type
$ \ \& \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Branch.	$T \ i$	Application
$ \ \mu t. T$	Rec., Var.	end	Null

Figure 8. Local types

from right to left, each step consists in merging the results from half of the inputs, following in this the butterfly pattern: each machine is successively involved in a butterfly with a machine whose number differs by only one bit. Note that the recursive partition over the value of a different bit at each step requires a particular bit-reversed ordering of the input vector: the machine number  $p$  initially receives  $x_{\bar{p}}$  where  $\bar{p}$  denotes the bit-reversal of  $p$ .

This parallel version of the FFT algorithm gives an excellent  $O(N)$  speedup on a butterfly network of  $N$  machines when applied on a vector of size  $N$ . It has also the advantage of being able to compute the DFT inverse by just changing the multiplication factors of each butterfly. Finally, this algorithm can be implemented easily on common network topologies such as the hypercube.

**Global Types** Figure 7(c) gives the global session type corresponding to the execution of the FFT. The size of the network is specified by the index parameter  $n$ : for a given  $n$ ,  $2^n$  machines compute the DFT of a vector of size  $2^n$ . The first iterator  $\mathbf{R} \ (\dots) \ \lambda k. \lambda u. k \rightarrow k : \langle \text{nat} \rangle. u$  concerns the initialisation: each of the machines sends the  $x_p$  value to themselves. Then we have an iteration over variable  $l$  for the  $n$  successive steps of the algorithm. The iterators over variables  $i, j$  work in a more complex way: at each step, the algorithm applies the butterfly pattern between pairs of machines whose numbers differ by only one bit (at step  $l$ , bit number  $n - l$  is concerned). Iterators over variables  $i$  and  $j$  thus generate all the values of the other bits: for each  $l$ ,  $i * 2^{n-l} + j$  and  $i * 2^{n-l} + 2^{n-l-1} + j$  range over all pairs of integers from  $2^n - 1$  to 0 that differ on the  $(n - l)$ th bit. The four repeated messages within the loops then correspond exactly to the four edges of the butterfly pattern.

**Processes** The processes that are run on each machine to execute the FFT algorithm are presented in figure 7(d). When  $p$  is the machine number,  $x_{\bar{p}}$  the initial value, and  $y$  the session channel, the machine starts by sending  $x_{\bar{p}}$  to itself:  $y!(x_{\bar{p}});$ . The main loop corresponds to the iteration over the  $n$  steps of the algorithm. At step  $l$ , each machine is involved in a butterfly corresponding to bit number  $n - l$ , i.e. whose number differs on the  $(n - l)$ th bit. In the process, we thus distinguish the two cases corresponding to each value of the  $(n - l)$ th bit (test on  $\text{bit}_{n-l}(p)$ ). In the two branches, we receive the previously computed value  $y?(x); \dots$ , then we send to and receive from the other machine (of number  $p + 2^{n-l-1}$  or  $p - 2^{n-l-1}$ , i.e. whose  $(n - l)$ th bit was flipped). We finally compute the new value and send it to ourselves: respectively by  $y!(x + z \omega_N^{g(l,p)}); X$  or  $y!(z + x \omega_N^{g(l,p)}); X$ . Note that the two branches do not present the same order of send and receive as the global session type specifies that the diagonal up arrow of the butterfly comes first.

We show our static type-checking can guarantee communication-safety and deadlock-freedom for a whole group of  $N$ -processes over this complex butterfly topology.

### 3. Typing Multiparty Communication Topologies

This section introduces the type system, by which we can statically type dynamically changing communication topologies.

#### 3.1 Local types and end-point projections

The first key challenge is to define a projection from a global type to the local types of individual participants, whose identities and numbers might only be known at runtime. The syntax of local types

$$\begin{aligned}
 p \rightarrow p' : \langle U \rangle. G \upharpoonright q &= \text{if } q=p=p' \text{ then } !\langle p, U \rangle; ?\langle p, U \rangle; G \upharpoonright q \\
 &\quad \text{else if } q=p \text{ then } !\langle p', U \rangle; G \upharpoonright q \\
 &\quad \text{else if } q=p' \text{ then } ?\langle p, U \rangle; G \upharpoonright q \\
 &\quad \text{else } G \upharpoonright q \\
 p \rightarrow p' : \{l_i : G_i\}_{i \in I} \upharpoonright q &= \text{if } q=p \text{ then } \oplus \langle p', \{l_i : G_i \upharpoonright q\}_{i \in I} \rangle \\
 &\quad \text{else if } q=p' \text{ then } \& \langle p, \{l_i : G_i \upharpoonright q\}_{i \in I} \rangle \\
 &\quad \text{else } \sqcup_{i \in I} G_i \upharpoonright q \\
 (\mathbf{R} \ G \ \lambda i : I. \lambda x. G') \upharpoonright q &= \mathbf{R} \ (G \upharpoonright q) \ \lambda i : I. \lambda x. (G' \upharpoonright q) \\
 (\mu t. G) \upharpoonright p &= \mu t. G \upharpoonright p \\
 t \upharpoonright p &= t \\
 x \upharpoonright p &= x \\
 (\Pi i : I. G) \upharpoonright p &= \Pi i : I. G \upharpoonright p \\
 (G \ i) \upharpoonright p &= (G \upharpoonright p) \ i \\
 \text{end} \upharpoonright p &= \text{end}
 \end{aligned}$$

Figure 9. Projection of global types to local types

is given in figure 8. Output expresses the sending to  $p$  of a value or of a channel of type  $U$ , followed by the interactions described in  $T$ . Selection represents the transmission to  $p$  of a label  $l_i$  chosen in  $\{l_i\}_{i \in I}$  followed by  $T_i$ . Input and Branching are their dual. The other types are similar to their global types counterparts. We use the following macro encoded by a recursor.

$$(\text{if } i \text{ then } T_1 \text{ else } T_2) = \mathbf{R} \ T_1 \ \lambda i. \lambda x. T_2 \ i$$

where  $i$  and  $x$  are not free in  $T_2$ . true is encoded by 0 and false by any other integer.

**End-point projection: a generic projection** The relation between local and global types is formalised by the projection relation: a global type can be projected to a local type according to each participant's viewpoint. Since the actual participant characteristics might only be determined at runtime, we cannot straightforwardly use the definition from [7, 21]. Instead, we use the power of dependent types: a generic end-point projection of  $G$  onto  $q$ , written  $G \upharpoonright q$ , represents the family of all the possible local types that a principal  $q$  can satisfy at run-time. Intuitively, we wish to ensure:

if  $T$  is semantically equivalent up to the generic end-point generator, and if  $P$  conforms to  $T$ , then  $P$  is type-safe.

where semantically equivalent might be the  $\beta$ -equality or some form of contextual congruence (the equivalence relation is defined formally in § 3.2).

The general endpoint generator is defined in figure 9 using  $\text{if } \_ \text{ then } \_ \text{ else } \_$  (defined above). The projection  $p \rightarrow p' : \langle U \rangle. G \upharpoonright q$  leads to a case analysis: if the participant  $q$  is equal to  $p$ , then the local type of  $q$  is an output of type  $U$  to  $p'$ ; if participant  $q$  is  $p'$  then  $q$  inputs  $U$  from  $p'$ ; else we skip the prefix. The fourth case corresponds to the possibility for the sender and receiver to be identical. Projecting the branching global type is similarly defined, but for the operator  $\sqcup$  explained below.

**Mergeability and injection of branching types** We first recall the example from [21], which explains that naïve branching projection leads to inconsistent end-point types. The usual projection of the following global type is *undefined*.

$$\begin{aligned}
 W[0] \rightarrow W[1] : \{ \text{ok} : W[1] \rightarrow W[2] : \langle \text{bool} \rangle, \\
 \text{quit} : W[1] \rightarrow W[2] : \langle \text{nat} \rangle \}
 \end{aligned}$$

When we project this type onto  $W[2]$ , regardless of the choice made by  $W[0]$ , both branches have to behave in the same way, as  $W[2]$  is not aware of the chosen branch. If we change the above  $\text{nat}$  to  $\text{bool}$ , the projection of  $W[2]$  would then be defined as  $?\langle W[2], \text{bool} \rangle; \text{end}$ . This illustrates the fact that the projection of all branches have to be identical except for the principals involved (which do not appear in local types).

In our framework this restriction is too strong since each branch may contain different dynamic interaction patterns. To solve this

problem, we propose two methods called *mergeability* and *injection* of branching types. Formally, the mergeability operator  $\bowtie$  is the smallest congruence relation over local types such that:<sup>1</sup>

$$\frac{\forall i \in (I \cap J). T_i \bowtie T'_i \quad \forall i \in (I \setminus J) \cup (J \setminus I). l_i \neq l_j}{\&\langle p, \{l_i : T_i\}_{i \in I} \rangle \bowtie \&\langle p, \{l_j : T'_j\}_{j \in J} \rangle}$$

When  $T_1 \bowtie T_2$  is defined, we define the injection  $\sqcup$  as a partial commutative operator over two local types such that:

$$\begin{aligned} \&\langle p, \{l_i : T_i\}_{i \in I} \rangle \sqcup \&\langle p, \{l_j : T'_j\}_{j \in J} \rangle &= \\ \&\langle p, \{l_i : T_i \sqcup T'_i\}_{i \in I \cap J} \cup \{l_i : T_i\}_{i \in I \setminus J} \cup \{l_j : T'_j\}_{j \in J \setminus I} \rangle \end{aligned}$$

and monomorphic for other types ( $T \sqcup T = T$ ).

The mergeability relation states that two local types are identical up to their branching types where branches with distinct labels are allowed to be different. By this extended condition, if we modify our previous global type example to add ok and quit labels to notify  $W[2]$ , we get:

$$\begin{aligned} W[0] &\rightarrow W[1] : \{ok : W[1] \rightarrow W[2] : \{ok : W[1] \rightarrow W[2] \langle \text{bool} \rangle\}, \\ \text{quit} : W[1] \rightarrow W[2] : \{\text{quit} : W[1] \rightarrow W[2] \langle \text{nat} \rangle\}\} \end{aligned}$$

Then  $W[2]$  can have the local type  $\&\langle W[1], \{ok : \langle W[1], \text{bool} \rangle, \text{quit} : \langle W[1], \text{nat} \rangle\} \rangle$ . This local type cannot be generated by the original projection rule in [7, 21]. This projection is sound up to the branching subtyping (cf. Lemma 4.2).

### 3.2 Type system

This subsection introduces the type system. Because free indices appear both in terms (e.g. participants in session initialisation) and in types, the formal definition of what constitutes a valid term and a valid type are interdependent and both in turn require a careful definition of a valid global type.

**Judgements and environments** One of the main differences with previous session type systems is that session environments  $\Delta$  can now be applied to indices and can contain dependent *process types*. The grammar of environments and process types are given below.

$$\begin{aligned} \Delta &::= \emptyset \mid \Delta, c:T \\ \Gamma &::= \emptyset \mid \Gamma, u : S \mid \Gamma, \mathbb{X} : S \mid \Gamma, i : I \mid \Gamma, p \mid \Gamma, X : \tau \\ \tau &::= \Delta \mid \Pi i : I. \tau \mid \tau \text{ i} \quad \kappa ::= \text{Type} \mid \Pi i : I. \kappa \end{aligned}$$

$\Delta$  is the *session environments* which associates channels to session types.  $\Gamma$  is the *standard environment* which associates variables to sort types, service names to global types, process variables to pairs of sort types and session types, indices to index sets, predicates to index variables and term variables to session types.  $\tau$  is the *process type* which is either session environment, dependent type or application with indices.  $\kappa$  denotes *kinding* which includes the kind of proper types or the kind of type families. We write  $\Gamma, u : S$  only if  $u \notin \text{dom}(\Gamma)$  where  $\text{dom}(\Gamma)$  denotes the domain of  $\Gamma$ . We use the same convention for other variables.

Following [31], in the typing rules, we assume given two semantically defined judgements:

$$\begin{aligned} \Gamma \models P &\quad \text{predicate } P \text{ is a consequence of } \Gamma \\ \Gamma \models i : I &\quad i : I \text{ follows from the assumptions of } \Gamma \end{aligned}$$

Our type system uses the judgements listed in figure 10. We write  $\Gamma \vdash J$  for arbitrary judgements and write  $\Gamma \vdash J, J'$  to stand for both  $\Gamma \vdash J$  and  $\Gamma \vdash J'$ . In addition, we use two additional judgements for the runtime systems (one for queues and one for runtime processes) which are identical with those in [7]; they are not the main focus of this paper, hence are omitted. The full rules are listed in [1].

**Kindings** As usual, a term  $i$  in the index syntax is well-formed in  $\Gamma$  if the set of free index variables in  $i$  is a subset of the set of the index variables in  $\Gamma$ . For contexts, we assume that no variable is

<sup>1</sup> The idea of mergeability is introduced informally in the tutorial paper [13]; this paper states the formal properties and proofs.

$\Gamma \vdash \text{Env}$	well-formed environments
$\Gamma \vdash \kappa$	well-formed kindings
$\Gamma \vdash \alpha \triangleright \kappa$	well-formed types
$\Gamma \vdash \alpha \equiv \beta$	type equivalence
$\Gamma \vdash \alpha \approx \beta$	type isomorphism
$\Gamma \vdash e \triangleright U$	expression
$\Gamma \vdash p \triangleright U_p$	participant with $U_p ::= \text{nat} \mid \Pi i : I. U_p$
$\Gamma \vdash P \triangleright \tau$	processes

**Figure 10.** Judgements ( $\alpha, \beta, \dots$  range over any local or global type) bound more than once and that the free index variables appearing in  $i : I$  and  $P$  are declared earlier in the context. In figure 11, we only list kinding rules for global types. Other rules are similarly defined including those for process types (noting  $\Delta$  is a well-formed environment if it only contains types  $T$  of kind Type).

$$\begin{aligned} &\frac{\Gamma \vdash p \triangleright \text{nat}, p' \triangleright \text{nat} \quad \Gamma \vdash G' \triangleright \text{Type} \quad \Gamma \vdash U \triangleright \text{Type} \quad \text{ftv}(U) = \emptyset}{\Gamma \vdash p \rightarrow p' : \langle U \rangle. G' \triangleright \text{Type}} \text{ [KIO]} \\ &\frac{\Gamma \vdash p \triangleright \text{nat}, p' \triangleright \text{nat} \quad \forall i \in I, \Gamma \vdash G_i \triangleright \text{Type}}{\Gamma \vdash p \rightarrow p' : \{l_i : G_i\}_{i \in I} \triangleright \text{Type}} \text{ [KBRA]} \\ &\frac{\Gamma \vdash G \triangleright \text{Type}}{\Gamma \vdash \mu t. G \triangleright \text{Type}} \text{ [KREC]} \quad \frac{\Gamma \vdash \text{Env}}{\Gamma \vdash t \triangleright \text{Type}} \text{ [KTVAR]} \\ &\frac{\Gamma \vdash G \triangleright \kappa\{0/j\} \quad \Gamma, i : I^- \vdash G' \triangleright \text{Type}}{\Gamma \vdash \mathbf{R} G \lambda i : I^- . \lambda x. G' \triangleright \Pi j : I. \kappa} \text{ [KRCR]} \\ &\frac{\Gamma \vdash \text{Env}}{\Gamma \vdash x \triangleright \text{Type}} \text{ [KVAR]} \quad \frac{\Gamma \vdash \text{Env}}{\Gamma \vdash \text{end} \triangleright \text{Type}} \text{ [KEND]} \\ &\frac{\Gamma, i : I \vdash G \triangleright \kappa}{\Gamma \vdash \Pi i : I. G \triangleright \Pi i : I. \kappa} \text{ [KPI]} \quad \frac{\Gamma \vdash G \triangleright \Pi i : I. \kappa \quad \Gamma \models i : I}{\Gamma \vdash G \text{ i} \triangleright \kappa\{i/i\}} \text{ [KAPP]} \end{aligned}$$

**Figure 11.** Kinding rules for global types

Rule [KIO] states that if both participants have  $\text{nat}$ -type, that the carried type  $U$  and the rest of the global type  $G'$  are kinded by Type, and that  $U$  does not contain any free type variables, then the resulting type is well-formed. This prevents these types from being dependent. The rule [KBRA] is similar, while rules [KREC, KTVar] are standard. Dependent types are introduced when kinding recursors in [KRCR] and abstractions in [KPI]. In [KRCR], we need an updated index range for  $i$  in the premise  $\Gamma, i : I^- \vdash G' \triangleright \text{Type}$  since the index substitution uses the predecessor of  $i$ . We define  $I^-$  using the abbreviation  $[0..j]^- = \{i : \text{nat} \mid i \leq j\}$ :

$$[0..0]^- = \emptyset \quad \text{and} \quad [0..i]^- = [0..i-1]$$

We use [KAPP] for both index applications. Note that [KAPP] checks whether the argument  $i$  satisfies the index set  $I$ .

**Type equivalence** Since our types include dependent types and recursors, we need a notion of type equivalence. In order to keep type-checking decidable, we treat dependent types and recursors by  $\beta$ -reductions, and separately recursive types by isomorphism: (1) we reduce the two types  $G_1$  and  $G_2$  to their weak head normal forms and check whether they are equal or not,  $\text{whnf}(G_1) \equiv_{\text{wf}} \text{whnf}(G_2)$  (we extend the standard method [4, §2] with the recursor); and (2) we reduce them to their normal forms and check they are isomorphic. Formally, we add the following rules for the global type equality (similarly for other types including the process type). We write  $\equiv$  for  $\equiv^\circ$  or  $\approx^\circ$ .

$$\frac{\Gamma \vdash \text{whnf}(G_1) \equiv_{\text{wf}} \text{whnf}(G_2)}{\Gamma \vdash G_1 \equiv^\circ G_2} \quad \frac{\Gamma \vdash \text{nf}(G_1) \approx \text{nf}(G_2)}{\Gamma \vdash G_1 \approx^\circ G_2}$$

$$\begin{array}{c}
\frac{\Gamma \vdash \text{Env}}{\Gamma \vdash n \triangleright \text{nat}} [\text{TNAT}] \quad \frac{\Gamma \vdash U_p}{\Gamma \vdash \text{Alice} \triangleright U_p} [\text{TID}] \\
\frac{\Gamma \vdash p \triangleright \Pi i : I. U_p \quad \Gamma \models i : I}{\Gamma \vdash p[i] \triangleright U_p\{i/i\}} [\text{TP}] \\
\frac{\Gamma, i : I^-, X : \tau\{i/j\} \vdash Q \triangleright \tau\{i+1/j\} \quad \Gamma \vdash P \triangleright \tau\{0/j\} \quad \Gamma \vdash \Pi j : I. \tau \blacktriangleright \Pi j : I. \kappa}{\Gamma \vdash \mathbf{R} P \lambda i. \lambda X. Q \triangleright \Pi j : I. \tau} [\text{TPREC}] \\
\frac{\Gamma \vdash P \triangleright \tau \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash P \triangleright \tau'} [\text{TEQ}] \quad \frac{\Gamma, X : \tau \vdash \text{Env}}{\Gamma, X : \tau \vdash X \triangleright \tau} [\text{TVar}] \\
\frac{\Gamma, i : I \vdash \tau}{\Gamma \vdash \lambda i. P \triangleright \Pi i : I. \tau} [\text{TFUN}] \quad \frac{\Gamma \vdash P \triangleright \Pi i : I. \tau \quad \Gamma \models i \in I}{\Gamma \vdash P i \triangleright \tau\{i/i\}} [\text{TAPP}]
\end{array}$$

**Figure 12.** Process typing (Part 1)

**Typing Processes (Index and Recursion)** This paragraph and the next explain a selection of typing rules for processes. We start by the rules presented in figure 12. Rule [TNAT] and [TVar] are standard. For participants, we check their typing by [TP] and [TID] in a similar way as [31]. For the same reason as in [KRCR], [TPRCR] needs to deal with the changed index range within the recursion body. More precisely, we first check  $\tau$ 's kind. Then we verify as the base case ( $j = 0$ ) that  $P$  has type  $\tau\{0/j\}$ . Last, we check the more complex inductive case:  $Q$  should have type  $\tau\{i+1/j\}$  under the environment  $\Gamma, i : I^-, X : \tau\{i/j\}$  where  $\tau\{i/j\}$  of  $X$  means that  $X$  satisfies the predecessor's (induction hypothesis) type. Note that  $i$ 's range is  $I^-$  since  $i$  is assigned to the argument's predecessor. The rule [TEQ] states that typing works up to type equivalence (defined in the previous paragraph). Rules [TFUN] and [TAPP] correspond to introduction and elimination rules for the dependent types.

$$\begin{array}{c}
\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \vdash p_0 \quad \Gamma \vdash p_i \triangleright \text{nat} \quad \Gamma \models \text{pid}(G) = \{p_0 \dots p_n\}}{\Gamma \vdash \bar{u}[p_0 \dots p_n](y). P \triangleright \Delta} [\text{TREQ}] \\
\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \vdash p \quad \Gamma \vdash p \triangleright \text{nat} \quad \Gamma \models p \in \text{pid}(G)}{\Gamma \vdash u[p](y). P \triangleright \Delta} [\text{TACC}] \\
\frac{\Gamma \vdash e \triangleright S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!(p, e); P \triangleright \Delta, c : !\langle p, S \rangle; T} [\text{TOUT}] \\
\frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(p, x); P \triangleright \Delta, c : ?\langle p, S \rangle; T} [\text{TIN}] \\
\frac{\Gamma, a : U \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta} [\text{TNU}] \quad \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} [\text{TPAR}]
\end{array}$$

**Figure 13.** Process typing (Part 2)

**Typing Processes (Communication)** We explain here the communication part of the typing rules (figure 13). One of the benefits of our approach is that no modification is needed for most of the communication rules since indices are only instantiated by  $\lambda$ -application, but not by communication. The accept and request rules (dealing with session initiation) must be however modified as their participant arguments (e.g.  $i$  in  $\bar{W}[i]$ ) or even the number of participants (e.g.  $\{p_0, \dots, p_n\}$  in  $\bar{u}[p_0 \dots p_n](y). P$ ) are not statically known.

[TREQ] types a session request on shared name  $u$ , binding channel  $y$  and requiring participants  $p_0 \dots p_n$ . The premise verify that

the type of  $y$  is the first projection of the global type  $G$  of  $u$  and that the participants in  $G$  (denoted by  $\text{pid}(G)$ ) can be semantically derived as  $\{p_0 \dots p_n\}$ . Note that the position of  $p_0$  is fixed, but others can be viewed as a set that can be altered by its evaluation.

[TACC] allows to type the  $p$ -th participant to the session initiated on  $u$ . The typing rule checks that the type of  $y$  is the  $p$ -th projection of the global type  $G$  of  $u$ . Note that not only the participant's name, but also  $G$  itself can contain the free index, permitting the number of repetitions or the total number of participants to be variable. The kind rule for  $\Gamma \vdash \text{Env}$  (in [1]) ensures that  $G$  is not a dependent type (i.e.  $G$ 's kind is Type).

Rule [TPAR] puts in parallel two processes only if their sessions environments have disjoint domains. Other rules are standard. We especially omit inaction, branching/selection, delegation, recursion, its variable introduction rule and the expression typing rules. The typing rules for queues are identical with those in [7, 21].

### 3.3 Typing multiparty communication topologies

We type the process representation for communication topologies from Introduction and § 2.2.

**Repeated protocols** this example illustrates the use of recursors to express a varying number of repetitions. Let the type in (1) in the introduction as  $\Pi n. G(n)$ . We type Alice. Following the projection rule in figure 9, the projection of  $(\Pi n. G(n))n \equiv G(n)$  to Alice is:

$$G(n) \upharpoonright \text{Alice} = (\mathbf{R} \text{ end } \lambda i. \lambda x. !\langle \text{Bob}, \text{nat} \rangle; x \ n)$$

Let  $\Delta(n) = \{y : (\mathbf{R} \text{ end } \lambda i. \lambda x. !\langle \text{Bob}, \text{nat} \rangle; x \ n)\}$  and  $\Gamma = n : \text{nat}, a : \langle G \rangle$ . First we note  $\Gamma \vdash \Pi j : I. \Delta(j) \blacktriangleright \Pi j : I. \text{Type}$  by renaming with  $I = [0..n+1]$ . For the base case, we have:  $\Gamma \vdash \mathbf{0} \triangleright y : \text{end}$  by [TNULL] (in [1]). For the inductive case, first we derive, by [TVar],  $\Gamma, i : I^-, X : \Delta(i) \vdash X \triangleright \Delta(i)$ . From this judgement, using [TOUT], we have:

$$\Gamma, i : I^-, X : \Delta(i) \vdash y! \langle \text{Bob}, e[i] \rangle; X \triangleright \Delta(i+1)$$

From [TPREC] and [TAPP],  $\Gamma \vdash \mathbf{R} \mathbf{0} \lambda i. \lambda x. y! \langle \text{Bob}, e[i] \rangle; X \ n \triangleright \Delta(n)$ . Now we can apply [TREQ] to obtain  $\Gamma \vdash \text{Alice}(n) \triangleright \emptyset$ . Similarly for Bob( $n$ ) and Carol( $n$ ). Finally we can compose by [PAR],  $n : \text{nat}, a : \langle G \rangle \vdash \text{Alice}(n) \mid \text{Bob}(n) \mid \text{Carol}(n) \triangleright \emptyset$ . Applying [TNU] and [TFUN], we have:

$$\emptyset \vdash \lambda n. (\nu a)(\text{Alice}(n) \mid \text{Bob}(n) \mid \text{Carol}(n)) \triangleright \Pi n : \text{nat}. \emptyset$$

**Ring topology - figure 4(a)** The highlight of this example is a type-equality between the general end-point generator and a *role-based* local type (called *role-types*). The role-type groups the participants who inhabit in the same parameterised protocol as a single role. We observe the ring topology consists of the three roles (when  $n \geq 1$ ): one is the starter  $W[n]$  who sends the message first and receives the final message from the final worker  $W[0]$ , and others are the middle workers who receive a data and send back to the next worker. The transformation starts from the general generator of this topology given below.

$$\begin{array}{l}
\mathbf{R} (W[0] \rightarrow W[n] : \langle \text{nat} \rangle. \text{end}) \upharpoonright p \\
\lambda i. \lambda x. \text{if } p = W[i+1] \text{ then } !\langle W[i], \text{nat} \rangle; x \\
\quad \text{elseif } p = W[i] \text{ then } ?\langle W[i+1], \text{nat} \rangle; x \\
\quad \text{elseif } x \quad n
\end{array}$$

First we note the kinding rules [KRCR] and [KAPP] ensure  $n \geq 1$ . From the figure 4(a), the user would design the local type as follows:

$$\begin{array}{l}
\text{if } p = W[n] \text{ then } !\langle W[n-1], \text{nat} \rangle; ?\langle W[0], \text{nat} \rangle; \\
\text{elseif } p = W[0] \text{ then } ?\langle W[1], \text{nat} \rangle; !\langle W[n], \text{nat} \rangle; \\
\text{elseif } 1 \leq i \leq n-1 \text{ and } p = W[i] \\
\quad \text{then } ?\langle W[i+1], \text{nat} \rangle; !\langle W[i-1], \text{nat} \rangle;
\end{array}$$

The first case denotes the protocol of the initiator; the second one corresponds to the last worker, while the third one to one of the middle workers. The type equality is easily proved by the case analysis by the induction of the recursor. From these types, the ring

processes are straightforwardly implemented and typable using **R**: the whole process has type  $\Delta$  where  $\Delta$  is types for the free sessions of the initiator and the last worker (note the  $i$ -th process under the recursion should contain no free sessions). We also discuss an algorithm for the automatic generation of role-types in § 6.

## 4. Properties of the Typing System

### 4.1 Basic properties

We prove here a series of consistency lemmas concerning permutations and weakening. We start with the substitution lemma.

**LEMMA 4.1** (Substitution Lemma). *1. If  $\Gamma, i : I, \Gamma' \vdash J$  and  $\Gamma \models n : I$ , then  $\Gamma, (\Gamma' \{n/i\}) \vdash J \{n/i\}$ .  
2. If  $\Gamma, X : \Delta_0 \vdash P \triangleright \tau$  and  $\Gamma \vdash Q : \Delta_0$ , then  $\Gamma \vdash P\{Q/X\} \triangleright \tau$ .  
3. If  $\Gamma, x : S \vdash P \triangleright \Delta$  and  $\Gamma \vdash v : S$ , then  $\Gamma \vdash P\{v/x\} \triangleright \Delta$ .  
4. If  $\Gamma \vdash P \triangleright \Delta, y : T$ , then  $\Gamma \vdash P\{s[\hat{p}]/y\} \triangleright \Delta, s[\hat{p}] : T$ .*

Note that substitutions may change session types and environments in the index case. The application of (1) to process judgements is especially useful for the Subject Reduction Theorem: if  $\Gamma, i : I, \Gamma' \vdash P \triangleright \tau$  and  $\Gamma \vdash n \triangleright \text{nat}$  with  $\Gamma \models n : I$ , then  $\Gamma, (\Gamma' \{n/i\}) \vdash P\{n/i\} \triangleright \tau \{n/i\}$ .

We also use the following lemma for the branching local types whose projections are extended by the mergeability operator  $\bowtie$ . Below,  $\leq$  denotes the standard branching subtyping [17, 21] defined by the selection rule (if  $\forall i \in I \subseteq J, \Gamma \vdash T_i \leq T'_i$ , then  $\Gamma \vdash \oplus(p, \{l_i : T_i\}_{i \in I}) \leq \oplus(p, \{l_j : T'_j\}_{j \in J})$ ) and the dual branching rule.

**LEMMA 4.2** (Soundness of mergeability). *Suppose  $G_1 \uparrow p \bowtie G_2 \uparrow p$  and  $\Gamma \vdash G_i$ . Then there exists  $G$  such that  $G \uparrow p = \sqcap \{T \mid T \leq G_i \uparrow p \ (i = 1, 2)\}$  where  $\sqcap$  denotes the maximum element with respect to  $\leq$ .*

This lemma states that mergeability is sound with respect to the branching subtyping — we can safely replace the third clause  $\sqcup_{i \in I} G_i \uparrow q$  of the branching in figure 9 by  $\sqcap \{T \mid \forall i \in I. T \leq (G_i \uparrow q)\}$ . This lemma allows us to prove subject reduction by including the subsumption in the runtime typing system as done in [21, § 5].

Ensuring termination of type-checking with dependent types is not an easy task since the type equivalences are often defined from term equivalences. To prove the termination, we first note that by strong normalisation of System  $\mathcal{T}$  [18], and the definition of  $\longrightarrow$ , the relation  $\longrightarrow$  on global and local types (i.e.  $G \longrightarrow G'$  and  $T \longrightarrow T'$ ) are strong normalising and confluent on well-formed kinds. Secondly we note that we do not require to prove the termination of process reduction since term equivalence is not used in our typing rules (cf. [31]). Then the termination of the type equivalence checking is proved by the termination of  $\beta$ -equality (by recursors and dependent type reductions) and isomorphism checking (proved decidable in the literature [17]). Below we assume the bound names and variables in  $P$  are annotated (e.g.  $(\nu a : (G))P$ ) following the standard manner [21].

**PROPOSITION 4.3** (Termination for Type-Checking). *Assuming that proving the judgements  $\Gamma \models J$  appearing in kinding, equality, projection and typing derivations is decidable (e.g. in [KPROJ]), then type-checking of  $\Gamma \vdash P \triangleright \emptyset$  terminates.*

To ensure the termination of  $\Gamma \models J$ , several solutions include the restriction of predicates to linear equalities over the natural numbers without multiplications (or to other decidable arithmetic subsets) or the restriction of indices to finite domains, cf. [31].

### 4.2 Subject reduction

As session environments record channel states, they evolve when communications proceed. This can be formalised by introducing a notion of session environments reduction. These rules are formalised below modulo  $\equiv$ .

- $\{s[\hat{p}] : !(\hat{q}, U); T, s[\hat{q}] : ?(\hat{p}, U); T'\} \Rightarrow \{s[\hat{p}] : T, s[\hat{q}] : T'\}$
- $\{s[\hat{p}] : T; \oplus(\hat{q}, \{l_i : T_i\}_{i \in I})\} \Rightarrow \{s[\hat{p}] : T; \oplus(\hat{q}, l_i); T_i\}$
- $\{s[\hat{p}] : \oplus(\hat{q}, l_j); T, s[\hat{q}] : \&(p, \{l_i : T_i\}_{i \in I})\} \Rightarrow \{s[\hat{p}] : T, s[\hat{q}] : T_j\}$
- $\Delta \cup \Delta' \Rightarrow \Delta' \cup \Delta''$  if  $\Delta \Rightarrow \Delta'$ .

The first rule corresponds to the reception of a value or channel by the participant  $\hat{q}$ ; the second rule treats the case of the choice of label  $l_j$  while the third rule propagate these choices to the receiver (participant  $\hat{q}$ ). Using the above notion we can state type preservation under reductions as follows:

**THEOREM 4.4** (Subject Congruence and Reduction).

- If  $\Gamma \vdash P \triangleright \Delta$  and  $P \equiv P'$ , then  $\Gamma \vdash P' \triangleright \Delta$ .
- If  $\Gamma \vdash P \triangleright \tau$  and  $P \longrightarrow^* P'$ , then  $\Gamma \vdash P' \triangleright \tau'$  for some  $\tau'$  such that  $\tau \Rightarrow^* \tau'$ .

Note that communication safety [21, Theorem 5.5] and session fidelity [21, Corollary 5.6] are corollaries of the above theorem. Progress [21, Theorem 5.6] can be also obtained by a similar method as the one found in [21].

**Proof.** We list only the most interesting cases for recursor, which uses mathematical induction in addition to induction on the derivation. **Case ZeroR:** Trivial.

**Case SuccR:** Suppose  $\Gamma \vdash \mathbf{R} P \lambda i. \lambda X. Q \ n + 1 \triangleright \tau$  and  $\mathbf{R} P \lambda i. \lambda X. Q \ n + 1 \longrightarrow P\{n/i\}\{\mathbf{R} P \lambda i. \lambda X. Q \ n/X\}$ . Then there exists  $\tau'$  such that

$$\Gamma, i : I^-, X : \tau\{i/j\} \vdash Q \triangleright \tau'\{i + 1/j\} \quad (2)$$

$$\Gamma \vdash P \triangleright \tau'\{0/i\} \quad (3)$$

$$\Gamma \vdash \Pi j : I. \tau \triangleright \Pi j : I. \kappa \quad (4)$$

with  $\tau \equiv (\Pi i : I. \tau')n + 1 \equiv \tau'\{n + 1/i\}$  and  $\Gamma \models n + 1 : I$ . By Substitution Lemma (Lemma 4.1 (1)), noting  $\Gamma \models n : I^-$ , we have:  $\Gamma, X : \tau\{i/j\}\{n/i\} \vdash Q\{n/i\} \triangleright \tau'\{i + 1/j\}\{n/i\}$ , which means that

$$\Gamma, X : \tau\{n/j\} \vdash Q\{n/i\} \triangleright \tau'\{n + 1/j\} \quad (5)$$

Then there are two cases.

**Base Case**  $n = 0$ : By applying Substitution Lemma (Lemma 4.1 (2)) to (5) with (3), we have  $\Gamma \vdash Q\{1/i\}\{P/X\} \triangleright \tau'\{1/j\}$ .

**Inductive Case**  $n \geq 1$ : By the inductive hypothesis on  $n$ , we assume:  $\Gamma \vdash \mathbf{R} P \lambda i. \lambda X. Q \ n \triangleright \tau'\{n/j\}$ . Then by applying Substitution Lemma (Lemma 4.1) to (5) with this hypothesis, we obtain  $\Gamma \vdash Q\{n/i\}\{\mathbf{R} P \lambda i. \lambda X. Q \ n/X\} \triangleright \tau'\{n + 1/j\}$ .

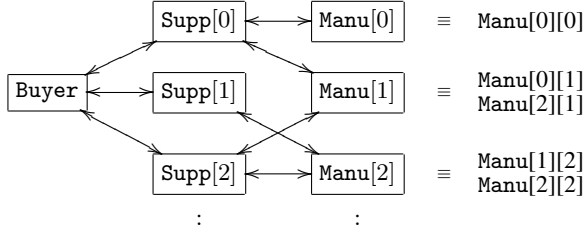
The proof of the communication rules uses the subtyping relation and the Substitution Lemma (4.1 (3), (4)), together with the subtyping rule for the extended mergeability (Lemma 4.2).  $\square$

## 5. Web Service Choreography and FFT

This section demonstrates the expressiveness of our type theory. We first program and type a real-world Web service usecase: Quote Request (C-U-002) is the most complex scenario described in [3], the public document authored by the W3C Choreography Description Language Working Group [30]. We then conclude by typing our implementation of the FFT algorithm and prove its correctness.

### 5.1 Choreography of Interactions in Web Services

**Quote Request usecase** The usecase is described below (as published in [3]). A buyer interacts with multiple suppliers who in turn



**Figure 14.** The Quote Request usecase (C-U-002) [3]

interact with multiple manufacturers in order to obtain quotes for some goods or services. The steps of the interaction are:

1. A buyer requests a quote from a set of suppliers. All suppliers receive the request for quote and send requests for a bill of material items to their respective manufacturers.
2. The suppliers interact with their manufacturers to build their quotes for the buyer. The eventual quote is sent back to the buyer.
3. EITHER
  - (a) The buyer agrees with one or more of the quotes and places the order or orders. OR
  - (b) The buyer responds to one or more of the quotes by modifying and sending them back to the relevant suppliers.
4. EITHER
  - (a) The suppliers respond to a modified quote by agreeing to it and sending a confirmation message back to the buyer. OR
  - (b) The supplier responds by modifying the quote and sending it back to the buyer and the buyer goes back to STEP 3. OR
  - (c) The supplier responds to the buyer rejecting the modified quote. OR
  - (d) The quotes from the manufacturers need to be renegotiated by the supplier. Go to STEP 2.

The usecase, depicted in figure 14, may seem simple, but it contains many challenges. The Requirements in Section 3.1.2.2 of [3] include: **[R1]** the ability to repeat the same set of interactions between different parties using a single definition and to compose them; **[R2]** the number of participants may be bounded at design time or at runtime; and **[R3]** the ability to *reference a global description from within a global description* to support *recursive behaviour* as denoted in STEP 4(b, d). The following works through a parameterised global type specification that satisfies these requirements.

**Modular programming using global types** We develop the specification of the usecase program modularly, starting from smaller global types. Here, *Buyer* stands for the buyer, *Supp* $[i]$  for a supplier, and *Manu* $[j]$  for a manufacturer. Then we alias manufacturers by *Manu* $[i][j]$  to identify that *Manu* $[j]$  is connected to *Supp* $[i]$  (so a single *Manu* $[j]$  can have multiple aliases *Manu* $[i']$  $[j]$ , see figure 14). Then, using the idioms presented in § 1, STEP 1 is defined as:

$$G_1 = \text{foreach}(i : I) \{ \text{Buyer} \rightarrow \text{Supp}[i] : \langle \text{Quote} \rangle . \text{end} \}$$

For STEP 2, we compose a nested loop and the subsequent action within the main loop ( $J_i$  gives all *Manu* $[j]$  connected to *Supp* $[i]$ ):

$$\begin{aligned} G_2 &= \text{foreach}(i : I) \{ \text{comp}(G_2[i], \text{Supp}[i] \rightarrow \text{Buyer} : \langle \text{Quote} \rangle . \text{end}) \} \\ G_2[i] &= \text{foreach}(j : J_i) \{ \text{Supp}[i] \rightarrow \text{Manu}[i][j] : \langle \text{Item} \rangle . \\ &\quad \text{Manu}[i][j] \rightarrow \text{Supp}[i] : \langle \text{Quote} \rangle . \text{end} \} \end{aligned}$$

$G_2[i]$  represents the second loop between the  $i$ -th supplier and its manufacturers. Regarding STEP 3, the specification involves buyer preference for certain suppliers. Since this can be encoded using dependent types (like the encoding of if), we omit this part and assume the preference is given by the (reverse) ordering of  $I$  in order

to focus on the description of the interaction structure.

$$\begin{aligned} G_3 &= \mathbf{R} \ t \ \lambda i. \lambda y. \text{Buyer} \rightarrow \text{Supp}[i] : \{ \\ &\quad \text{ok} : \quad \text{end} \\ &\quad \text{modify} : \text{Buyer} \rightarrow \text{Supp}[i] : \langle \text{Quote} \rangle \\ &\quad \quad \text{Supp}[i] \rightarrow \text{Buyer} : \{ \text{ok} : \quad \text{end} \\ &\quad \quad \quad \text{retryStep3} : \text{y} \\ &\quad \quad \quad \text{reject} : \quad \text{end} \} \} i \end{aligned}$$

In the innermost branch, *ok*, *retryStep3* and *reject* correspond to STEP 4(a), (b) and (c) respectively. Type variable  $t$  is for (d). We can now compose all these subprotocols together. Taking  $G_{23} = \mu t. \text{comp}(G_2, G_3)$  and assuming  $I = [0..i]$ , the full global type is

$$\lambda i. \lambda \tilde{J}. \text{comp}(G_1, G_{23})$$

where we have  $i$  suppliers, and  $\tilde{J}$  gives the  $J_i$  (continuous) index sets of the *Manu* $[j]$ s connected with each *Supp* $[i]$ .

**End-point types** We show the local type for suppliers, who engage in the most complex interaction structures among the participants. The projections corresponding to  $G_1$  and  $G_2$  are straightforward:

$$\begin{aligned} G_1 \upharpoonright \text{Supp}[n] &= ? \langle \text{Buyer}, \text{Quote} \rangle \\ G_2 \upharpoonright \text{Supp}[n] &= \text{foreach}(j : J_i) \{ ! \langle \text{Manu}[n][j], \text{Item} \rangle ; \\ &\quad ? \langle \text{Manu}[n][j], \text{Quote} \rangle \} ; ! \langle \text{Buyer}, \text{Quote} \rangle \end{aligned}$$

For  $G_3 \upharpoonright \text{Supp}[n]$ , we use the branching injection and mergeability theory developed in § 3.1. After the relevant application of  $\lfloor \text{TEQ} \rfloor$ , we can obtain the following projection:

$$\begin{aligned} \&\langle \text{Buyer}, \{ \text{ok} : \quad \text{end} \\ &\quad \text{modify} : \ ? \langle \text{Buyer}, \text{Quote} \rangle ; \oplus \langle \text{Buyer}, \{ \\ &\quad \quad \text{ok} : \quad \text{end} \\ &\quad \quad \text{retryStep3} : \ T \\ &\quad \quad \text{reject} : \quad \text{end} \} \} \rangle \end{aligned}$$

where  $T$  is a type for the invocation from Buyer:

$$\begin{aligned} \text{if } n \leq i \text{ then } \&\langle \text{Buyer}, \{ \text{closed} : \text{end}, \text{retryStep3} : t \} \rangle \\ \text{elseif } i = n \text{ then } t \end{aligned}$$

To tell the other suppliers whether the loop is being reiterated or if it is finished, we can simply insert the following closing notification  $\text{foreach}(j : I \setminus i) \{ \text{Buyer} \rightarrow \text{Supp}[j] : \{ \text{close} : \} \}$  before each end, and a similar retry notification (with label *retryStep3*) before  $t$ . Finally, each end-point type is formed by the following composition:

$$\text{comp}(G_1 \upharpoonright \text{Supp}[n], \mu t. \text{comp}(G_2 \upharpoonright \text{Supp}[n], G_3 \upharpoonright \text{Supp}[n]))$$

Following this specification, the local projections can be implemented in various end-point languages (such as CDL or BPEL). We have implemented this usecase [1] (with the number of participants bound at runtime) in SJ [22]; see § 6 for further notes on implementation experience.

## 5.2 Correctness of FFT processes

We prove here that the processes given in the FFT example in § 2.5 are typable against the given global type. Their termination (i.e. deadlock-freedom) is then obtained as a corollary.

We assume index  $n$  to be a parameter as in figure 7. The main loop is an iteration over the  $n$  steps of the algorithm. We first project the global type given in 7(c) directly, and show how it is equated to the target end-point type.

Forgetting for now the content of the main loop, the generic projection for machine  $p$  has the following skeleton:

$$\Pi n. (\mathbf{R} \ (\mathbf{R} \ \text{end} \ \lambda l. \lambda x. (\dots) \ n) \\ \lambda k. \lambda u. \text{if } p = k \text{ then } ! \langle k, U \rangle ; ? \langle k, U \rangle ; \mathbf{u} \ \text{else} \ \mathbf{u}) \ 2^n$$

A simple induction on  $n$  gives us the equivalent type:

$$\Pi n. ! \langle p, U \rangle ; ? \langle p, U \rangle ; (\mathbf{R} \ \text{end} \ \lambda l. \lambda x. (\dots) \ n) \ 2^n$$

We now consider the inner loops. The generic projection gives

nested sequences of conditionals of the form:

```
if  $p = i * 2^{n-l} + 2^{n-l-1} + j = i * 2^{n-l} + j$  then ...
else if  $p = i * 2^{n-l} + 2^{n-l-1} + j$  then  $!\langle i * 2^{n-l} + j, U \rangle; \dots$  else
```

The above code is translated to the following two branches where the first one corresponds to the upper part of the butterfly while the second one corresponds to the lower part in figure 7(a).

```
if  $\text{bit}_{n-l}(p) = 0$ 
then  $?\langle p + 2^{n-l-1}, U \rangle; !\langle p + 2^{n-l-1}, U \rangle; !\langle p, U \rangle; ?\langle p, U \rangle; x$ 
else  $!\langle p - 2^{n-l-1}, U \rangle; ?\langle p - 2^{n-l-1}, U \rangle; !\langle p, U \rangle; ?\langle p, U \rangle; x$ 
```

We use an induction over  $p$  and simple arithmetic over binary numbers to obtain the above type. For programming reasons (as seen in the processes, the natural implementation starts by sending a first initialisation message with the  $x_k$  value), we want to shift the self-receive  $?\langle p, U \rangle$ ; from the initialisation to the beginning of the loop iteration at the price of adding the last self-receive to the end:  $?\langle p, U \rangle$ ; end. The resulting equivalent type up to  $\equiv$  is:

```
 $\Pi n. !\langle p, U \rangle;$ 
 $(R. ?\langle p, U \rangle); \text{end } \lambda l. \lambda x.$ 
if  $\text{bit}_{n-l}(p) = 0$ 
then  $?\langle p, U \rangle; ?\langle p + 2^{n-l-1}, U \rangle; !\langle p + 2^{n-l-1}, U \rangle; !\langle p, U \rangle; x$ 
else  $?\langle p, U \rangle; !\langle p - 2^{n-l-1}, U \rangle; ?\langle p - 2^{n-l-1}, U \rangle; !\langle p, U \rangle; x \rangle n$ 
```

From this local type, implementing and typing the processes defined in figure 7(c) in § 2.5 becomes straightforward.

Finally we prove type-safety and deadlock-freedom for the FFT processes. Let  $P_{\text{fit}}$  be the following process:

$$P_{\text{fit}} = \lambda n. (\nu a) (R. \bar{a}[p_0..p_n](y). P(n, p_0, x_{\bar{p}_0}, y) \\ \lambda i. \lambda Y. (\bar{a}[p_{i+1}](y). P(i+1, p_{i+1}, x_{\bar{p}_{i+1}}, y) \mid Y) n)$$

As we reasoned above, each  $P(n, p, x_{\bar{p}}, y)$  is straightforwardly typable by the local type which is equivalent with one projected from  $G$  listed above. Checking the correctness of the projection (which we obtained by induction) is not easy though: we need here to rely on the finite domain restriction. We note, however, that once  $P_{\text{fit}}$  is applied to a natural number  $m$ , it always terminates as indices strictly decrease at each recursor application. We finally get:

**THEOREM 5.1 (Type Safety and Deadlock-freedom of FFT).** *For all  $m$ ,  $\emptyset \vdash P_{\text{fit}} m \triangleright \emptyset$ ; and for all  $Q$  such that  $P_{\text{fit}} m \longrightarrow^* Q$ , we have  $Q \longrightarrow^* 0$ .*

## 6. Extensions

**Role-type projection** As we discussed in the ring example in § 2.4, we have also investigated an alternative *role-based approach*, for checking the conformance of processes against global type specifications. Although this method currently works only if all participants are parameterised and without index multiplication, we can generate role-types for participants directly without using type-equality, dispensing with recursors. The projection algorithm returns the distinct sets of role-types that are possible for a given global type, which are then used to type-check the processes. Taking the example of the ring topology defined in §3.3 (figure 4a), one set of role-types generated by the projection has the following:

```
Starter  $\triangleq !\langle W[n-1], \text{nat} \rangle; ?\langle W[0], \text{nat} \rangle; \text{end}@W[n],$ 
Middle  $\triangleq ?\langle W[i+1], \text{nat} \rangle; !\langle W[i-1], \text{nat} \rangle; \text{end}@W[i],$ 
Final  $\triangleq ?\langle W[1], \text{nat} \rangle; !\langle W[n], \text{nat} \rangle; \text{end}@W[0],$ 
```

where  $1 \leq i \leq n-1$  and  $T@W[i]$  denotes role-type  $T$  assigned to participant  $W[i]$ . The projection algorithm also returns an (inequality) on  $n$  for which each role-type set is valid, e.g. the above role-types are valid for  $n \geq 2$ . For  $n = 1$ , the role-types generated are only Starter (which becomes  $W[1]$ ) and Final. For  $n = 0$ , the single role-type generated is  $!\langle W[0], \text{nat} \rangle; ?\langle W[0], \text{nat} \rangle; \text{end}@W[0]$ , a different role to the ones defined above. Briefly, this projection algorithm works over 4 stages: calculation of the participant ranges for

each reference to  $i$ -indexed participants, intersection of these ranges, arrangement of actions in a role-type, and concatenation of the role-types for  $i = n$  and  $i = 0$ . For the Middle role of the ring example, the range of  $W[i]$  is  $[W[n-1]..W[0]]$  since  $0 \leq i \leq n-1$  from the definition of  $R$ ; the intersection of the ranges between  $W[i]$  and  $W[i+1]$ , i.e.  $[W[n-1]..W[1]]$ , is defined for  $n \geq 2$  and denotes the participants that will perform the actions defined for both  $W[i]$  and  $W[i+1]$ ; the actions  $?\langle W[i+1], \text{nat} \rangle$  and  $!\langle W[i-1], \text{nat} \rangle$  are arranged in this order as determined from the global type; and finally  $x$  is substituted by the case for  $i = 0$ . Similarly for Starter and Final, and the role-types for  $n = 1$  and  $n = 0$ .

**Implementation experience** We discuss some preliminary benchmark results that demonstrate how the greater expressiveness of dependent multiparty types in comparison to binary sessions permits significant performance improvement. An overview of parallel algorithm implementation using SJ, [2, 22], an extension of Java for *binary* session programming with multicast constructs, was discussed in [5]. The features of SJ make it a suitable target language for (an adapted) dependent global type projection, giving the main interaction structures for concrete, executable end-point implementations.

One of the algorithms implemented in [5] is a parallel simulation of the  $n$ -Body Problem [19]. The  $n$ -Body simulation features an advanced interaction structure based on the ring topology, for which the projection algorithm has been outlined above. ([5] also presents session-typed implementations of algorithms featuring other topologies, such as mesh-based Jacobi iteration.) Due to the restrictions of binary session typing, the direct SJ implementation involves the creation of one fresh session per simulation step between Starter and Final (the final ring link) in addition to the main sessions between each of the other neighbouring nodes [5]. Without transport-specific runtime support for reusing old connections, this can become a costly overhead. However, with dependent global types, we can design the behaviour of each role clearly within the construction of the whole multiparty protocol, completely avoiding this nested session problem. Communication-safety and deadlock-freedom for the resulting implementation are ensured for any configuration of  $m > 1$  parties.

In the experiment, we compared the performance of the original binary session version against the implementations derived from projection (for the purposes of practical benchmarks, we also make use of asynchronous communication subtyping [25] after projection). The benchmark was executed in a low latency cluster environment (TCP over gigabit Ethernet), varying the number of processes, simulations steps and particles (distributed evenly). As expected, the latter, without the nested session overhead, performs significantly better across all parameter combinations; for instance, with 10 processes performing 100 simulations steps on 100 particles each, the direct SJ implementation took on average 180s whilst the projected implementation took 103s, an improvement of 43%. The full source code for the benchmark applications, exact benchmark environment details and the complete results can be found at [1].

## 7. Related Work

**Dependent types** The idea of using primitive recursive functionals for dependent types comes from Nelson's  $\mathcal{T}^\pi$  in [26] for the  $\lambda$ -calculus, which is (as he stated) a rediscovery of  $\mathcal{T}^\infty$  by Tait and Martin L  f [24, 29] for forming infinite sequences of terms and types. The system [26] can type functions previously untypable in ML, and the finite representability of dependent types makes it possible to construct a type-reconstruction algorithm. We also use aspects from the DML dependent typing system in [4, 31] where type dependency is only allowed for index sorts, hence type-checking can be reduced to a constraint-solving problem over indices. Our design choice to combine both systems gives (1) the simplest formulation of finite growing sequences of global and local types and processes

based on the recursor; (2) a precise specification for parameters appearing in the participants based on index sorts; and (3) a clear integration with the full session types and general recursion, whilst ensuring decidability of type-checking (if the constraint-solving problem is decidable). From the basis of these works, our type equivalence does not have to rely on behavioural equivalence between processes, but only strongly normalising *types* represented by recursors.

Dependent types have been also studied in the context of process calculi, where the dependency centres on locations (e.g. [20]), and channels (e.g. [32]) for mobile agents or higher-order processes. An effect-based session typing system for corresponding assertions to specify fine-grained communication specifications is studied in [9] where effects can appear both in types and processes. None of these investigate growing global specifications using dependent types. Our main typing rules require a careful treatment for type soundness not found in the previous works, due to the simultaneous instantiation of terms and indices by the recursor, with reasoning by mathematical induction (note that type soundness was left open in [26]).

### **Multiparty session types and other typing systems for processes**

The first papers on multiparty session types were [8] and [21], the former uses a distributed calculus where each channel connects a master end-point to one or more slave endpoints; instead of global types, they use only local types. After [21], several extensions have been studied in [6, 7, 25, 27]. One of the advantages of our typing system in the present work is that it can be easily applied to these previously developed multiparty typing systems since no changes to the runtime typing components would be needed. The work [12] presented an *executable language* for Web interactions in the form of global processes and provided the framework for projecting to local processes. The use of global descriptions as *types* had not been developed in [12], and the type disciplines for the two global and local process calculi in [12] are based on *binary* session types; hence safety and progress for multiparty interactions were not considered, while the current paper offers safe, parameterised multiparty interactions towards enriching the facility of global types.

Formal theories of contracts using multiparty interaction structures other than multiparty session types are studied in [14] using CCS-like processes as a type representation. The recent work [15] extends [14] with the treatment of mobile and forwarding channels, comparing its expressiveness with session types. Another recent work [11] presents a typed calculus for service orientation by extending the  $\pi$ -calculus with context-sensitive interactions, equipped with service and request primitives. Our method differs from these approaches since we start from programming general topologies as global specifications, hence various dynamic, parameterised communications can be explicitly described for a shared agreement among multiple peers. Our system allows participant identities to change during program execution by index instantiation, which is effective, in particular, for data exchange across complex structures such as the butterfly topology. The work [10] proposes a distributed calculus with sessions that incorporates the merging of running sessions; however, type-safety for interleaved sessions is left as an open problem. Parameterisation and repetition are widely appearing idioms in most parallel algorithms and choreographic interactions, and with session delegations, they can represent a mechanism for merging conversations with progress combining with [7]; extensions to dynamic features such as late joining and a service discovery facility are interesting research topics. The preceding works do not treat the main technical problems addressed in the present work — programming methodology and a formal system for dynamically changing global specifications and type-safety via a general projection method in a dependent type format, backed-up by efficient type-checking and resource usage, and ensuring strong safety properties for complex topologies based on the parameterised multiparty session type discipline.

## **References**

- [1] An Online Appendix of this paper. <http://www.doc.ic.ac.uk/~yoshida/dependent/>.
- [2] SJ homepage. <http://www.doc.ic.ac.uk/~rhu/sessionj.html>.
- [3] Web Services Choreography Requirements (No. 11). <http://www.w3.org/TR/ws-chor-reqs/>.
- [4] D. Aspinall and M. Hofmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT, 2005.
- [5] A. Bejleri, R. Hu, and N. Yoshida. Session-based programming for parallel algorithms: Expressiveness and performance. In *PLACES'09*, 2009. [http://www.doc.ic.ac.uk/~ab406/parallel\\_algorithms.html](http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html).
- [6] A. Bejleri and N. Yoshida. Synchronous multiparty session types. In *PLACES'08*, ENTCS. Elsevier, 2009. To appear.
- [7] L. Bettini et al. Global progress in dynamically interfered multiparty sessions. In *CONCUR'08*, volume 5201 of *LNCS*, 2008.
- [8] E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, 2008.
- [9] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.
- [10] R. Bruni et al. Multiparty Sessions in SOC. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.
- [11] L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
- [12] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17, 2007.
- [13] M. Carbone, N. Yoshida, and K. Honda. Asynchronous session types: Exceptions and multiparty interactions. In *SFM'09*, volume 5569 of *LNCS*, pages 187–212. Springer, 2009.
- [14] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL*, pages 261–272, 2008.
- [15] G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, *LNCS*, 2009. To appear.
- [16] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
- [17] S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
- [18] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. CUP, 1989.
- [19] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.
- [20] M. Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
- [21] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
- [22] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
- [23] F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1991.
- [24] P. Martin-Löf. Infinite terms and a system of natural deduction. In *Compositio Mathematica*, pages 93–103. Wolters-Noordhoff, 1972.
- [25] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.
- [26] N. Nelson. Primitive recursive functionals with dependent types. In *MFPS*, volume 598 of *LNCS*, pages 125–143, 1991.
- [27] L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sum types. Technical Report DTR09-8, Computing, Imperial College, 2009.
- [28] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [29] W. W. Tait. Infinitely long terms of transfinite type. In *Formal Systems and Recursive Functions*, pages 177–185. North Holland, 1965.
- [30] Web Services Choreography Working Group. Choreography Description Language. <http://www.w3.org/2002/ws/chor/>.
- [31] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.
- [32] N. Yoshida. Channel dependent types for higher-order mobile processes. In *POPL '04*, pages 147–160. ACM Press, 2004.