Parameterised Multiparty Session Types ⋆

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu

Department of Computing, Imperial College London

Abstract. For many application-level distributed protocols and parallel algorithms, the set of participants, the number of messages or the interaction structure are only known at run-time. This paper proposes a dependent type theory for multiparty sessions which can statically guarantee type-safe, deadlock-free multiparty interactions among processes whose specifications are parameterised by indices. We use the primitive recursion operator from Gödel's System $\mathcal{T}$ to express a wide range of communication patterns while keeping type checking decidable. 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. We prove the termination of the type-checking algorithm in the full system with both multiparty session types and recursive types. We illustrate our type theory through non-trivial programming and verification examples taken from parallel algorithms and Web services usecases.

1 Introduction

As the momentum around communications-based computing grows, the need for effective frameworks to globally *coordinate* and *structure* the application-level interactions is pressing. The structures of interactions are naturally distilled as *protocols*. Each protocol describes a bare skeleton of how interactions should proceed, through e.g. sequencing, choices and repetitions. In the theory of multiparty session types [3, 6, 23], such protocols can be captured as types for interactions, and type checking can statically ensure runtime safety and fidelity to a stipulated protocol.

One of the particularly challenging aspects of protocol descriptions is the fact that many actual communication protocols are highly *parametric* in the sense that the number of participants and even the interaction structure itself are not fixed at design time. Examples include parallel algorithms such as the Fast Fourier Transform (run on any number of communication nodes depending on resource availability) and Web services such as business negotiation involving an arbitrary number of sellers and buyers. This nature is important, for instance, for the programmer of a parallel algorithm where the size or shape of the communication topology, or the number of available threads might be altered depending on the number of available cores in the machine. Another scenario is web services where the participant sets may be known at design time, or instantiated later. This paper introduces a robust dependent type theory which can statically ensure communication-safe, deadlock-free process interactions which follow parameterised multiparty protocols.

⋆ The work is partially supported by EPSRC EP/G015635/1 and EP/F003757/1.

We illustrate the key ideas of our proposed parametric type structures through examples. Let us first consider a simple protocol where participant `Alice` sends a message of type nat to participant `Bob`. To develop the code for this protocol, we start by specifying the global type, which can concisely and clearly describe a high-level protocol for multiple participants [3, 23, 30], as follows (end denotes protocol termination):

$$G_1 \;=\; \texttt{Alice} \to \texttt{Bob}\colon \langle \mathsf{nat} \rangle.\mathsf{end}$$

The flow of communication is indicated with the symbol $\to$ and upon agreement on G_1 as a specification for `Alice` and `Bob`, each program can be implemented separately, e.g. as $y!\langle 100 \rangle$ (output 100 to y) by `Alice` and $y?(z); \mathbf{0}$ (input at y) by `Bob`. For type-checking, G_1 is *projected* into end-point session types: one from `Alice`'s point of view, $!\langle \texttt{Bob}, \mathsf{nat} \rangle$ (output to `Bob` with nat-type), and another from `Bob`'s point of view, $?\langle \texttt{Alice}, \mathsf{nat} \rangle$ (input from `Alice` with nat-type), against which the respective `Alice` and `Bob` programs are checked to be compliant.

The first step towards generalised type structures for multiparty sessions is to allow modular specifications of protocols using arbitrary compositions and repetitions of interaction units (this is a standard requirement in multiparty contracts [39]). Consider the type $G_2 \;=\; \texttt{Bob} \to \texttt{Carol}\colon \langle \mathsf{nat} \rangle.\mathsf{end}$. The designer may wish to compose sequentially G_1 and G_2 together to build a larger protocol:

$$G_3 = G_1; G_2 = \texttt{Alice} \to \texttt{Bob}\colon \langle \mathsf{nat} \rangle.\texttt{Bob} \to \texttt{Carol}\colon \langle \mathsf{nat} \rangle.\mathsf{end}$$

We may also want to iterate the composed protocols n-times, which can be written by $\texttt{foreach}(i < \mathrm{n})\{G_1; G_2\}$, and moreover bind the number of iteration n by a dependent product to build a *family of global specifications*, as in (Πn binds variable n):

$$\Pi n.\texttt{foreach}(i < n)\{G_1; G_2\} \tag{1}$$

Beyond enabling a variable number of exchanges between a fixed set of participants, the ability to parameterise *participant identities* can represent a wide class of the communication topologies found in the literature. For example, the use of indexed participants $\texttt{W}[i]$ (denoting the i-th worker) allows to specify a family of session types such that neither the number of participants nor message exchanges are known before the runtime instantiation of the parameters. The following type and diagram both describe a sequence of messages from $\texttt{W}[n]$ to $\texttt{W}[0]$ (indices decrease in our `foreach`, see § 2):

$$\Pi n.(\texttt{foreach}(i < n)\{\texttt{W}[i+1] \to \texttt{W}[i]\colon \langle \mathsf{nat} \rangle\}) \qquad \boxed{\mathrm{n}} \longrightarrow \boxed{\mathrm{n\text{-}1}} \longrightarrow \cdots \longrightarrow \boxed{0} \tag{2}$$

Here we face an immediate question: *what is the underlying type structure for such parametrisation, and how can we type-check each (parametric) end-point program?* The type structure should allow the projection of a parameterised global type to an end-point type *before* knowing the exact shape of the concrete topology.

In (1), corresponding end-point types are parameterised *families* of session types. For example, `Bob` would be typed by $\Pi j.\texttt{foreach}(i < j)\{?\langle \texttt{Alice}, \mathsf{nat} \rangle; !\langle \texttt{Carol}, \mathsf{nat} \rangle\}$, which represents the product of session interactions with different lengths. The choice is made when j is instantiated, i.e. before execution. The difficulty of the projection arises

in (2): if $\mathrm{n} \geq 2$, there are three distinct *communication patterns* inhabiting this specification: the initiator $\mathtt{W}[\mathrm{n}]$ (send only), the $\mathrm{n}-1$ middle workers (receive then send), and the last worker $\mathtt{W}[0]$ (receive only). This is no longer the case when $\mathrm{n}=1$ (there is only the initiator and the last worker) or when $\mathrm{n}=0$ (no communication at all). Can we provide a decidable projection and static type-checking by which we can preserve the main properties of the session types such as progress and communication-safety in parameterised process topologies? The key technique proposed in this paper is 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 represent the class of all possible end-point types.

The main contributions of this paper follow:

- *A new expressive framework to globally specify and program* a wide range of parametric communication protocols (§ 2). We achieve this result by combining dependent type theories derived from Gödel's System $\mathcal{T}$ [31] (for expressiveness) and indexed dependent types from [40] (for parameter control), with multiparty session types.
- *Decidable and flexible projection methods* based on a generic end-point generator and mergeability of branching types, enlarging the typability (§ 3.1).
- *A dependent typing system* that treats the full multiparty session types integrated with dependent types (3).
- *Properties of the dependent typing system* which include decidability of type-checking. The resulting static typing system also guarantees type-safety and deadlock-freedom (progress) for well-typed processes involved in parameterised multiparty communication protocols (§ 4).
- *Applications* featuring various process topologies (§ 2,§ 5), including the complex butterfly network of the parallel FFT algorithm (§ 2.6, § 5.5). As far as we know, this is the first time such a complex protocol is specified by a single *type* and that its implementation can be automatically type-checked to prove communication-safety and deadlock-freedom. We also extend the calculus with a new asynchronous primitive for session initialisation and apply it to Web services usecases [36] (§ 5.6).

Section 2 gives the definition of the parameterised types and processes, with their semantics. Section 3 describes the type system. The main properties of the type system are presented in Section 4. Section 5 shows typing examples. Section 6 concludes and discusses related work.

This article is a full version expanded from [42], with complete definitions and additional results with detailed proofs. It includes more examples with detailed explanations and verifications, as well as expanded related work. Some additional material related to implementations and programming examples can be found in [18].

2 Types and processes for parameterised multiparty sessions

2.1 Global types

Global types allow the description of the parameterised conversations of multiparty sessions as a type signature. Our type syntax integrates elements from three different theories: (1) global types from [3]; (2) dependent types with primitive recursive combinators

based on [31]; and (3) parameterised dependent types from a simplified Dependent ML [1, 40].

$\mathtt{i} ::= i \mid \mathrm{n} \mid \mathtt{i} \circ\mathtt{p}\ \mathtt{i}'$	Indices	$G ::=$		Global types
$\mathtt{P} ::= \mathtt{P} \wedge \mathtt{P} \mid \mathtt{i} \leq \mathtt{i}'$	Propositions		$\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G$	Message
$I ::= \mathsf{nat} \mid \{i : I \mid \mathtt{P}\}$	Index sorts		$\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K}$	Branching
$\mathcal{P} ::= \mathtt{Alice} \mid \mathtt{Worker} \mid \ldots$	Participants		$\mu \mathbf{x} . G$	Recursion
$\mathtt{p} ::= \mathtt{p}[\mathtt{i}] \mid \mathcal{P}$	Principals		$\mathbf{R}\ G\ \lambda i : I . \lambda \mathbf{x} . G'$	Primitive recursion
$S ::= \mathsf{nat} \mid \langle G \rangle$	Value type		$\mathbf{x}$	Type variable
$U ::= S \mid T$	Payload type		$G\ \mathtt{i}$	Application
$\mathrm{K} ::= \{\mathrm{n}_0, ..., \mathrm{n}_k\}$	Finite integer set		end	Null

Fig. 1. Global types

$$\begin{array}{ll} \mathbf{R}\ G\ \lambda i : I . \lambda \mathbf{x} . G'\ 0 & \longrightarrow G \\ \mathbf{R}\ G\ \lambda i : I . \lambda \mathbf{x} . G'\ (\mathrm{n}{+}1) & \longrightarrow G'\{\mathrm{n}/i\}\{\mathbf{R}\ G\ \lambda i : I . \lambda \mathbf{x} . G'\ \mathrm{n}/\mathbf{x}\} \end{array}$$

Fig. 2. Global type reduction

The grammar of global types ($G, G', ...$) is given in Figure 1. *Parameterised principals* $\mathtt{p}, \mathtt{p}', \mathtt{q}, ...$ can be indexed by one or more parameters, e.g. $\mathtt{Worker}[5][i+1]$. Index $\mathtt{i}$ ranges over index variables i, j, n, naturals n or arithmetic operations. A global interaction can be a message exchange ($\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G$), where $\mathtt{p}, \mathtt{p}'$ 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 type nat and session channel types $\langle G \rangle$), or *end-point types* T (which correspond to the behaviour of one of the session participants and will be explained in § 3) for delegation. Branching ($\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K}$) allows the session to follow one of the different G_k paths in the interaction (K is a ground and finite set of integers). $\mu \mathbf{x} . G$ is a recursive type where type variable $\mathbf{x}$ is guarded in the standard way (they only appear under some prefix) [35].

The main novelty is the primitive recursive operator $\mathbf{R}\ G\ \lambda i : I . \lambda \mathbf{x} . G'$ from Gödel's System $\mathcal{T}$ [20] whose reduction semantics is given in Figure 2. Its parameters are a global type G, an index variable i with range I, a type variable for recursion $\mathbf{x}$ and a recursion body G'.[1] When applied to an index $\mathtt{i}$, its semantics corresponds to the repetition $\mathtt{i}$-times of the body G', with the index variable i value going down by one at each iteration, from $\mathtt{i}-1$ to 0. The final behaviour is given by G when the index reaches 0. The index sorts comprise the set of natural numbers and its restrictions by predicates ($\mathtt{P}, \mathtt{P}', ..$) that are, in our case, conjunctions of inequalities. $\circ\mathtt{p}$ represents first-order indices operators (such as $+$, $-$, $*$,...). We often omit I and end in our examples.

[1] We distinguish recursion and primitive recursion in order to get decidability results, see § 4.1.

Using $\mathbf{R}$, we define the product, composition, repetition and test operators as syntactic sugar (seen in § 1):

$$\begin{array}{l|l} \Pi i.G \;= \mathbf{R}\ \mathsf{end}\ \lambda i.\lambda \mathbf{x}.G\{i+1/i\} & \mathtt{foreach}(i<j)\{G\} = \mathbf{R}\ \mathsf{end}\ \lambda i.\lambda \mathbf{x}.G\{\mathbf{x}/\mathsf{end}\}\ j \\ G_1; G_2 = \mathbf{R}\ G_2\ \lambda i.\lambda \mathbf{x}.G_1\{\mathbf{x}/\mathsf{end}\}\ 1 & \mathsf{if}\ j\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2 = \mathbf{R}\ G_2\ \lambda i.\lambda \mathbf{x}.G_1\ j \end{array}$$

where we assume that $\mathbf{x}$ is not free in G and G_1, and that the leaves of the syntax trees of G_1 and G are end. These definitions rely on a special substitution of each end by $\mathbf{x}$ (for example, $\mathtt{p} \to \mathtt{p}'\{l_1{:}!\langle \mathsf{nat}\rangle; \mathsf{end}, l_2{:}\,\mathsf{end}\}\{\mathbf{x}/\mathsf{end}\} = \mathtt{p} \to \mathtt{p}'\{l_1{:}!\langle \mathsf{nat}\rangle; \mathbf{x}, l_2{:}\,\mathbf{x}\}$). The composition operator (which we usually write ';') appends the execution of G_2 to G_1; the repetition operator above repeats G j-times[2]; the boolean values are integers 0 (false) and 1 (true). Similar syntactic sugar is defined for local types and processes.

Note that composition and repetition do not necessarily impose sequentiality: only the order of the asynchronous messages and the possible dependencies [23] between receivers and subsequent senders controls the sequentiality. For example, a parallel version of the sequence example of (§ 1 (2)) can be written in our syntax as follows:

$$\Pi n.(\mathtt{foreach}(i < n)\{\mathtt{W}[n-i] \to \mathtt{W}[n-i-1]\colon\langle \mathsf{nat}\rangle\}) \tag{3}$$

where each worker $\mathtt{W}[j]$ sends asynchronously a value v_j to its next worker $\mathtt{W}[j-1]$ without waiting for the message from $\mathtt{W}[j+1]$ to arrive first (i.e. each choice of v_j is independent from the others).

2.2 Examples of parameterised global types

We present some examples of global types that implement some communication patterns specific to typical network topologies found in classical parallel algorithms textbooks [27].

Ring - Figure 3(a) The ring pattern consists of $n+1$ workers (named $\mathtt{W}[0], \mathtt{W}[1], \ldots, \mathtt{W}[n]$) that each talks to its two neighbours: the worker $\mathtt{W}[i]$ communicates with the worker $\mathtt{W}[i-1]$ and $\mathtt{W}[i+1]$ ($1 \leq i \leq n-1$), with the exception of $\mathtt{W}[0]$ and $\mathtt{W}[n]$ who share a direct link. The type specifies that the first message is sent by $\mathtt{W}[0]$ to $\mathtt{W}[1]$, and the last one is sent from $\mathtt{W}[n]$ back to $\mathtt{W}[0]$. To ensure the presence of all three roles in the workers of this topology, the parameter domain is set to $n \geq 2$.

Multicast - Figure 3(b) The multicast session consists of `Alice` sending a message to n workers `W`. The first message is thus sent from `Alice` to $\mathtt{W}[0]$, then to $\mathtt{W}[1]$, until $\mathtt{W}[n-1]$. Note that, while the index i bound by the iteration $\mathtt{foreach}(i < n)\{\mathtt{Alice} \to \mathtt{W}[n-1-i]\colon\langle \mathsf{nat}\rangle\}$ decreases from $n-1$ to 0, the index $\mathrm{n}-1-i$ in $\mathtt{W}[n-1-i]$ increases from 0 to $n-1$.

[2] This version of `foreach` uses decreasing indices. One can write an increasing version, see § 2.2.

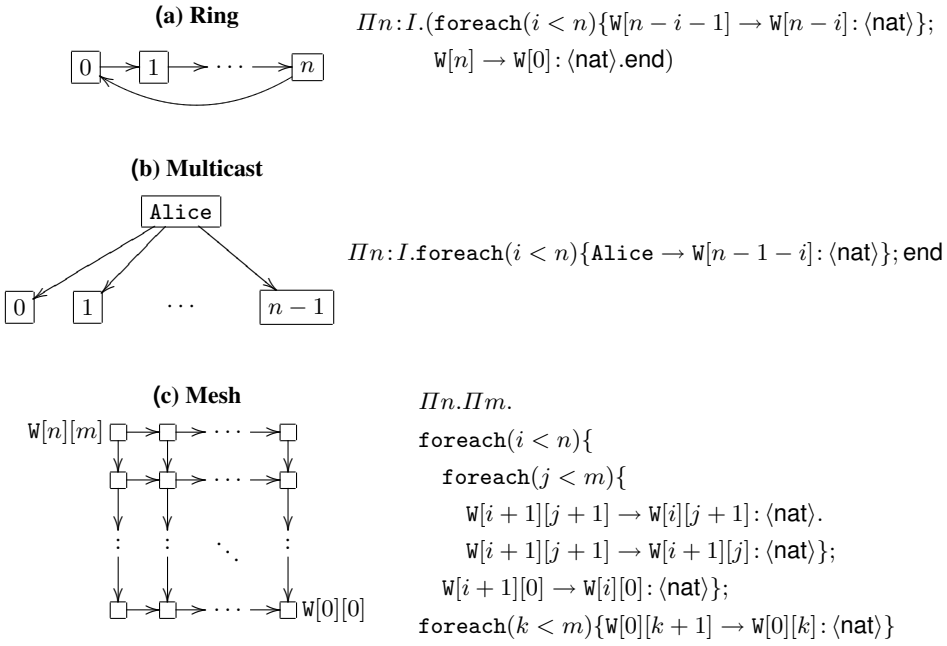


Fig. 3. Parameterised multiparty protocol on a mesh topology

Mesh - Figure 3(c) The session presented in Figure 3(c) describes a particular protocol over a standard mesh topology [27]. In this two dimensional array of workers W, each worker receives messages from his left and top neighbours (if they exist) before sending messages to his right and bottom (if they exist). 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 $\mathtt{W}[i+1][j+1] \to \mathtt{W}[i][j+1] : \langle \mathsf{nat} \rangle$ and $\mathtt{W}[i+1][j+1] \to \mathtt{W}[i+1][j] : \langle \mathsf{nat} \rangle$ for all i and j. The communication flow goes from the top-left worker $\mathtt{W}[n][m]$ and converges towards the bottom-right worker $\mathtt{W}[0][0]$ in $n+m$ steps of asynchronous message exchanges.

2.3 Process syntax

The syntax of expressions and processes is given in Figure 4, extended from [3], adding the primitive recursion operator and a new request process. Identifiers u can be variables x or channel names a. Values v are either channels a or natural numbers n. Expressions e are built out of indices $\mathtt{i}$, values v, variables x, session end points (for delegation) and operations over expressions. Participants $\mathtt{p}$ can include the indices which are substituted by values and evaluated during reductions (see the next subsection). In processes, sessions are asynchronously initiated by $\bar{u}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y).P$. It spawns, for each of the $\{\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}\}$, a request that is accepted by the participant through $u[\mathtt{p}](y).P$. Messages are sent by $c!\langle \mathtt{p}, e\rangle; P$ to the participant $\mathtt{p}$ and received by $c?(\mathtt{q}, x); P$ from the participant $\mathtt{q}$. Selection $c \oplus \langle \mathtt{p}, l\rangle; P$, and branching $c\&(\mathtt{q}, \{l_k : P_k\}_{k\in K})$, allow a participant to choose a branch from those supported by another. Standard language constructs in-

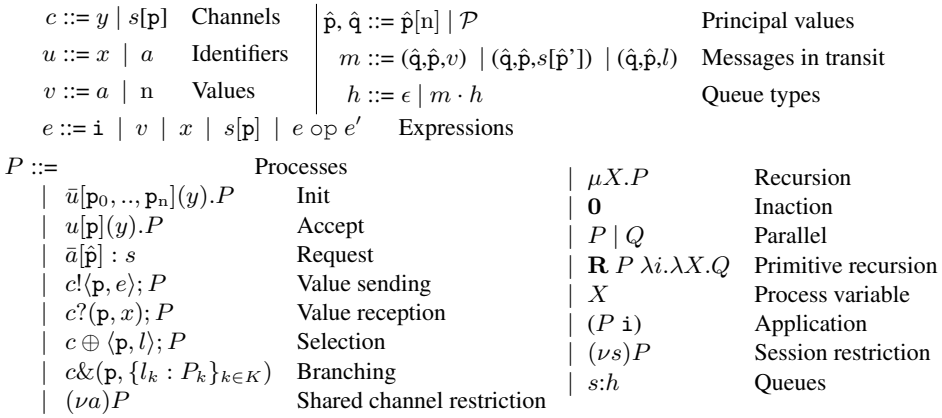


Fig. 4. Syntax for user-defined and run-time processes

clude recursive processes $\mu X.P$, restriction $(\nu a)P$ and $(\nu s)P$, and parallel composition $P \mid Q$. The primitive recursion operator $\mathbf{R}\ P\ \lambda i.\lambda X.Q$ takes as parameters a process P, a function taking an index parameter i and a recursion variable X. A queue $s : h$ stores the asynchronous messages in transit.

An *annotated* P is the result of annotating P's bound names and variables by their types or ranges as in e.g. $(\nu a : \langle G\rangle)Q$ or $s?(\mathtt{p}, x : U); Q$ or $\mathbf{R}\ Q\ \lambda i : I.\lambda X.Q'$. We omit the annotations unless needed. We often omit $\mathbf{0}$ and the participant $\mathtt{p}$ from the session primitives. Requests, session restriction and channel queues appear only at runtime, as explained below.

2.4 Semantics

The semantics is defined by the reduction relation $\longrightarrow$ presented in Figure 5. The standard definition of evaluation contexts (that allow e.g. $\mathtt{W}[3+1]$ to be reduced to $\mathtt{W}[4]$) is in Figure 6. The metavariables $\hat{\mathtt{p}}, \hat{\mathtt{q}}, ..$ range over principal values (where all indices have been evaluated). Rules [ZeroR] and [SuccR] are standard and identical to their global type counterparts. Rule [Init] describes the initialisation of a session by its first participant $\bar{a}[\mathtt{p}_0,..,\mathtt{p}_n](y_0).P_0$. It spawns asynchronous requests $\bar{a}[\hat{\mathtt{p}}_k] : s$ that allow delayed acceptance by the other session participants (rule [Join]). After the connection, the participants share the private session name s, and the queue associated to s (which is initially empty by rule [Init]). The variables $y_{\mathtt{p}}$ in each participant $\mathtt{p}$ are then replaced with the corresponding session channel, $s[\mathtt{p}]$. An equivalent, but symmetric, version of [Init] (where any participant can start the session, not only $\mathtt{p}_0$) can be also used. Rule [Init] would then be replaced by the following:

$$\bar{a}[\hat{\mathtt{p}}_0,..,\hat{\mathtt{p}}_n] \longrightarrow (\nu s)(s : \epsilon \mid \bar{a}[\hat{\mathtt{p}}_0] : s \mid ... \mid \bar{a}[\hat{\mathtt{p}}_n] : s)$$

The rest of the session reductions are standard [3, 23]. The output rules [Send] and [Label] push values, channels and labels into the queue of the session s. Rules [Recv] and [Branch] perform the complementary operations. Note that these operations check that

$$\mathbf{R}\ P\ \lambda i.\lambda X.Q\ 0 \longrightarrow P \qquad \text{[ZeroR]}$$

$$\mathbf{R}\ P\ \lambda i.\lambda X.Q\ \mathrm{n}+1 \longrightarrow Q\{\mathrm{n}/i\}\{\mathbf{R}\ P\ \lambda i.\lambda X.Q\ \mathrm{n}/X\} \qquad \text{[SuccR]}$$

$$\bar{a}[\hat{\mathtt{p}}_0, .., \hat{\mathtt{p}}_\mathrm{n}](y).P \longrightarrow (\nu s)(P\{s[\hat{\mathtt{p}}_0]/y\} \mid s : \emptyset \mid \bar{a}[\hat{\mathtt{p}}_1] : s \mid ... \mid \bar{a}[\hat{\mathtt{p}}_\mathrm{n}] : s) \qquad \text{[Init]}$$

$$\bar{a}[\hat{\mathtt{p}}_k] : s \mid a[\hat{\mathtt{p}}_k](y_k).P_k \longrightarrow P_k\{s[\hat{\mathtt{p}}_k]/y_k\} \qquad \text{[Join]}$$

$$s[\hat{\mathtt{p}}]!\langle \hat{\mathtt{q}}, v\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{\mathtt{p}}, \hat{\mathtt{q}}, v) \qquad \text{[Send]}$$

$$s[\hat{\mathtt{p}}] \oplus \langle \hat{\mathtt{q}}, l\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{\mathtt{p}}, \hat{\mathtt{q}}, l) \qquad \text{[Label]}$$

$$s[\hat{\mathtt{p}}]?(\hat{\mathtt{q}}, x); P \mid s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, v) \cdot h \longrightarrow P\{v/x\} \mid s : h \qquad \text{[Recv]}$$

$$s[\hat{\mathtt{p}}]\&(\hat{\mathtt{q}}, \{l_k : P_k\}_{k\in K}) \mid s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, l_{k_0}) \cdot h \longrightarrow P_{k_0} \mid s : h \quad (k_0 \in K) \qquad \text{[Branch]}$$

$$P \longrightarrow P' \quad \Rightarrow \quad P\ e \longrightarrow P'\ e \qquad P \longrightarrow P' \quad \Rightarrow \quad (\nu r)P \longrightarrow (\nu r)P' \qquad \text{[App,Scop]}$$

$$P \longrightarrow P' \quad \Rightarrow \quad P \mid Q \longrightarrow P' \mid Q \qquad \text{[Par]}$$

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

$$e \longrightarrow e' \quad \Rightarrow \quad \mathcal{E}[e] \longrightarrow \mathcal{E}[e'] \qquad \text{[Context]}$$

Fig. 5. Reduction rules

$\mathcal{E}[\_, \ldots, \_]$ ::=		Evaluation contexts
	$\_\ \mathtt{op}\ e \mid v\ \mathtt{op}\ \_$	Expression
	$(P\ \_)$	Application
	$\bar{a}[\hat{\mathtt{p}}_1, .., \hat{\mathtt{p}}_n, \_, \mathtt{p}_{n+1}, .., \mathtt{p}_m](y).P$	Request
	$a[\_](y).P$	Accept
	$s[\_]!\langle \mathtt{p}, e\rangle; P \mid s[\hat{\mathtt{p}}]!\langle \_, e\rangle; P \mid s[\hat{\mathtt{p}}]!\langle \hat{\mathtt{q}}, \_\rangle; P$	Send
	$s[\_] \oplus \langle \mathtt{p}, l\rangle; P \mid s[\hat{\mathtt{p}}] \oplus \langle \_, l\rangle; P$	Selection
	$s[\_]?(\mathtt{p}, x); P \mid s[\hat{\mathtt{p}}]?(\_, x); P$	Receive
	$s[\_]\&(\mathtt{p}, \{l_k : P_k\}_{k\in K}) \mid s[\hat{\mathtt{p}}]\&(\_, \{l_k : P_k\}_{k\in K})$	Branching

Fig. 6. Evaluation contexts

the sender and receiver match. Processes are considered modulo structural equivalence, denoted by $\equiv$ (in particular, we note $\mu X.P \equiv P\{\mu X.P/X\}$), whose definition is found in Figure 7. Besides the standard rules [29], we have a rule for rearranging messages when the senders or the receivers are different, and a rule for the garbage-collection of unused and empty queues.

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

$$(\nu r)\ \mathbf{0} \equiv \mathbf{0} \quad (\nu s)\ s : \emptyset \equiv \mathbf{0} \quad (\nu r)\ P \mid Q \equiv (\nu r)\ (P \mid Q) \qquad \text{if } r \notin \mathrm{fn}(Q)$$

$$s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, z) \cdot (\hat{\mathtt{q}}', \hat{\mathtt{p}}', z') \cdot h \equiv s : (\hat{\mathtt{q}}', \hat{\mathtt{p}}', z') \cdot (\hat{\mathtt{q}}, \hat{\mathtt{p}}, z) \cdot h \qquad \text{if } \hat{\mathtt{p}} \neq \hat{\mathtt{p}}' \text{ or } \hat{\mathtt{q}} \neq \hat{\mathtt{q}}'$$

$$\mu X.P \equiv P\{\mu X.P/X\}$$

r ranges over a, s. z ranges over v, $s[\hat{\mathtt{p}}]$ and l.

Fig. 7. Structural equivalence

2.5 Processes for parameterised multiparty protocols

We give here the processes corresponding to the interactions described in § 1 and § 2.1, then introduce a parallel implementation of the Fast Fourier Transform algorithm. There are various ways to implement end-point processes from a single global type, and we show one instance for each example below.

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

$$\Pi n.(\mathbf{R}\ \mathsf{end}\ \lambda i.\lambda \mathbf{x}.\mathtt{Alice} \to \mathtt{Bob}\colon\langle \mathsf{nat}\rangle.\mathtt{Bob} \to \mathtt{Carol}\colon\langle \mathsf{nat}\rangle.\mathbf{x}\ n)$$

Then `Alice` and `Bob` can be implemented with recursors as follows (we abbreviate `Alice` by `a`, `Bob` by `b` and `Carol` by `c`).

$$\begin{aligned}
&\mathtt{Alice}(n) = \bar{a}[\mathtt{a},\mathtt{b},\mathtt{c}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \mathtt{b}, e[i]\rangle; X\ n)\\
&\mathtt{Bob}(n) = a[\mathtt{b}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y?(\mathtt{a}, z); y!\langle \mathtt{c}, z\rangle; X\ n)\\
&\mathtt{Carol}(n) = a[\mathtt{c}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y?(\mathtt{b}, z); X\ n)
\end{aligned}$$

`Alice` repeatedly sends a message $e[i]$ to `Bob` n-times. Then n can be bound by λ-abstraction, allowing the user to dynamically assign the number of the repetitions.

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

Sequence from § 1 (2) The process below generates all participants using a recursor:

$$\begin{aligned}
\Pi n.(\mathsf{if}\ n = 0\ \mathsf{then}\ \ &\mathbf{0}\\
\mathsf{else}\ \ &(\mathbf{R}\ (\bar{a}[\mathtt{W}[n],..,\mathtt{W}[0]](y).y!\langle \mathtt{W}[n-1], v\rangle; \mathbf{0}\\
&\quad \mid a[\mathtt{W}[0]](y).y?(\mathtt{W}[1], z); \mathbf{0})\\
&\quad \lambda i.\lambda X.(a[\mathtt{W}[i+1]](y).y?(\mathtt{W}[i+2], z); y!\langle \mathtt{W}[i], z\rangle; \mathbf{0} \mid X) \quad n-1)
\end{aligned}$$

When $n = 0$ no message is exchanged. In the other case, the recursor creates the $n-1$ workers through the main loop and finishes by spawning the initial and final ones.

As an illustration of the semantics, we show the reduction of the above process for $n = 2$. After several applications of the [SuccR] and [ZeroR] rules, we have:

$$\bar{a}[\mathtt{W}[2], \mathtt{W}[1], \mathtt{W}[0]](y).y!\langle \mathtt{W}[1], v\rangle; \mathbf{0} \mid a[\mathtt{W}[0]](y).y?(\mathtt{W}[1], z); \mathbf{0} \mid a[\mathtt{W}[1]](y).y?(\mathtt{W}[2], z); y!\langle \mathtt{W}[0], z\rangle; \mathbf{0}$$

which, with [Init], [Join], [Send], [Recv], gives:

$$\begin{aligned}
\longrightarrow\ \ &(\nu s)(s : \epsilon \mid s[\mathtt{W}[2]]!\langle \mathtt{W}[1], v\rangle; \mathbf{0} \mid \bar{a}[\mathtt{W}[1]] : s \mid \bar{a}[\mathtt{W}[0]] : s \mid\\
&\quad a[\mathtt{W}[0]](y).y?(\mathtt{W}[1], z); \mathbf{0} \mid a[\mathtt{W}[1]](y).y?(\mathtt{W}[2], z); y!\langle \mathtt{W}[0], z\rangle; \mathbf{0})\\
\longrightarrow\ \ &(\nu s)(s : \epsilon \mid s[\mathtt{W}[2]]!\langle \mathtt{W}[1], v\rangle; \mathbf{0} \mid \bar{a}[\mathtt{W}[1]] : s \mid\\
&\quad s[\mathtt{W}[0]]?(\mathtt{W}[1], z); \mathbf{0} \mid a[\mathtt{W}[1]](y).y?(\mathtt{W}[2], z); y!\langle \mathtt{W}[0], z\rangle; \mathbf{0})\\
\longrightarrow^*\ \ &(\nu s)(s : \emptyset \mid s[\mathtt{W}[2]]!\langle \mathtt{W}[1], v\rangle; \mathbf{0} \mid s[\mathtt{W}[0]]?(\mathtt{W}[1], z); \mathbf{0} \mid s[\mathtt{W}[1]]?(\mathtt{W}[2], z); s[\mathtt{W}[1]]!\langle \mathtt{W}[0], z\rangle; \mathbf{0})\\
\longrightarrow^*\ \ &(\nu s)(s : \emptyset \mid s[\mathtt{W}[0]]?(\mathtt{W}[1], z); \mathbf{0} \mid s[\mathtt{W}[1]]!\langle \mathtt{W}[0], v\rangle; \mathbf{0})\\
\longrightarrow^*\ \ &\equiv \mathbf{0}
\end{aligned}$$

Ring - Figure 3(a) The process that generates all the roles using a recursor is as follows:

$$\begin{array}{l} \Pi n.(\mathbf{R}\ \bar{a}[\mathtt{W}[0],...,\mathtt{W}[n]](y).y!\langle \mathtt{W}[1], v\rangle; y?(\mathtt{W}[n], z); P \\ \qquad a[\mathtt{W}[n]](y).y?(\mathtt{W}[n-1], z); y!\langle \mathtt{W}[0], z\rangle; Q \\ \qquad\quad \lambda i.\lambda X.(a[\mathtt{W}[i+1]](y).y?(\mathtt{W}[i], z); y!\langle \mathtt{W}[i+2], z\rangle;\ |\ X) \quad n-1) \end{array}$$

We take the range of n to be $n \geq 2$.

Mesh - Figure 3 (c) In this example, when n and m are bigger than 2, there are 9 distinct patterns of communication.

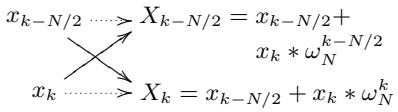
We write below these processes. We assume the existence of a function $f(z_1, z_2, i, j)$ which computes from z_1 and z_2 the value to be transmitted to $\mathtt{W}[i][j]$. We then designates the processes based on their position in the mesh. The initiator $\mathtt{W}[n][m]$ is in the top-left corner of the mesh and is implemented by $P_{\text{top-left}}$. The workers that are living in the other corners are implemented by $P_{\text{top-right}}$ for $\mathtt{W}[n][0]$, $P_{\text{bottom-left}}$ for $\mathtt{W}[0][m]$ and $P_{\text{bottom-right}}$ for the final worker $\mathtt{W}[0][0]$. The processes P_{top}, P_{left}, P_{bottom} and P_{right} respectively implement the workers from the top row, the leftmost column, the bottom row and the rightmost column. The workers that are in the central part of the mesh are played by the $P_{\text{center}}(i, j)$ processes.

$$\begin{array}{ll} P_{\text{top-left}}(z_1, z_2, n, m) & = \bar{a}[\mathtt{W}[n][m], ..., \mathtt{W}[0][0]](y).y!\langle \mathtt{W}[n-1][m], f(z_1, z_2, n-1, m)\rangle; \\ & \quad y!\langle \mathtt{W}[n][m-1], f(z_1, z_2, n, m-1)\rangle; \mathbf{0} \\ P_{\text{top-right}}(z_2, n) & = a[\mathtt{W}[n][0]](y).y?(\mathtt{W}[n][1], z_1); y!\langle \mathtt{W}[n-1][0], f(z_1, z_2, n-1, 0)\rangle; \mathbf{0} \\ P_{\text{bottom-left}}(z_1, m) & = a[\mathtt{W}[0][m]](y).y?(\mathtt{W}[1][m], z_2); y!\langle \mathtt{W}[0][m-1], f(z_1, z_2, 0, m-1)\rangle; \mathbf{0} \\ P_{\text{bottom-right}}(m) & = a[\mathtt{W}[0][0]](y).y?(\mathtt{W}[1][0], z_1); y?(\mathtt{W}[0][1], z_2); \mathbf{0} \\ P_{\text{top}}(z_2, n, k) & = a[\mathtt{W}[n][k+1]](y).y?(\mathtt{W}[n][k+2], z_1); \\ & \quad y!\langle \mathtt{W}[n-1][k+1], f(z_1, z_2, n-1, k+1)\rangle; y!\langle \mathtt{W}[n][k], f(z_1, z_2, n, k)\rangle; \mathbf{0} \\ P_{\text{bottom}}(k) & = a[\mathtt{W}[0][k+1]](y).y?(\mathtt{W}[1][k+1], z_1); y?(\mathtt{W}[0][k+2], z_2); \\ & \quad y!\langle \mathtt{W}[0][k], f(z_1, z_2, 0, k)\rangle; \mathbf{0} \\ P_{\text{left}}(z_1, m, i) & = a[\mathtt{W}[i+1][m]](y).y?(\mathtt{W}[i+2][m], z_2); y!\langle \mathtt{W}[i][m], f(z_1, z_2, i, m)\rangle; \\ & \quad y!\langle \mathtt{W}[i+1][m-1], f(z_1, z_2, i+1, m-1)\rangle; \\ P_{\text{right}}(i) & = a[\mathtt{W}[i+1][0]](y).y?(\mathtt{W}[i+2][0], z_1); y?(\mathtt{W}[i+1][1], z_2); \\ & \quad y!\langle \mathtt{W}[i][0], f(z_1, z_2, i, 0)\rangle; \mathbf{0} \\ P_{\text{center}}(i, j) & = a[\mathtt{W}[i+1][j+1]](y).y?(\mathtt{W}[i+2][j+1], z_1); y?(\mathtt{W}[i+1][j+2], z_2); \\ & \quad y!\langle \mathtt{W}[i][j+1], f(z_1, z_2, i, j+1)\rangle; y!\langle \mathtt{W}[i+1][j], f(z_1, z_2, i+1, j)\rangle; \mathbf{0} \end{array}$$

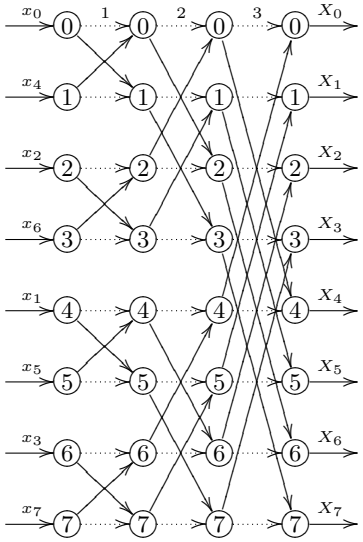
The complete implementation can be generated using the following process:

$$\begin{array}{l} \Pi n.\Pi m.(\mathbf{R}\ (\mathbf{R}\ P_{\text{top-left}}(z_1, z_2, n, m)|P_{\text{bottom-right}}(m)|P_{\text{top-right}}(z_2, n)|P_{\text{bottom-left}}(z_1, m)) \\ \qquad \lambda k.\lambda Z.(P_{\text{top}}(z_2, n, k)|P_{\text{bottom}}(k)|Z) \\ \qquad m-1) \\ \quad \lambda i.\lambda X.(\mathbf{R}\ P_{\text{left}}(z_1, m, i)|P_{\text{right}}(i)|X \\ \qquad\quad \lambda j.\lambda Y.(P_{\text{center}}(i, j)|Y) \\ \qquad\quad m-1) \\ \quad n-1) \end{array}$$

(a) Butterfly pattern



(b) FFT diagram



(c) Global type $G =$

$$
\begin{array}{l}
\Pi n. \\
\texttt{foreach}(i < 2^n)\{i \to i : \langle \mathsf{nat} \rangle\}; \\
\texttt{foreach}(l < n)\{ \\
\ \texttt{foreach}(i < 2^l)\{ \\
\ \ \texttt{foreach}(j < 2^{n-l-1})\{ \\
\ \ \ \texttt{foreach}(k < 2)\{ \\
\ \ \ \ \texttt{foreach}(k' < 2)\{ \\
\ \ \ \ \ i * 2^{n-l} + k * 2^{n-l-1} + j \\
\ \ \ \ \ \to i * 2^{n-l} + k' * 2^{n-l-1} + j : \langle \mathsf{nat} \rangle \}\}\}\}\}
\end{array}
$$

(d) Processes $P(n, \mathtt{p}, x_{\bar{\mathtt{p}}}, y, r_{\mathtt{p}}) =$

$$
\begin{array}{l}
y!\langle \mathtt{p}, x_{\bar{\mathtt{p}}} \rangle; \\
\texttt{foreach}(l < n)\{ \\
\ \mathsf{if}\ \mathrm{bit}_{n-l}(\mathtt{p}) = 0 \\
\ \mathsf{then}\ y?(\mathtt{p}, x); y!\langle \mathtt{p} + 2^{n-l-1}, x \rangle; \\
\qquad y?(\mathtt{p} + 2^{n-l-1}, z); y!\langle \mathtt{p}, x + z\,\omega_N^{g(l,\mathtt{p})} \rangle; \\
\ \mathsf{else}\ y?(\mathtt{p}, x); y!\langle \mathtt{p} - 2^{n-l-1}, x \rangle; \\
\qquad y?(\mathtt{p} - 2^{n-l-1}, z); y!\langle \mathtt{p}, z + x\,\omega_N^{g(l,\mathtt{p})} \rangle; \}; \\
y?(\mathtt{p}, x); r_{\mathtt{p}}!\langle 0, x \rangle; \mathbf{0}
\end{array}
$$

where $g(l, \mathtt{p}) = \mathtt{p} \mod 2^l$

Fig. 8. Fast Fourier Transform on a butterfly network topology

2.6 Fast Fourier Transform

We describe a parallel implementation of the Fast Fourier Transform algorithm (more precisely the radix-2 variant of the Cooley-Tukey algorithm [14]). 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 $\vec{x} = x_0, \ldots, x_{N-1}$ 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 $\vec{X} = X_0, \ldots, X_{N-1}$ defined by:

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

with $\omega_N^k = e^{\imath \frac{2k\pi}{N}}$ one of the n-th primitive roots 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 the radix-2 variant of the Cooley-Tukey algorithm [14]. uses a divide-and-conquer strategy based on the following equation (we use the fact that $\omega_N^{2k} = \omega_{N/2}^k$):

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

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 8(a) illustrates this recursive principle, called *butterfly*, where two different intermediary values 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 8(b). It features the application of the FFT on a network of $N = 2^3$ machines on an hypercube network computing the discrete Fourier transform of vector $x_0, \ldots, x_7$. 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. 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 $\mathtt{p}$ initially receives $x_{\overline{\mathtt{p}}}$ where $\overline{\mathtt{p}}$ denotes the bit-reversal of $\mathtt{p}$.

Global Types Figure 8(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 $\mathbf{R}\ (\ldots)\ \lambda k.\lambda \mathbf{u}.k \to k : \langle \mathsf{nat} \rangle.\mathbf{u}$ concerns the initialisation: each of the machines sends the $x_{\mathtt{p}}$ 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). The 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 8(d). When $\mathtt{p}$ is the machine number, $x_{\overline{\mathtt{p}}}$ the initial value, and y the session channel, the machine starts by sending $x_{\overline{\mathtt{p}}}$ to itself: $y!\langle x_{\overline{\mathtt{p}}} \rangle;$. 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 $\mathrm{bit}_{n-l}(\mathtt{p})$). In the two branches, we receive the previously computed value $y?(x);..$, then we send to and receive from the other machine (of number $\mathtt{p}+2^{n-l-1}$ or $\mathtt{p}-2^{n-l-1}$, i.e. whose $(n-l)$th bit was flipped). We finally compute the new value and send it to ourselves: respectively by $y!\langle x + z\,\omega_N^{g(l,\mathtt{p})}\rangle; X$ or $y!\langle z + x\,\omega_N^{g(l,\mathtt{p})}\rangle; X$. 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. At the end of the algorithm, the calculated values are sent to some external channels: $r_{\mathtt{p}}!\langle 0, x\rangle$.

3 Typing parameterised multiparty interactions

This section introduces the type system, by which we can statically type parameterised global specifications.

3.1 End-point types and end-point projections

T ::=		End-point types	$\mid \mu\mathbf{x}.T$		Recursion
$\mid$	$!\langle \mathtt{p}, U\rangle; T$	Output	$\mid \mathbf{R}\ T\ \lambda i{:}I.\lambda\mathbf{x}.T'$		Primitive recursion
$\mid$	$?\langle \mathtt{p}, U\rangle; T$	Input	$\mid \mathbf{x}$		Type variable
$\mid$	$\oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle$	Selection	$\mid T\ \mathtt{i}$		Application
$\mid$	$\&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle$	Branching	$\mid$	end	End

Fig. 9. End-point types

A global type is projected to an *end-point type* according to each participant's viewpoint. The syntax of end-point types is given in Figure 9. Output expresses the sending to $\mathtt{p}$ of a value or channel of type U, followed by the interactions T. Selection represents the transmission to $\mathtt{p}$ of a label l_k chosen in $\{l_k\}_{k\in K}$ followed by T_k. Input and branching are their dual counterparts. The other types are similar to their global versions.

End-point projection: a generic projection The relation between end-point types and global types is formalised by the projection relation. Since the actual participant characteristics might only be determined at runtime, we cannot straightforwardly use the definition from [3, 23]. Instead, we rely on the expressive power of the primitive recursive operator: *a generic end-point projection of G onto* $\mathtt{q}$, written $G \upharpoonright \mathtt{q}$, represents the family of all the possible end-point types that a principal $\mathtt{q}$ can satisfy at run-time.

The general endpoint generator is defined in Figure 10 using the derived construct $\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_$. The projection $\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G \upharpoonright \mathtt{q}$ leads to a case analysis: if the participant $\mathtt{q}$ is equal to $\mathtt{p}$, then the end-point type of $\mathtt{q}$ is an output of type U to $\mathtt{p}'$; if participant $\mathtt{q}$ is $\mathtt{p}'$ then $\mathtt{q}$ inputs U from $\mathtt{p}'$; else we skip the prefix. The first 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. For

$$
\begin{array}{rl}
\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G \upharpoonright \mathtt{q} = & \mathsf{if}\ \mathtt{q}{=}\mathtt{p}{=}\mathtt{p}'\ \mathsf{then}\ !\langle \mathtt{p}, U\rangle; ?\langle \mathtt{p}, U\rangle; G \upharpoonright \mathtt{q} \\
& \mathsf{else\ if}\ \mathtt{q}{=}\mathtt{p}\ \mathsf{then}\ !\langle \mathtt{p}', U\rangle; G \upharpoonright \mathtt{q} \\
& \mathsf{else\ if}\ \mathtt{q}{=}\mathtt{p}'\ \mathsf{then}\ ?\langle \mathtt{p}, U\rangle; G \upharpoonright \mathtt{q} \\
& \mathsf{else}\ G \upharpoonright \mathtt{q} \\
\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K} \upharpoonright \mathtt{q} = & \mathsf{if}\ \mathtt{q}{=}\mathtt{p}\ \mathsf{then} \oplus \langle \mathtt{p}', \{l_k : G_k \upharpoonright \mathtt{q}\}_{k \in K}\rangle \\
& \mathsf{else\ if}\ \mathtt{q}{=}\mathtt{p}'\ \mathsf{then}\ \& \langle \mathtt{p}, \{l_k : G_k \upharpoonright \mathtt{q}\}_{k \in K}\rangle \\
& \mathsf{else} \sqcup_{k \in K} G_k \upharpoonright \mathtt{q} \\
\mathbf{R}\ G\ \lambda i{:}I.\lambda \mathbf{x}.G' \upharpoonright \mathtt{q} = & \mathbf{R}\ G \upharpoonright \mathtt{q}\ \lambda i{:}I.\lambda \mathbf{x}.G' \upharpoonright \mathtt{q} \\
(\mu \mathbf{x}.G) \upharpoonright \mathtt{p} = & \mu \mathbf{x}.G \upharpoonright \mathtt{p} \\
\mathbf{x} \upharpoonright \mathtt{p} = & \mathbf{x} \\
(G\ \mathtt{i}) \upharpoonright \mathtt{p} = & (G \upharpoonright \mathtt{p})\ \mathtt{i} \\
\mathsf{end} \upharpoonright \mathtt{p} = & \mathsf{end}
\end{array}
$$

Fig. 10. Projection of global types to end-point types

the other cases (as well as for our derived operators), the projection is homomorphic. We also identify $\mu \mathbf{x}.\mathbf{x}$ as end ($\mu \mathbf{x}.\mathbf{x}$ is generated when a target participant is not included under the recursion, for example, $\mathtt{p} \to \mathtt{p}' : \langle U \rangle . \mu \mathbf{x}.\mathtt{q} \to \mathtt{q}' : \langle U \rangle . \mathbf{x} \upharpoonright \mathtt{p} = !\langle \mathtt{p}, U\rangle; \mu \mathbf{x}.\mathbf{x}$) and $\mu \mathbf{x}.T$ as T if $\mathbf{x} \notin \mathsf{ftv}(T)$.

Mergeability of branching types We first recall the example from [23], which explains that naïve branching projection leads to inconsistent end-point types.

$$\mathtt{W}[0] \to \mathtt{W}[1] : \{\mathsf{ok} : \mathtt{W}[1] \to \mathtt{W}[2] : \langle \mathsf{bool} \rangle,\ \ \mathsf{quit} : \mathtt{W}[1] \to \mathtt{W}[2] : \langle \mathsf{nat} \rangle\}$$

We cannot project the above type onto $\mathtt{W}[2]$ because, while the branches behave differently, $\mathtt{W}[0]$ makes a choice without informing $\mathtt{W}[2]$ who thus cannot know the type of the expected value. A solution is to define projection only when the branches are identical, i.e. we change the above nat to bool in our example above.

In our framework, this restriction is too strong since each branch may contain different parametric interaction patterns. To overcome this, below we propose a method called *mergeability* of branching types.[3]

Definition 3.1 (Mergeability) The mergeability relation $\bowtie$ is the smallest congruence relation over end-point types such that:

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

When $T_1 \bowtie T_2$ is defined, we define the operation $\sqcup$ as a partial commutative operator over two types such that $T \sqcup T = T$ for all types and that:

$$\begin{array}{l}
\&\langle \mathtt{p}, \{l_k : T_k\}_{k \in K}\rangle \sqcup \&\langle \mathtt{p}, \{l_j : T'_j\}_{j \in J}\rangle\ = \\
\quad \&\langle \mathtt{p}, \{l_k : T_k \sqcup T'_k\}_{k \in K \cap J} \cup \{l_k : T_k\}_{k \in K \setminus J} \cup \{l_j : T'_j\}_{j \in J \setminus K}\rangle
\end{array}$$

and homomorphic for other types (i.e. $\mathcal{C}[T_1] \sqcup \mathcal{C}[T_2] = \mathcal{C}[T_1 \sqcup T_2]$ where $\mathcal{C}$ is a context for local types).

[3] The idea of mergeability is introduced informally in the tutorial paper [12].

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

$$\mathtt{W}[0] \to \mathtt{W}[1] : \{\mathsf{ok} : \mathtt{W}[1] \to \mathtt{W}[2] : \{\mathsf{ok} : \mathtt{W}[1] \to \mathtt{W}[2]\langle \mathsf{bool} \rangle\ \},\ \mathsf{quit} : \mathtt{W}[1] \to \mathtt{W}[2] : \{\mathsf{quit} : \mathtt{W}[1] \to \mathtt{W}[2]\langle \mathsf{nat} \rangle\}\}\}$$

Then $\mathtt{W}[2]$ can have the type $\&\langle \mathtt{W}[1],\ \{\mathsf{ok} : \langle \mathtt{W}[1], \mathsf{bool}\rangle,\ \mathsf{quit} : \langle \mathtt{W}[1], \mathsf{nat}\rangle\}\rangle$ which could not be obtained through the original projection rule in [3, 23]. This projection is sound up to branching subtyping (it will be proved in Lemma 4.7 later).

3.2 Type system (1): environments, judgements and kinding

This subsection introduces the environments and kinding systems. 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.

Environments One of the main differences with previous session type systems is that session environments Δ can contain dependent *process types*. The grammar of environments, process types and kinds are given below.

$$\Delta ::= \emptyset \mid \Delta, c{:}T \qquad \Gamma ::= \emptyset \mid \Gamma, \mathtt{P} \mid \Gamma, u : S \mid \Gamma, i : I \mid \Gamma, X : \tau \qquad \tau ::= \Delta \mid \Pi i{:}I.\tau$$

Δ is the *session environment* which associates channels to session types. Γ is the *standard environment* which contains predicates and which associates variables to sort types, service names to global types, indices to index sets and process variables to session types. τ is a *process type* which is either a session environment or a dependent type. We write $\Gamma, u : S$ only if $u \notin dom(\Gamma)$ where $dom(\Gamma)$ denotes the domain of Γ. We use the same convention for others.

$\Gamma \vdash \mathsf{Env}$	well-formed environments	$\Gamma \vdash \alpha \approx \beta$	type isomorphism
$\Gamma \vdash \kappa$	well-formed kindings	$\Gamma \vdash e \triangleright U$	expression
$\Gamma \vdash \alpha \blacktriangleright \kappa$	well-formed types	$\Gamma \vdash \mathtt{p} \triangleright U_p$	participant
$\Gamma \vdash \alpha \equiv \beta$	type equivalence	$\Gamma \vdash P \triangleright \tau$	processes

Fig. 11. Judgements ($\alpha, \beta, ...$ range over any types)

Judgements Our type system uses the judgements listed in Figure 11.

Following [40], we assume given in the typing rules two semantically defined judgements: $\Gamma \models \mathtt{P}$ (predicate $\mathtt{P}$ is a consequence of Γ) and $\Gamma \models \mathtt{i} : I$ ($\mathtt{i} : I$ follows from the assumptions of Γ).

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 $\Gamma \vdash_{\{s\}} s : h \triangleright \Delta$ and one for runtime processes $\Gamma \vdash_{\Sigma} P \triangleright \Delta$) which are similar with those in [3] and listed in the Appendix. We often omit Σ from $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ if it is not important.

Kinding The definition of kinds is given below:

$$\kappa ::= \Pi j : I.\kappa \mid \mathsf{Type} \qquad U_p ::= \mathsf{nat} \mid \Pi i{:}I.U_p$$

We inductively define well-formed types using a kind system [18]. The judgement $\Gamma \vdash \alpha \blacktriangleright \kappa$ means type U has kind κ. Kinds include proper types for global, value, principal, end-point and process types (denoted by Type), and the kind of type families, written by $\Pi i{:}I.\kappa$. The kinding rules are defined in Figure 14 and Figure 13 in this section and Figure 20 in the Appendix. The environment well-formedness rules are in Figure 12.

The kinding rules for types, value types, principals, index sets and process types are listed in Figure 13. In $\lfloor$KMAR$\rfloor$ in the value types, $\mathsf{ftv}(G)$ denotes a set of free type variables in G. The condition $\mathsf{ftv}(G) = \emptyset$ means that shared channel types are always closed. Rule $\lfloor$KINDEX$\rfloor$ forms the index sort which contains only natural number (by the condition $0 \leq i$). Other rules in Figure 13 and the rules in Figure 12 are standard.

We next explain the global type kinding rules from Figure 14. The local type kinding in Figure 20 in Appendix is similar.

Rule $\lfloor$KIO$\rfloor$ states that if both participants have 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 $\lfloor$KBRA$\rfloor$ is similar, while rules $\lfloor$KREC,KTVAR$\rfloor$ are standard.

Dependent types are introduced when kinding recursors in $\lfloor$KRCR$\rfloor$. In $\lfloor$KRCR$\rfloor$, we need an updated index range for i in the premise $\Gamma, i : I^- \vdash G' \blacktriangleright \mathsf{Type}$ since the index substitution uses the predecessor of i. We define I^- using the abbreviation $[0..\mathtt{j}] = \{i{:}\mathsf{nat} \mid i \leq \mathtt{j}\}$:

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

Note that the second argument ($\lambda i{:}I^-.\lambda \mathbf{x}.G'$) is closed (i.e. it does not contain free type variables). We use $\lfloor$KAPP$\rfloor$ for both index applications. Note that $\lfloor$KAPP$\rfloor$ checks wether the argument $\mathtt{i}$ satisfies the index set I. Other rules are similarly understood including those for process types (noting Δ is a well-formed environment if it only contains types T of kind Type).

$$\frac{-}{\emptyset \vdash \mathsf{Env}}\ \lfloor\text{ENUL}\rfloor \qquad \frac{\Gamma \models \mathtt{P}}{\Gamma, \mathtt{P} \vdash \mathsf{Env}}\ \lfloor\text{EPRE}\rfloor \qquad \frac{\Gamma \vdash S \blacktriangleright \mathsf{Type} \quad u \notin dom(\Gamma)}{\Gamma, u{:}S \vdash \mathsf{Env}}\ \lfloor\text{ESORT}\rfloor$$

$$\frac{\Gamma \vdash I \quad i \notin dom(\Gamma)}{\Gamma, i : I \vdash \mathsf{Env}}\ \lfloor\text{EINDEX}\rfloor \qquad \frac{\Gamma \vdash \tau \blacktriangleright \kappa \quad X \notin dom(\Gamma)}{\Gamma, X : \tau \vdash \mathsf{Env}}\ \lfloor\text{VENV}\rfloor$$

Fig. 12. Well-formed environments

3.3 Type system (2): type equivalence

Since our types include dependent types and recursors, we need a notion of type equivalence. We extend the standard method of [1, §2] with the recursor. The rules are found

Type

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{Type}}\ \lfloor \textsc{KBase} \rfloor \qquad \frac{\Gamma, i{:}I \vdash \kappa}{\Gamma \vdash \Pi i{:}I.\kappa}\ \lfloor \textsc{KSeq} \rfloor$$

Value Types

$$\frac{\Gamma \vdash G \blacktriangleright \mathsf{Type} \quad \mathsf{ftv}(G) = \emptyset}{\Gamma \vdash \langle G \rangle \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KMar} \rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat} \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KNat} \rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{bool} \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KBool} \rfloor$$

Principals

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat} \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KPNat} \rfloor \qquad \frac{\Gamma, i{:}I \vdash U_p \blacktriangleright \kappa}{\Gamma \vdash U_p \blacktriangleright \Pi i{:}I.\kappa}\ \lfloor \textsc{KProd} \rfloor$$

Index Sets

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat}}\ \lfloor \textsc{KINat} \rfloor \qquad \frac{\Gamma, i{:}I \models \mathtt{P} \wedge 0 \leq i}{\Gamma \vdash \{i{:}I \mid \mathtt{P} \wedge 0 \leq i\}}\ \lfloor \textsc{KIIndex} \rfloor$$

Process Types

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \emptyset \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KPNul} \rfloor \qquad \frac{\Gamma \vdash \Delta \blacktriangleright \mathsf{Type} \quad \Gamma \vdash T \blacktriangleright \mathsf{Type}}{\Gamma \vdash \Delta, c : T \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KPCh} \rfloor \qquad \frac{\Gamma, i : I \vdash \tau \blacktriangleright \kappa}{\Gamma \vdash \Pi i{:}I.\tau \blacktriangleright \Pi i{:}I.\kappa}\ \lfloor \textsc{KPProd} \rfloor$$

Fig. 13. Kinding system for types, values, principals, index sets and process types

$$\frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \Gamma \vdash \mathtt{p}' \rhd \mathsf{nat} \quad \Gamma \vdash G' \blacktriangleright \mathsf{Type} \quad \Gamma \vdash U \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mathtt{p} \to \mathtt{p}' : \langle U \rangle.G' \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KIO} \rfloor$$

$$\frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat}, \Gamma \vdash \mathtt{p}' \rhd \mathsf{nat} \quad \forall k \in K,\ \Gamma \vdash G_k \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K} \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KBra} \rfloor$$

$$\frac{\Gamma \vdash G \blacktriangleright \kappa\{0/j\} \quad \Gamma, i : I^- \vdash G' \blacktriangleright \kappa\{i+1/j\}}{\Gamma \vdash \mathbf{R}\, G\, \lambda i{:}I^-.\lambda \mathbf{x}.G' \blacktriangleright \Pi j{:}I.\kappa}\ \lfloor \textsc{KRcr} \rfloor$$

$$\frac{\Gamma \vdash G \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mu \mathbf{x}.G \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KRec} \rfloor \qquad \frac{\Gamma \vdash \kappa}{\Gamma \vdash \mathbf{x} \blacktriangleright \kappa}\ \lfloor \textsc{KVar} \rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \blacktriangleright \mathsf{Type}}\ \lfloor \textsc{KEnd} \rfloor$$

$$\frac{\Gamma \vdash G \blacktriangleright \Pi i{:}I.\kappa \quad \Gamma \models \mathtt{i} : I}{\Gamma \vdash G\ \mathtt{i} \blacktriangleright \kappa\{\mathtt{i}/i\}}\ \lfloor \textsc{KApp} \rfloor$$

Fig. 14. Kinding rules for global types

in Figure 15 and applied following the order appeared in Figure 15. For example, $\lfloor$WFBASE$\rfloor$ has a higher priority than $\lfloor$WFAPP$\rfloor$, and $\lfloor$WFREC$\rfloor$ has a higher priority than $\lfloor$WFRECF$\rfloor$. We only define the rules for G. The same set of rules can be applied to T and τ.

Rule $\lfloor$WFBASE$\rfloor$ is the main rule defining $G_1 \equiv G_2$ and relies on the existence of a common weak head normal form for the two types.

Rules $\lfloor$WFIO$\rfloor$ and $\lfloor$WFBRA$\rfloor$ say if subterms are equated and each type satisfies the kinding rule, then the resulting global types are equated.

Rule $\lfloor$WFPREC$\rfloor$ says the two recursive types are equated only if the bodies are equated. Note that we do not check whether unfolded recursive types are equated or not.

Rule $\lfloor$WFRVAR$\rfloor$ and $\lfloor$WFEND$\rfloor$ are the base cases.

Two recursors are equated if either (1) each subgraph is equated by $\equiv$ (rule $\lfloor$WFREC$\rfloor$), or if not, (2) they reduce to the same normal forms when applied to a finite number of indices (rule $\lfloor$WFRECF$\rfloor$). Note that rule $\lfloor$WFREC$\rfloor$ has a higher priority than rule $\lfloor$WFRECF$\rfloor$ (since it is more efficient without reducing recursors). If $\mathbf{R}\ G_1\ \lambda i : I.\lambda \mathbf{x}.G_1' \equiv_{\text{wf}} \mathbf{R}\ G_2\ \lambda i : I.\lambda \mathbf{x}.G_2'$ is derived by applying $\lfloor$WFREC$\rfloor$ under finite I, then the same equation can be derived using $\lfloor$WFRECF$\rfloor$. Thus, when the index range is finite, $\lfloor$WFREC$\rfloor$ subsumes $\lfloor$WFRECF$\rfloor$. On the other hand, $\lfloor$WFREC$\rfloor$ can be used for infinite index sets.

Similarly, $\lfloor$WFBASE$\rfloor$ has a higher priority than $\lfloor$WFAPP$\rfloor$. This ensures that the premise of $\lfloor$WFRECF$\rfloor$ always matches with $\lfloor$WFBASE$\rfloor$, not with $\lfloor$WFAPP$\rfloor$ (it avoids the infinite application of rules $\lfloor$WFRECF$\rfloor$ and $\lfloor$WFAPP$\rfloor$). A use of these rules are given in the examples later. Other rules are standard.

Type equivalence with meta-logic reasoning The set of rules in Figure 15 are designed with algorithmic checking in mind (see § 4.2). In some examples, in order to type processes with types that are not syntactically close, it is interesting to extend the equivalence classes on types, at the price of the decidability of type checking.

We propose in Figure 16 an additional equivalence rule that removes from rule $\lfloor$WFRECF$\rfloor$ the finiteness assumption on I. It allows to prove the equivalence of two recursor-based types if it is possible to prove meta-logically that they are extensionally equivalent. This technique can be used to type several of our examples (see § 5).

3.4 Typing processes

We explain here (Figure 17) the typing rules for the initial processes. Rules [TNAT] and [TVAR] are standard. Judgement $\Gamma \vdash \mathsf{Env}$ (defined in Figure 12) in the premise means that Γ is well-formed. For participants, we check their typing by [TID] and [TP] in a similar way as [40]. Rule [TPREC] deals with the changed index range within the recursor body. More precisely, we first check τ's kind. Then we verify for 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 type (induction hypothesis). Rule [TAPP] is the elimination rule for dependent types.

$$\frac{\Gamma \vdash \text{whnf}(G_1) \equiv_{\text{wf}} \text{whnf}(G_2)}{\Gamma \vdash G_1 \equiv G_2} \lfloor \textsc{WfBase} \rfloor$$

$$\frac{\Gamma \vdash U_1 \equiv U_2 \quad \Gamma \vdash G_1 \equiv G_2 \quad \Gamma \vdash \mathtt{p} \to \mathtt{p}' : \langle U_i \rangle . G_i \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mathtt{p} \to \mathtt{p}' : \langle U_1 \rangle . G_1 \equiv_{\text{wf}} \Gamma \vdash \mathtt{p} \to \mathtt{p}' : \langle U_2 \rangle . G_2} \lfloor \textsc{WfIO} \rfloor$$

$$\frac{\forall k \in K.\, \Gamma \vdash G_{1k} \equiv G_{2k} \quad \Gamma \vdash \mathtt{p} \to \mathtt{q} \colon \{l_k : G_{jk}\}_{k \in K} \blacktriangleright \mathsf{Type} \;\; (j = 1, 2)}{\Gamma \vdash \mathtt{p} \to \mathtt{q} \colon \{l_k : G_{1k}\}_{k \in K} \equiv_{\text{wf}} \mathtt{p} \to \mathtt{q} \colon \{l_k : G_{2k}\}_{k \in K}} \lfloor \textsc{WfBra} \rfloor$$

$$\frac{\Gamma \vdash G_1 \equiv G_2}{\Gamma \vdash \mu \mathbf{x}.G_1 \equiv_{\text{wf}} \mu \mathbf{x}.G_2} \lfloor \textsc{WfPRec} \rfloor \quad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathbf{x} \equiv_{\text{wf}} \mathbf{x}} \lfloor \textsc{WfRVar} \rfloor \quad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \equiv_{\text{wf}} \mathsf{end}} \lfloor \textsc{WfEnd} \rfloor$$

$$\frac{\Gamma \vdash G_1 \equiv G_2 \quad \Gamma, i : I \vdash G_1' \equiv G_2'}{\Gamma \vdash \mathbf{R}\, G_1\, \lambda i : I.\lambda \mathbf{x}.G_1' \equiv_{\text{wf}} \mathbf{R}\, G_2\, \lambda i : I.\lambda \mathbf{x}.G_2'} \lfloor \textsc{WfRec} \rfloor$$

$$\frac{\begin{array}{l}\Gamma \vdash G_1 \equiv G_2 \\ \Gamma \vdash \mathbf{R}\, G_1\, \lambda i : I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \mathbf{R}\, G_2\, \lambda i : I.\lambda \mathbf{x}.G_2'\ \mathrm{n} \quad \Gamma \models I = [0..\mathrm{m}] \quad 1 \leq \mathrm{n} \leq \mathrm{m}\end{array}}{\Gamma \vdash \mathbf{R}\, G_1\, \lambda i : I.\lambda \mathbf{x}.G_1' \equiv_{\text{wf}} \mathbf{R}\, G_2\, \lambda i : I.\lambda \mathbf{x}.G_2'} \lfloor \textsc{WfRecF} \rfloor$$

$$\frac{\Gamma \vdash G_1 \equiv_{\text{wf}} G_2 \quad \Gamma \models \mathtt{i}_1 : I = \mathtt{i}_2 : I \quad \Gamma \vdash G_i \mathtt{i}_i \blacktriangleright \kappa \quad (i = 1, 2)}{\Gamma \vdash G_1 \mathtt{i}_1 \equiv_{\text{wf}} G_2 \mathtt{i}_2} \lfloor \textsc{WfApp} \rfloor$$

Fig. 15. Global type equivalence rules

$$\frac{\begin{array}{c}\Gamma \vdash G_1 \equiv G_2 \\ \forall \mathrm{n} \in I. \Gamma \vdash \mathbf{R}\, G_1\, \lambda i : I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \mathbf{R}\, G_2\, \lambda i : I.\lambda \mathbf{x}.G_2'\ \mathrm{n}\end{array}}{\Gamma \vdash \mathbf{R}\, G_1\, \lambda i : I.\lambda \mathbf{x}.G_1' \equiv \mathbf{R}\, G_2\, \lambda i : I.\lambda \mathbf{x}.G_2'} \lfloor \textsc{WfRecExt} \rfloor$$

Fig. 16. Global type equivalence rule

$$\frac{\Gamma \models 0 \leq \mathtt{i} \mathbin{\mathrm{op}} \mathtt{i}'}{\Gamma \vdash \mathtt{i} \mathbin{\mathrm{op}} \mathtt{i}' \rhd \mathsf{nat}}\ [\textsc{TIOp}] \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathrm{n} \rhd \mathsf{nat}}\ [\textsc{TNat}]$$

$$\frac{\Gamma, i : I \vdash \mathsf{Env}}{\Gamma, i : I \vdash i \rhd \mathsf{nat}}\ [\textsc{TVari}] \qquad \frac{\Gamma \vdash \kappa}{\Gamma \vdash \mathtt{Alice} \rhd \kappa}\ [\textsc{TId}] \qquad \frac{\Gamma \vdash \mathtt{p} \rhd \Pi i{:}I.\kappa \quad \Gamma \models \mathtt{i}{:}I}{\Gamma \vdash \mathtt{p}[\mathtt{i}] \rhd \kappa\{\mathtt{i}/i\}}\ [\textsc{TP}]$$

$$\frac{\Gamma, i : I^-, X : \tau\{i/j\} \vdash Q \rhd \tau\{i+1/j\} \quad \Gamma \vdash P \rhd \tau\{0/j\} \quad \Gamma, j{:}I \vdash \tau \blacktriangleright \kappa}{\Gamma \vdash \mathbf{R}\ P\ \lambda i.\lambda X.Q \rhd \Pi j{:}I.\tau}\ [\textsc{TpRec}]$$

$$\frac{\Gamma \vdash P \rhd \tau \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash P \rhd \tau'}\ [\textsc{TEq}] \qquad \frac{\Gamma \vdash P \rhd \Pi i{:}I.\tau \quad \Gamma \models \mathtt{i} : I}{\Gamma \vdash P\ \mathtt{i} \rhd \tau\{\mathtt{i}/i\}}\ [\textsc{TApp}]$$

$$\frac{\Gamma, X : \tau \vdash P \rhd \tau}{\Gamma \vdash \mu X.P \rhd \tau}\ [\textsc{TRec}] \qquad \frac{\Gamma, X : \tau \vdash \mathsf{Env} \quad \Gamma \vdash \tau \approx \tau'}{\Gamma, X : \tau \vdash X \rhd \tau'}\ [\textsc{TVar}]$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, y : G \upharpoonright \mathtt{p}_0 \quad \Gamma \vdash \mathtt{p}_i \rhd \mathsf{nat} \quad \Gamma \models \mathsf{pid}(G) = \{\mathtt{p}_0..\mathtt{p}_\mathrm{n}\}}{\Gamma \vdash \bar{u}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y).P \rhd \Delta}\ [\textsc{TInit}]$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, y : G \upharpoonright \mathtt{p} \quad \Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \Gamma \models \mathtt{p} \in \mathsf{pid}(G)}{\Gamma \vdash u[\mathtt{p}](y).P \rhd \Delta}\ [\textsc{TAcc}]$$

$$\frac{\Gamma \vdash a : \langle G \rangle \ \ \Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \ \ \Gamma \models \mathtt{p} \in \mathsf{pid}(G)}{\Gamma \vdash \bar{a}[\mathtt{p}] : s \rhd s[\mathtt{p}] : G \upharpoonright \mathtt{p}}\ [\textsc{TReq}]$$

$$\frac{\Gamma \vdash e \rhd S \quad \Gamma \vdash P \rhd \Delta, c : T}{\Gamma \vdash c!\langle \mathtt{p}, e \rangle; P \rhd \Delta, c :!\langle \mathtt{p}, S \rangle; T}\ [\textsc{TOut}] \qquad \frac{\Gamma, x : S \vdash P \rhd \Delta, c : T}{\Gamma \vdash c?(\mathtt{p}, x); P \rhd \Delta, c :?\langle \mathtt{p}, S \rangle; T}\ [\textsc{TIn}]$$

$$\frac{\Gamma \vdash P \rhd \Delta, c : T}{\Gamma \vdash c!\langle \mathtt{p}, c' \rangle; P \rhd \Delta, c :!\langle \mathtt{p}, T' \rangle; T, c' : T'}\ [\textsc{TDeleg}] \qquad \frac{\Gamma \vdash P \rhd \Delta, c : T, y : T'}{\Gamma \vdash c?(\mathtt{p}, y); P \rhd \Delta, c :?\langle \mathtt{p}, T' \rangle; T}\ [\textsc{TRecep}]$$

$$\frac{\Gamma \vdash P \rhd \Delta, c : T_j \quad j \in K}{\Gamma \vdash c \oplus \langle \mathtt{p}, l_j \rangle; P \rhd \Delta, c : \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k \in K} \rangle}\ [\textsc{TSel}]$$

$$\frac{\forall k \in K, \Gamma \vdash P_k \rhd \Delta, c : T_k}{\Gamma \vdash c\&(\mathtt{p}, \{l_k : P_k\}_{k \in K}) \rhd \Delta, c : \&\langle \mathtt{p}, \{l_k : T_k\}_{k \in K} \rangle}\ [\textsc{TBra}]$$

$$\frac{\Gamma, a : U \vdash P \rhd \Delta}{\Gamma \vdash (\nu a)P \rhd \Delta}\ [\textsc{TNu}] \qquad \frac{\Gamma \vdash \Delta \quad \Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \rhd \Delta}\ [\textsc{TNull}] \qquad \frac{\Gamma \vdash P \rhd \Delta \quad \Gamma \vdash Q \rhd \Delta'}{\Gamma \vdash P \mid Q \rhd \Delta, \Delta'}\ [\textsc{TPar}]$$

Fig. 17. Initial expression and process typing

Rule [TEQ] states that typing works up to type equivalence where $\equiv$ is defined in the previous subsection. Recursion [TREC] rule is standard. In rule [TVAR], $\Delta \approx \Delta'$ denotes the standard isomorphism rules for recursive types (i.e. we identify $\mu \mathbf{x}.T$ and $T\{\mu \mathbf{x}.T/\mathbf{x}\}$), see Appendix A.1. Note that we apply isomorphic rules only when recursive variables are introduced. This way, we can separate type isomorphism for recursive types and type equalities with recursors.

Rule [TINIT] types a session initialisation on shared channel u, binding channel y and requiring participants $\{\mathtt{p}_0, .., \mathtt{p}_n\}$. The premise verifies that the type of y is the first projection of the global type G of u and that the participants in G (denoted by $\mathtt{pid}(G)$) can be semantically derived as $\{\mathtt{p}_0, .., \mathtt{p}_n\}$.

Rule [TACC] allows to type the $\mathtt{p}$-th participant to the session initiated on u. The typing rule checks that the type of y is the $\mathtt{p}$-th projection of the global type G of u and that G is fully instantiated. The kind rule ensures that G is fully instantiated (i.e. G''s kind is Type). Rule [TREQ] types the process that waits for an accept from a participant: its type corresponds to the end-point projection of G.

The next four rules are associate the input/output processes to the input/output types, and delegation input/output processes to session input/output types. Then the next two rules are branching/selection rules.

Rule [TNULL] checks that Δ is well-formed and only contains end-type for weakening (Δ end only means $\forall c \in dom(\Delta).\Delta(c) = \mathsf{end}$). Rule [TPAR] puts in parallel two processes only if their sessions environments have disjoint domains. Other rules are standard.

4 Properties of typing

We study the two main properties of the typing system: one is the termination of type-checking and another is type-soundness. The proofs require a careful analysis due to the subtle interplay between dependent types, recursors, recursive types and branching types.

4.1 Basic properties

We prove here a series of consistency lemmas concerning permutations and weakening. They are invariably deduced by induction on the derivations in the standard manner.

We use the following additional notations: $\Gamma \subseteq \Gamma'$ iff $u : S \in \Gamma$ implies $u : S \in \Gamma'$ and similarly for other mappings. In other words, $\Gamma \subseteq \Gamma'$ means that Γ' is a permutation of an extension of Γ.

Lemma 4.1 *1. (Permutation and Weakening) Suppose $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \mathsf{Env}$. Then $\Gamma \vdash J$ implies $\Gamma' \vdash J$.*

2. *(Strengthening) $\Gamma, u : U, \Gamma' \vdash J$ and $u \notin \mathsf{fv}(\Gamma', J) \cup \mathsf{fn}(\Gamma', J)$. Then $\Gamma, \Gamma' \vdash J$. Similarly for other mapping.*
3. *(Agreement)*
   (a) $\Gamma \vdash J$ implies $\Gamma \vdash \mathsf{Env}$.
   (b) $\Gamma \vdash G \blacktriangleright \kappa$ implies $\Gamma \vdash \kappa$. Similarly for other judgements.

(c) $\Gamma \vdash G \equiv G'$ *implies* $\Gamma \vdash G \blacktriangleright \kappa$. *Similarly for other judgements.*
(d) $\Gamma \vdash P \rhd \tau$ *implies* $\Gamma \vdash \tau \blacktriangleright \kappa$. *Similarly for other judgements.*

4. *(Exchange)*
   (a) $\Gamma, u:U, \Gamma' \vdash J$ *and* $\Gamma \vdash U \equiv U'$. *Then* $\Gamma, u:U', \Gamma' \vdash J$. *Similarly for other mappings.*
   (b) $\Gamma, i:I, \Gamma' \vdash J$ *and* $\Gamma \models i:I = i:I'$. *Then* $\Gamma, i:I', \Gamma' \vdash J$.
   (c) $\Gamma, \mathtt{P}, \Gamma' \vdash J$ *and* $\Gamma \models \mathtt{P} = \mathtt{P}'$. *Then* $\Gamma, \mathtt{P}', \Gamma' \vdash J$.

Proof. By induction on the derivations. We note that the proofs are done simultaneously. For the rules which use substitutions in the conclusion of the rule, such as [TAPP] in Figure 17, we require to use the next substitution lemma simultaneously. We only show the most interesting case with a recursor.

Proof of (3)(b). Case $\lfloor$KRCR$\rfloor$**:** Suppose $\Gamma \vdash \mathbf{R}\ G\ \lambda i{:}I^{-}.\lambda \mathbf{x}.G' \blacktriangleright \Pi j{:}I.\kappa$ is derived by $\lfloor$KRCR$\rfloor$ in Figure 14. We prove $\Gamma \vdash \Pi j{:}I.\kappa$. From $\Gamma, i : I^{-} \vdash G' \blacktriangleright \kappa\{i+1/j\}$ in the premise of $\lfloor$KRCR$\rfloor$, we have $\Gamma, i : I^{-} \vdash \kappa\{i+1/j\}$ by inductive hypothesis. By definition of I^{-}, this implies $\Gamma, j : I \vdash \kappa$. Now by $\lfloor$KSEQ$\rfloor$, we have $\Gamma \vdash \Pi j{:}I.\kappa$, as desired. □

The following lemma which states that well-typedness is preserved by substitution of appropriate values for variables, is the key result underlying Subject Reduction. This also guarantees that the substitution for the index which affects to a shared environment and a type of a term, and the substitution for a process variable are always well-defined. Note that substitutions may change session types and environments in the index case.

Lemma 4.2 (Substitution lemma) *1. If* $\Gamma, i : I, \Gamma' \vdash_{\Sigma} J$ *and* $\Gamma \models \mathrm{n} : I$, *then* $\Gamma, (\Gamma'\{\mathrm{n}/i\}) \vdash_{\Sigma} J\{\mathrm{n}/i\}$.
2. If $\Gamma, X : \Delta_0 \vdash_{\Sigma} P \rhd \tau$ *and* $\Gamma \vdash Q : \Delta_0$, *then* $\Gamma \vdash_{\Sigma} P\{Q/X\} \rhd \tau$.
3. If $\Gamma, x : S \vdash_{\Sigma} P \rhd \Delta$ *and* $\Gamma \vdash v : S$, *then* $\Gamma \vdash_{\Sigma} P\{v/x\} \rhd \Delta$.
4. If $\Gamma \vdash_{\Sigma} P \rhd \Delta, y : T$, *then* $\Gamma \vdash_{\Sigma} P\{s[\hat{\mathrm{p}}]/y\} \rhd \Delta, s[\hat{\mathrm{p}}] : T$.

Proof. By induction on the derivations. We prove the most interesting case: if $\Gamma, i : I, \Gamma' \vdash_{\Sigma} P \rhd \tau$ and $\Gamma \vdash \mathrm{n} \rhd \mathsf{nat}$ with $\Gamma \models \mathrm{n} : I$, then $\Gamma, (\Gamma'\{\mathrm{n}/i\}) \vdash_{\Sigma} P\{\mathrm{n}/i\} \rhd \tau\{\mathrm{n}/i\}$ when the last applied rule is [TPREC]. Assume

$$\Gamma, k : J, \Gamma' \vdash \mathbf{R}\ P\ \lambda i.\lambda X.Q \rhd \Pi j{:}I.\tau \quad \text{with} \quad \Gamma \models \mathrm{n} : J$$

is derived from [TPREC]. This is derived by:

$$\Gamma, k : J, \Gamma', i : I^{-}, X : \tau\{i/j\} \vdash Q \rhd \tau\{i+1/j\} \tag{4}$$
$$\Gamma, k : J, \Gamma' \vdash P \rhd \tau\{0/j\} \tag{5}$$
$$\Gamma, k : J, \Gamma', j{:}I \vdash \tau \blacktriangleright \kappa \tag{6}$$
From (4) $\Gamma, \Gamma'\{\mathrm{n}/k\}, i : I^{-}\{\mathrm{n}/k\}, X : \tau\{i/j\}\{\mathrm{n}/k\} \vdash Q\{\mathrm{n}/k\} \rhd \tau\{i+1/j\}\{\mathrm{n}/k\}$ (7)
From (5) $\Gamma, \Gamma'\{\mathrm{n}/k\} \vdash P\{\mathrm{n}/k\} \rhd \tau\{0/j\}\{\mathrm{n}/k\}$ (8)
From (6) $\Gamma, \Gamma'\{\mathrm{n}/k\}, j{:}I\{\mathrm{n}/k\} \vdash \tau\{\mathrm{n}/k\} \blacktriangleright \kappa\{\mathrm{n}/k\}$ (9)

From (7), (8) and (9), by [TPREC], we obtain $\Gamma, \Gamma'\{\mathrm{n}/k\} \vdash (\mathbf{R}\ P\ \lambda i.\lambda X.Q)\{\mathrm{n}/k\} \rhd (\Pi j{:}I.\tau)\{\mathrm{n}/k\}$ as required. □

4.2 Termination of equality checking and type checking

This subsection proves the termination of the type-checking (we assume that we use the type equality rules in Figure 15). Ensuring termination of type-checking with dependent types is not an easy task since type equivalences are often derived from term equivalences. We rely on the strong normalisation of System $\mathcal{T}$ [20] for the termination proof.

Proposition 4.3 (Termination and confluence) *The reduction relation $\longrightarrow$ on global and end-point types (i.e. $G \longrightarrow G'$ and $T \longrightarrow T'$ for closed types in Figure 2) are strong normalising and confluent on well-formed kindings. More precisely, if $\Gamma \vdash G \blacktriangleright \kappa$, then there exists a unique term $G' = whnf(G)$ such that $G \longrightarrow^* G' \not\longrightarrow$.*

Proof. By strong normalisation of System $\mathcal{T}$ [20]. For the confluence, we first note that the reduction relation on global types defined in Figure 2 and on expressions with the first-order operators in the types is *deterministic*, i.e. if $G \longrightarrow G_i$ by rules in Figure 2, then $G_1 = G_2$. Hence it is locally confluent, i.e. if $G \longrightarrow G_i$ $(i = 1, 2)$ then $G_i \longrightarrow^* G'$. Then we achieve the result by Newman's Lemma. The second clause is a direct consequence from the fact that $\longrightarrow$ coincides with the head reduction. □

Proving the termination of type equality checking requires a detailed analysis since the premises of the mathematical induction rules compare the two types whose syntactic sizes are larger than those of their conclusions The size of the judgements are defined in Figure 18, using the following functions from [1, § 2.4.3], taking rule $\lfloor$WFRECF$\rfloor$ in Figure 15 into account.

1. $|G|$ is the size of the structure of G where we associate ω^2-valued weight to each judgement to represent a possible reduction to a weak head normal form.
2. $||G||$ is the size of the structure of G where unfolding of recursors with finite index sets is considered, taking $\lfloor$WFRECF$\rfloor$ in Figure 15 into account.
3. $\mu(G)$ denotes an upper bound on the length of any $\longrightarrow$ reduction sequence (see Figure 2) starting from G and its subterms.
4. $\mu^\star(G)$ denotes an upper bound on the length of any $\longrightarrow$ reduction sequence (see Figure 2) starting from G and its subterms. It unfolds recursors with finite index sets.

The definition of the size of judgement $w(\Gamma \vdash G_1 \equiv_{\text{wf}} G_2)$ follows [1, § 2.4.3]. We use $\mu^\star(G)$ and $||G||$ because of $\lfloor$WFRECF$\rfloor$ in Figure 15. Note that $|G|$ corresponds to $\mu(G)$, while $||G||$ corresponds to $\mu^\star(G)$. In $\mu^\star(G)$, we incorporate the number of reductions of unfolded recursors. Because the reduction of expressions strongly normalises, we choose the size of e to be 1 and the length of reductions of (closed) e is assumed to be 0.

The termination of type equality checking then is proved by the following main lemma.

Lemma 4.4 (Size of equality judgements) *The weight of any premise of a rule is always less than the weight of the conclusion.*

Judgements $w(\cdot)$ $w(\Gamma \vdash G_1 \equiv G_2) = w(\Gamma \vdash G_1 \equiv_{\text{wf}} G_2) + 1$

$w(\Gamma \vdash G_1 \equiv_{\text{wf}} G_2) = \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 1$

Types $|\cdot|$

Value	$\lvert\mathsf{nat}\rvert = 1, \lvert\langle G\rangle\rvert = \lvert G\rvert + 1$
Global	$\lvert\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G\rvert = 2 + \lvert U\rvert + \lvert G\rvert$
	$\lvert\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k\in K}\rvert = 2 + \Sigma_{k\in K}(1 + \lvert G_k\rvert)$
	$\lvert\mu\mathbf{x}.G\rvert = \lvert G\rvert + 2, \quad \lvert\mathbf{x}\rvert = \lvert\mathrm{n}\rvert = \lvert\mathsf{end}\rvert = 1$
	$\lvert G\ \mathtt{i}\rvert = \lvert G\rvert + 2\ (\mathsf{fv}(\mathtt{i}) = \emptyset) \quad \lvert G\ \mathtt{i}\rvert = \lvert\lvert G\rvert\rvert + 2\ (\mathsf{fv}(\mathtt{i}) \neq \emptyset)$
	$\lvert\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G'\rvert = 4 + \lvert\lvert G\rvert\rvert + \lvert\lvert G'\rvert\rvert$
Local	$\lvert!\langle\mathtt{p}, U\rangle; T\rvert = 3 + \lvert U\rvert + \lvert T\rvert, \quad \lvert?\langle\mathtt{p}, U\rangle; T\rvert = 3 + \lvert U\rvert + \lvert T\rvert$
	$\lvert\oplus\langle\mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\rvert = \lvert\&\langle\mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\rvert = 2 + \Sigma_{k\in K}(1 + \lvert T_k\rvert)$
	$\lvert\mu\mathbf{x}.T\rvert = \lvert T\rvert + 2, \quad \lvert\mathbf{x}\rvert = \lvert\mathrm{n}\rvert = \lvert\mathsf{end}\rvert = 1$
	$\lvert T\ \mathtt{i}\rvert = \lvert T\rvert + 2\ (\mathsf{fv}(\mathtt{i}) = \emptyset) \quad \lvert T\ \mathtt{i}\rvert = \lvert\lvert T\rvert\rvert + 2\ (\mathsf{fv}(\mathtt{i}) \neq \emptyset)$
	$\lvert\mathbf{R}\ T\ \lambda i\!:\!I.\lambda\mathbf{x}.T'\rvert = 4 + \lvert\lvert T\rvert\rvert + \lvert\lvert T'\rvert\rvert$
Principals	$\lvert\Pi i\!:\!I.U_p\rvert = 2 + \lvert U_p\rvert$
Processes	$\lvert\emptyset\rvert = 0, \lvert\Delta, c\!:\!T\rvert = \lvert\Delta\rvert + \lvert T\rvert + 1, \lvert\Pi i\!:\!I.\tau\rvert = 2 + \lvert\tau\rvert$

Types $||\cdot||$

Global	$\lvert\lvert\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G'\rvert\rvert = \Sigma_{\mathrm{n}\in I}\lvert\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G'\mathrm{n}\rvert$ (I finite)
Local	$\lvert\lvert\mathbf{R}\ T\ \lambda i\!:\!I.\lambda\mathbf{x}.T'\rvert\rvert = \Sigma_{\mathrm{n}\in I}\lvert\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G'\mathrm{n}\rvert$ (I finite)
	Others are $\lvert\lvert G\rvert\rvert = \lvert G\rvert$.

Types $\mu(\cdot)$

Global	$\mu(G\ \mathtt{i}) = \mu^\star(G)\ (\mathsf{fv}(\mathtt{i}) \neq \emptyset), \quad \mu(\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G') = \mu^\star(G) + \mu^\star(G')$
Local	$\mu(T\ \mathtt{i}) = \mu^\star(T)\ (\mathsf{fv}(\mathtt{i}) \neq \emptyset), \quad \mu(\mathbf{R}\ T\ \lambda i\!:\!I.\lambda\mathbf{x}.T') = \mu^\star(T) + \mu^\star(T')$

Types $\mu^\star(\cdot)$

Global	$\mu^\star(\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G') = \Sigma_{\mathrm{n}\in I}\mu(\mathbf{R}\ G\ \lambda i\!:\!I.\lambda\mathbf{x}.G'\mathrm{n})$
Local	$\mu^\star(\mathbf{R}\ T\ \lambda i\!:\!I.\lambda\mathbf{x}.T') = \Sigma_{\mathrm{n}\in I}\mu(\mathbf{R}\ T\ \lambda i\!:\!I.\lambda\mathbf{x}.T'\mathrm{n})$
	Others are $\mu^\star(G) = \mu(G)$ and homomorphic.

Fig. 18. Size of types and judgements and the upper bound of reductions with unfolding

Proof. Our proof is by induction on the length of reduction sequences and the size of terms.

Case $\lfloor$WFBASE$\rfloor$**.** The case $\mathrm{whnf}(G_1) = G_1$ and $\mathrm{whnf}(G_2) = G_2$ are obvious by definition. Hence we assume $\mathrm{whnf}(G_1) \neq G_1$. Thus there exists at least one step reduction such that $G_1 \longrightarrow G_1'$. Note that for any G, we have $||G|| < \omega$. Hence we have

$$\begin{aligned}
&w(\Gamma \vdash \mathrm{whnf}(G_1) \equiv_{\mathrm{wf}} \mathrm{whnf}(G_2)) \\
&= \omega \cdot \mu^\star(\mathrm{whnf}(G_1)) + ||\mathrm{whnf}(G_1)|| + \omega \cdot \mu^\star(\mathrm{whnf}(G_2)) + ||\mathrm{whnf}(G_2)|| + 1 \\
&< \omega \cdot \mu^\star(G_1) + ||G_1|| + \omega \cdot \mu^\star(G_2) + ||G_2|| + 2 \\
&= w(\Gamma \vdash G_1 \equiv G_2)
\end{aligned}$$

Case $\lfloor$WFIO$\rfloor$**.** Similar with $\lfloor$WFBRA$\rfloor$ below.

Case $\lfloor$WFBRA$\rfloor$**.**

$$\begin{aligned}
&\Sigma_{k\in K} w(\Gamma \vdash G_{1k} \equiv G_{2k}) \\
&= \Sigma_{k\in K}(w(\Gamma \vdash G_{1k} \equiv_{\mathrm{wf}} G_{2k}) + 1) \\
&= \Sigma_{k\in K}(\omega \cdot (\mu^\star(G_{1k}) + \mu^\star(G_{2k})) + ||G_{1k}|| + ||G_{2k}|| + 2) \\
&= \Sigma_{k\in K}(\omega \cdot (\mu(G_{1k}) + \mu(G_{2k})) + |G_{1k}| + |G_{2k}| + 2) \\
&< \Sigma_{k\in K}(\omega \cdot (\mu(G_{1k}) + \mu(G_{2k})) + |G_{1k}| + |G_{2k}| + 2) + 4 \\
&= w(\Gamma \vdash \mathtt{p} \to \mathtt{q}\colon \{l_k : G_{1k}\}_{k\in K} \equiv_{\mathrm{wf}} \mathtt{p} \to \mathtt{q}\colon \{l_k : G_{2k}\}_{k\in K})
\end{aligned}$$

We note that G_{1k} and G_{2k} cannot be recursors since $\Gamma \vdash G_{ik} \blacktriangleright \mathsf{Type}$ by the kinding rule, hence $\mu(G_{ik}) = \mu^\star(G_{ik})$ and $|G_{ik}| = ||G_{ik}||$ in the above third equation.

Case $\lfloor$WFPREC$\rfloor$**.** Similar with $\lfloor$WFBRA$\rfloor$ above.

Cases $\lfloor$WFRVAR$\rfloor$**,**$\lfloor$WFEND$\rfloor$**.** By definition.

Case $\lfloor$WFREC$\rfloor$**.** The case I is infinite.

$$\begin{aligned}
&w(\Gamma \vdash G_1 \equiv G_2) + w(\Gamma, i\colon I \vdash G_1' \equiv G_2') \\
&= \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 2 + \omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 2 \\
&< \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 4 + \omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 4 \\
&= w(\Gamma \vdash \mathbf{R}\, G_1\ \lambda i\colon I.\lambda \mathbf{x}.G_1' \equiv_{\mathrm{wf}} \mathbf{R}\, G_2\ \lambda i\colon I.\lambda \mathbf{x}.G_2')
\end{aligned}$$

The case I is finite.

$$\begin{aligned}
&w(\Gamma \vdash G_1 \equiv G_2) + w(\Gamma, i\colon I \vdash G_1' \equiv G_2') \\
&= \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 2 + \omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 2 \\
&< \Sigma_{n\in I}(\omega \cdot (m_{1n} + \mu^\star(G_{1n}'') + m_{2n} + \mu^\star(G_{2n}''))) \\
&\quad + \Sigma_{n\in I}(\omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 4 + 2) \\
&\quad + \Sigma_{n\in I}(\omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 4 + 2) \\
&= w(\Gamma \vdash \mathbf{R}\, G_1\ \lambda i\colon I.\lambda \mathbf{x}.G_1' \equiv_{\mathrm{wf}} \mathbf{R}\, G_2\ \lambda i\colon I.\lambda \mathbf{x}.G_2')
\end{aligned}$$

where we assume $m_{i\mathrm{n}}$ is the length of the reduction sequence from $\mathbf{R}\, G_i\ \lambda i\colon I.\lambda \mathbf{x}.G_i'\ \mathrm{n}$ (in Figure 2), and $\mathbf{R}\, G_i\ \lambda i\colon I.\lambda \mathbf{x}.G_i'\ \mathrm{n} \longrightarrow^* G_{i\mathrm{n}}'' \not\longrightarrow$ for all $\mathrm{n} \in I$.

Case $\lfloor$WFRECF$\rfloor$**.** Assume $I = [0, \ldots, \mathrm{m}]$.

$$
\begin{aligned}
& w(\Gamma \vdash G_1 \equiv G_2) + \Sigma_{1 \leq \mathrm{n} \leq m} w(\Gamma \vdash \mathbf{R}\ G_1\ \lambda i{:}I.\lambda \mathbf{x}.G_1'\mathrm{n} \equiv \mathbf{R}\ G_2\ \lambda i{:}I.\lambda \mathbf{x}.G_2'\mathrm{n}) \\
= {} & \omega \cdot \mu^\star(G_1) + ||G_1|| + \omega \cdot \mu^\star(G_2) + ||G_2|| + 2 \\
& + \Sigma_{1 \leq \mathrm{n} \leq m}(w(\Gamma \vdash \mathbf{R}\ G_1\ \lambda i{:}I.\lambda \mathbf{x}.G_1'\mathrm{n} \equiv_{\mathrm{wf}} \mathbf{R}\ G_2\ \lambda i{:}I.\lambda \mathbf{x}.G_2'\mathrm{n}) + 1) \\
= {} & \omega \cdot ((m_{11} - 1) + \mu^\star(G_{11}'') + (m_{21} - 1) + \mu^\star(G_{21}'')) + ||G_1|| + ||G_2|| + 2 \\
& + \Sigma_{1 \leq \mathrm{n} \leq m}(\omega \cdot (m_{1n} + \mu^\star(G_{1n}'') + m_{2n} + \mu^\star(G_{2n}''))) \\
& + \Sigma_{1 \leq \mathrm{n} \leq m}(\omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 4 + 2) \\
& + \Sigma_{1 \leq \mathrm{n} \leq m}(\omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 4 + 2) \\
< {} & \Sigma_{n \in I}(\omega \cdot (m_{1n} + \mu^\star(G_{1n}'') + m_{2n} + \mu^\star(G_{2n}''))) \\
& + \Sigma_{n \in I}(\omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 4 + 2) \\
& + \Sigma_{n \in I}(\omega \cdot (\mu^\star(G_1') + \mu^\star(G_2')) + ||G_1'|| + ||G_2'|| + 4 + 2) \\
= {} & w(\Gamma \vdash \mathbf{R}\ G_1\ \lambda i{:}I.\lambda \mathbf{x}.G_1' \equiv_{\mathrm{wf}} \mathbf{R}\ G_2\ \lambda i{:}I.\lambda \mathbf{x}.G_2')
\end{aligned}
$$

where we assume $m_{i\mathrm{n}}$ is the length of the reduction sequence from $\mathbf{R}\ G_i\ \lambda i{:}I.\lambda \mathbf{x}.G_i'\ \mathrm{n}$ (in Figure 2), and $\mathbf{R}\ G_i\ \lambda i{:}I.\lambda \mathbf{x}.G_i'\ \mathrm{n} \longrightarrow^* G_{i\mathrm{n}}'' \not\longrightarrow$ for all $\mathrm{n} \in I$.
Case $\lfloor$WFAPP$\rfloor$ First, since $\lfloor$WFAPP$\rfloor$ has a lower priority than $\lfloor$WFBASE$\rfloor$, $G_i \mathtt{i}_i \not\longrightarrow$. Hence $\mu^\star(G_i \mathtt{i}_i) = \mu^\star(G_i)$. We also note that:

1. If G_i is not recursor, then $||G_i|| = |G_i|$; and
2. If G_i is a recursor, then $\mathtt{i}_i$ contains free variables (since $\lfloor$WFAPP$\rfloor$ has a lower priority than $\lfloor$WFBASE$\rfloor$, if $\mathtt{i}_i$ is closed, it reduces to m for some m), hence $||G_i\ \mathtt{i}_i|| = ||G_i|| + 2$.

There is no case such that G_i is a recursor and $\mathtt{i}_i$ is some natural number n since, if so, we can apply $\lfloor$WFBASE$\rfloor$. Thus, by (1,2), $||G_i\ \mathtt{i}_i|| = ||G_i|| + 2$. Hence we have:

$$
\begin{aligned}
& w(\Gamma \vdash G_1 \equiv_{\mathrm{wf}} G_2) \\
= {} & \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 1 \\
< {} & \omega \cdot (\mu^\star(G_1) + \mu^\star(G_2)) + ||G_1|| + ||G_2|| + 4 + 1 \\
= {} & w(\Gamma \vdash G_1\mathtt{i}_1 \equiv_{\mathrm{wf}} G_2\mathtt{i}_2)
\end{aligned}
$$

as required. □

Proposition 4.5 (Termination for type equality checking) *Assuming that proving the predicates $\Gamma \models \mathtt{P}$ appearing in type equality derivations is decidable, then type-equality checking of $\Gamma \vdash G \equiv G'$ terminates. Similarly for other types.*

Proof. By Lemma 4.4 and termination of kinding and well-formed environment checking. □

We first formally define annotated processes which are processes with explicit type annotations for bound names and variables (see § 2.3).

$$
\begin{aligned}
P ::= {} & \bar{u}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y{:}T).P \mid u[\mathtt{p}](y{:}T).P \mid c?(\mathtt{p}, x{:}T); P \mid (\nu a{:}\langle G \rangle)P \\
& \mid \mu X{:}\tau.P \mid \mathbf{R}\ P\ \lambda i{:}I.\lambda X.Q \mid X^\tau
\end{aligned}
$$

Theorem 4.6 (Termination of type checking) *Assuming that proving the predicates $\Gamma \models \mathtt{P}$ appearing in kinding, equality, projection and typing derivations is decidable, then type-checking of annotated process P, i.e. $\Gamma \vdash P \triangleright \emptyset$ terminates.*

Proof. First, it is straightforward to show that kinding checking, well-formed environment checking and projection are decidable as long as deciding the predicates $\Gamma \models \mathtt{P}$ appearing in the rules is possible.

Secondly, we note that by the standard argument from indexed dependent types [1, 40], for the dependent λ-applications ([TAPP] in Figure 17), we do not require equality of terms (i.e. we only need the equality of the indices by the semantic consequence judgements).

Thirdly, by the result from [19, Corollary 2, page 217], the type isomorphic checking $\tau \approx \tau'$ terminates so that the type isomorphic checking in [TVAR] in Figure 17 (between τ in the environment and τ' of $X^{\tau'}$) always terminates.

Forth, it is known that type checking for annotated terms with session types terminates with subtyping [19, § 5.2] and multiparty session types [23].

Hence the rest of the proof consists in eliminating the type equality rule [TEQ] in order to make the rules syntax-directed. We include the type equality check into [TINIT,TREQ,TACC] (between the global type and its projected session type), the input rules [TIN,TRECEP] (between the session type and the type annotating x), [TREC] (between the session type and the type annotating X), and [TREC] (between the session type and the type annotating X in Γ). We show the three syntax-directed rules. The first rule is the initialisation.

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, y : T \quad \Gamma \vdash G \upharpoonright \mathtt{p}_0 \equiv T \quad \Gamma \vdash \mathtt{p}_i \rhd \mathsf{nat} \quad \Gamma \models \mathsf{pid}(G) = \{\mathtt{p}_0..\mathtt{p}_n\}}{\Gamma \vdash \bar{u}[\mathtt{p}_0, .., \mathtt{p}_n](y).P \rhd \Delta} \text{ [TINIT]}$$

Then it is straightforward to check $\Gamma \vdash u : \langle G \rangle$ terminates. Checking $\Gamma \vdash P \rhd \Delta, y : T$ also terminates by inductive hypothesis. Checking $\Gamma \vdash G \upharpoonright \mathtt{p}_0 \equiv T$ terminates since the projection terminates and checking $\alpha \equiv \beta$ (for any type α and β) terminates by Lemma 4.4. Checking $\Gamma \vdash \mathtt{p}_i \rhd \mathsf{nat}$ terminates since the kinding checking terminates. Finally checking $\Gamma \models \mathsf{pid}(G) = \{\mathtt{p}_0..\mathtt{p}_n\}$ terminates by assumption.

The second rule is the session input.

$$\frac{\Gamma \vdash P \rhd \Delta, c : T, y : T' \quad \Gamma \vdash T_0 \equiv T'}{\Gamma \vdash c?(\mathtt{p}, y{:}T_0); P \rhd \Delta, c :?\langle \mathtt{p}, T' \rangle; T} \text{ [TRECEP]}$$

Then checking $\Gamma \vdash P \rhd \Delta, c : T, y : T'$ terminates by inductive hypothesis and checking $\Gamma \vdash T_0 \equiv T'$ terminates by Lemma 4.4.

The third and forth rules are recursions.

$$\frac{\Gamma, X : \tau \vdash P \rhd \tau' \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash \mu X{:}\tau.P \rhd \tau'} \text{ [TREC]} \qquad \frac{\Gamma, X : \tau \vdash \mathsf{Env} \quad \Gamma \vdash \tau \approx \tau' \quad \Gamma \vdash \tau' \equiv \tau_0}{\Gamma, X : \tau \vdash X^{\tau'} \rhd \tau_0} \text{ [TVAR]}$$

In [TREC], we assume $P \neq X$ (such a term is meaningless). Then [TREC] terminates by inductive hypothesis and Lemma 4.4, while [TVAR] terminates by termination of isomorphic checking (as said above) and Lemma 4.4.

Other rules are similar, hence we conclude the proof. □

To ensure the termination of $\Gamma \models \mathtt{P}$, possible solutions include the restriction of predicates to linear equalities over natural numbers without multiplication (or to other decidable arithmetic subsets) or the restriction of indices to finite domains, cf. [40].

4.3 Type soundness and progress

The following lemma states that mergeability is sound with respect to the branching subtyping $\leq$. By this, we can safely replace the third clause $\sqcup_{k\in K} G_k \upharpoonright \mathtt{q}$ of the branching case from the projection definition by $\sqcap\{T \mid \forall k \in K.T \leq (G_k \upharpoonright \mathtt{q})\}$. This allows us to prove subject reduction by including subsumption as done in [23].

Lemma 4.7 (Soundness of mergeability) *Suppose $G_1 \upharpoonright \mathtt{p} \bowtie G_2 \upharpoonright \mathtt{p}$ and $\Gamma \vdash G_i$. Then there exists G such that $G \upharpoonright \mathtt{p} = \sqcap\{T \mid T \leq G_i \upharpoonright \mathtt{p}\ (i = 1, 2)\}$ where $\sqcap$ denotes the maximum element with respect to $\leq$.*

Proof. The only interesting case is when $G_1 \upharpoonright \mathtt{p}$ and $G_2 \upharpoonright \mathtt{p}$ take a form of the branching type. Suppose $G_1 = \mathtt{p}' \to \mathtt{p}\colon\{l_i : {G'}_i\}_{i\in I}$ and $G_2 = \mathtt{p}' \to \mathtt{p}\colon\{l_j : {G''}_j\}_{j\in J}$ with $G_1 \upharpoonright \mathtt{p} \bowtie G_2 \upharpoonright \mathtt{p}$. Let $G'_i \upharpoonright \mathtt{p} = T_i$ and $G''_j \upharpoonright \mathtt{p} = T'_j$. Then by the definition of $\bowtie$ in § 3.1, we have $G_1 \upharpoonright \mathtt{p} = \&\langle \mathtt{p}', \{l_i : T_i\}_{i\in I}\rangle$ and $G_2 \upharpoonright \mathtt{p} = \&\langle \mathtt{p}', \{l_j : T'_j\}_{j\in J}\rangle$ with $\forall i \in (I \cap J).T_i \bowtie T'_j \quad \forall i \in (I \setminus J) \cup (J \setminus I).l_i \neq l_j$. By the assumption and inductive hypothesis on $T_i \bowtie T'_j$, we can set

$$T = \&\langle \mathtt{p}', \{l_k : T''_k\}_{k\in K}\rangle$$

such that $K = I \cup J$; and (1) if $k \in I \cap J$, then $T''_k = T_k \sqcap T'_k$; (2) if $k \in I, k \notin J$, then $T''_k = T_k$; and (3) if $k \in J, k \notin I$, then $T''_k = T'_k$. Set $G_{0k} \upharpoonright \mathtt{p} = T''_k$. Then we can obtain

$$G = \mathtt{p}' \to \mathtt{p}\colon\{k_k : G_{0k}\}_{k\in K}$$

which satisfies $G \upharpoonright \mathtt{p} = \sqcap\{T \mid T \leq G_i \upharpoonright \mathtt{p}\ (i = 1, 2)\}$, as desired. □

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 $\equiv$.

- $\{s[\hat{\mathtt{p}}] :!\langle \hat{\mathtt{q}}, U\rangle; T, s[\hat{\mathtt{q}}] :?\langle \hat{\mathtt{p}}, U\rangle; T'\} \Rightarrow \{s[\hat{\mathtt{p}}] : T, s[\hat{\mathtt{q}}] : T'\}$
- $\{s[\hat{\mathtt{p}}] :!\langle \hat{\mathtt{p}}, U\rangle; ?\langle \hat{\mathtt{p}}, U\rangle; T'\} \Rightarrow \{s[\hat{\mathtt{p}}] : T'\}$
- $\{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, \{l_k : T_k\}_{k\in K}\rangle\} \Rightarrow \{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, l_j\rangle; T_j\}$
- $\{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, l_j\rangle; T, s[\hat{\mathtt{q}}] : \&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\} \Rightarrow \{s[\hat{\mathtt{p}}] : T, s[\hat{\mathtt{q}}] : T_j\}$
- $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta''$ if $\Delta \Rightarrow \Delta'$.

The first rule corresponds to the reception of a value or channel by the participant $\hat{\mathtt{q}}$; the second rule formalises the reception of a value or channel sent by itself $\hat{\mathtt{p}}$; the third rule treats the case of the choice of label l_j while the forth rule propagate these choices to the receiver (participant $\hat{\mathtt{q}}$).

For the subject reduction theorem, we need to define *the coherence of the session environment* Δ, which means that each end-point type is dual with other end-point types.

Definition 4.1. *A session environment Δ is* coherent for the session s *(notation* $\mathsf{co}(\Delta, s)$*) if $s[\mathtt{p}] : T \in \Delta$ and $T \upharpoonright \mathtt{q} \neq \mathsf{end}$ imply $s[\mathtt{q}] : T' \in \Delta$ and $T \upharpoonright \mathtt{q} \bowtie T' \upharpoonright \mathtt{p}$. A session environment Δ is* coherent *if it is coherent for all sessions which occur in it.*

The definitions for $T \upharpoonright \mathtt{q}$ and $\bowtie$ are defined in Appendix B. Intuitively, $T \upharpoonright \mathtt{q}$ is a projection of T onto $\mathtt{q}$ which is similarly defined as $G \upharpoonright \mathtt{q}$; and $T \upharpoonright \mathtt{q} \bowtie T' \upharpoonright \mathtt{p}$ means actions in T onto $\mathtt{q}$ and actions in T' onto $\mathtt{p}$ are dual (i.e. input matches output, and branching matches with selections, and vice verse). Note that two projections of a same global type are always dual: let G a global type and $\mathtt{p}, \mathtt{q} \in G$ with $\mathtt{p} \neq \mathtt{q}$. Then $(G \upharpoonright \mathtt{p}) \upharpoonright \mathtt{q} \bowtie (G \upharpoonright \mathtt{q}) \upharpoonright \mathtt{p}$, i.e. session environments corresponding to global type are always coherent.

Using the above notion we can state type preservation under reductions as follows:

Theorem 4.8 (Subject Congruence and Reduction)

1. *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \equiv P'$, then $\Gamma \vdash_\Sigma P' \triangleright \Delta$.*
2. *If $\Gamma \vdash_\Sigma P \triangleright \tau$ and $P \longrightarrow^* P'$ with τ coherent, then $\Gamma \vdash_\Sigma P' \triangleright \tau'$ for some τ' such that $\tau \Rightarrow^* \tau'$ with τ' coherent.*

Proof. We only list the crucial cases of the proof of subject reduction: the recursor (where mathematical induction is required), the initialisation, the input and the output. The proof of subject congruence is essentially as the same as that in [3, 23]. Our proof works by induction on the length of the derivation $P \longrightarrow^* P'$. The base case is trivial. We then proceed by a case analysis on the reduction $P \longrightarrow P'$. We omit the hat from principal values and Σ for readability.

Case [ZeroR]. Trivial.

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

$$\Gamma, i : I^-, X : \tau\{i/j\} \vdash Q \triangleright \tau'\{i+1/j\} \tag{10}$$

$$\Gamma \vdash P \triangleright \tau'\{0/i\} \tag{11}$$

$$\Gamma \vdash \Pi j\!:\!I.\tau \blacktriangleright \Pi j\!:\!I.\kappa \tag{12}$$

with $\tau \equiv (\Pi i : I.\tau')\mathrm{n}+1 \equiv \tau'\{\mathrm{n}+1/i\}$ and $\Gamma \models \mathrm{n}+1 : I$. By Substitution Lemma (Lemma 4.2 (1)), noting $\Gamma \models \mathrm{n} : I^-$, we have: $\Gamma, X : \tau\{i/j\}\{\mathrm{n}/i\} \vdash Q\{\mathrm{n}/i\} \triangleright \tau'\{i+1/j\}\{\mathrm{n}/i\}$, which means that

$$\Gamma, X : \tau\{\mathrm{n}/j\} \vdash Q\{\mathrm{n}/i\} \triangleright \tau'\{\mathrm{n}+1/j\} \tag{13}$$

Then there are two cases.

Base Case $\mathrm{n} = 0$*:* By applying Substitution Lemma (Lemma 4.2 (2)) to (13) with (11), we have $\Gamma \vdash Q\{1/i\}\{P/X\} \triangleright \tau'\{1/j\}$.

Inductive Case $\mathrm{n} \geq 1$*:* By the inductive hypothesis on n, we assume: $\Gamma \vdash \mathbf{R}\ P\ \lambda i.\lambda X.Q\ \mathrm{n} \triangleright \tau'\{\mathrm{n}/j\}$. Then by applying Substitution Lemma (Lemma 4.2) to (13) with this hypothesis, we obtain $\Gamma \vdash Q\{\mathrm{n}/i\}\{\mathbf{R}\ P\ \lambda i.\lambda X.Q\ \mathrm{n}/X\} \triangleright \tau'\{\mathrm{n}+1/j\}$.

Case [Init].

$$\bar{a}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y).P \longrightarrow (\nu s)(P\{s[\mathtt{p}_0]/y\} \mid s : \epsilon \mid \bar{a}[\mathtt{p}_1] : s \mid ... \mid \bar{a}[\mathtt{p}_\mathrm{n}] : s)$$

We assume that $\Gamma \vdash_\emptyset \bar{a}[\mathtt{p}_0, .., \mathtt{p}_n](y).P \triangleright \Delta$. Inversion of [TINIT] and [TSUB] gives that $\Delta' \leq \Delta$ and:

$$\forall i \neq 0,\ \Gamma \vdash \mathtt{p}_i \triangleright \mathsf{nat} \tag{14}$$
$$\Gamma \vdash a : \langle G \rangle \tag{15}$$
$$\Gamma \models \mathsf{pid}(G) = \{\mathtt{p}_0..\mathtt{p}_n\} \tag{16}$$
$$\Gamma \vdash P \triangleright \Delta', y : G \upharpoonright \mathtt{p}_0 \tag{17}$$
From (17) and Lemma 4.2 (4), $$\Gamma \vdash P\{s[\mathtt{p}_0]/y\} \triangleright \Delta, s[\mathtt{p}_0] : G \upharpoonright \mathtt{p}_0 \tag{18}$$
From Lemma 4.1 (3a) and $\lfloor$QINIT$\rfloor$, $$\Gamma \vdash_s s : \epsilon \triangleright \emptyset \tag{19}$$
From (14), (15), (16) and [TREQ], $\forall i \neq 0$, $$\Gamma \vdash \bar{a}[\mathtt{p}_i] : s \triangleright s[\mathtt{p}_i] : G \upharpoonright \mathtt{p}_i \tag{20}$$

Then [TPAR] on (18), (19) and (20) gives:

$$\Gamma \vdash P\{s[\mathtt{p}_0]/y\} \mid \bar{a}[\mathtt{p}_1] : s \mid ... \mid \bar{a}[\mathtt{p}_n] : s \triangleright \Delta', s[\mathtt{p}_0] : G \upharpoonright \mathtt{p}_0, ..., s[\mathtt{p}_n] : G \upharpoonright \mathtt{p}_n$$

From $\lfloor$GINIT$\rfloor$ and $\lfloor$GPAR$\rfloor$, we have:

$$\Gamma \vdash_s P\{s[\mathtt{p}_0]/y\} \mid \bar{a}[\mathtt{p}_1] : s \mid ... \mid \bar{a}[\mathtt{p}_n] : s \mid s : \epsilon \triangleright \Delta', s[\mathtt{p}_0] : G \upharpoonright \mathtt{p}_0, ..., s[\mathtt{p}_n] : G \upharpoonright \mathtt{p}_n$$

From Lemma 4.7 we know that $\mathsf{co}((s[\mathtt{p}_0] : G \upharpoonright \mathtt{p}_0, ..., s[\mathtt{p}_n] : G \upharpoonright \mathtt{p}_n), s)$. We can then use $\lfloor$GSRES$\rfloor$ to get:

$$\Gamma \vdash_\emptyset (\nu s)(P\{s[\mathtt{p}_0]/y\} \mid \bar{a}[\mathtt{p}_1] : s \mid ... \mid \bar{a}[\mathtt{p}_n] : s \mid s : \epsilon) \triangleright \Delta'$$

We conclude from [TSUB].

Case [Join].

$$\bar{a}[\mathtt{p}] : s \mid a[\mathtt{p}](y).P \longrightarrow P\{s[\mathtt{p}]/y\}$$

We assume that $\Gamma \vdash \bar{a}[\mathtt{p}] : s \mid a[\mathtt{p}](y).P \triangleright \Delta$. Inversion of [TPAR] and [TSUB] gives that $\Delta = \Delta', s[\mathtt{p}] : T$ and :

$$\Gamma \vdash \bar{a}[\mathtt{p}] : s \triangleright s[\mathtt{p}] : G \upharpoonright \mathtt{p} \tag{21}$$
$$T \geq G \upharpoonright \mathtt{p} \tag{22}$$
$$\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta' \tag{23}$$
By inversion of [TACC] from (23) $$\Gamma \vdash P \triangleright \Delta', y : G \upharpoonright \mathtt{p} \tag{24}$$
From (24) and Lemma 4.2 (4), $$\Gamma \vdash P\{s[\mathtt{p}]/y\} \triangleright \Delta', s[\mathtt{p}] : G \upharpoonright \mathtt{p} \tag{25}$$

We conclude by [TSUB] from (25) and (22).

Case [Send].

$$s[\mathtt{q}]!\langle \mathtt{p}, v\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{q}, \mathtt{p}, v)$$

By inductive hypothesis, $\Gamma \vdash_\Sigma s[\mathtt{q}]!\langle \mathtt{p}, e\rangle; P \mid s : h \triangleright \Delta$ with $\Sigma = \{s\}$ and

$$\Gamma \vdash s[\mathtt{q}]!\langle \mathtt{p}, v\rangle; P \triangleright \Delta_1 \tag{26}$$
$$\Gamma \vdash_{\{s\}} s : h \triangleright \Delta_2 \tag{27}$$

where $\Delta = \Delta_2 * \Delta_1$. From (26), we have

$$\Delta_1 = \Delta_1', s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle; T$$
$$\Gamma \vdash v : S \tag{28}$$
$$\Gamma \vdash P \triangleright \Delta_1', s[\mathtt{q}] : T. \tag{29}$$

Using $\lfloor$QSEND$\rfloor$ on (27) and (28) we derive

$$\Gamma \vdash_{\{s\}} s : h \cdot (\mathtt{q}, \mathtt{p}, v) \triangleright \Delta_2; \{s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle\}. \tag{30}$$

Using $\lfloor$GINIT$\rfloor$ on (29) we derive

$$\Gamma \vdash_{\emptyset} P \triangleright \Delta_1', s[\mathtt{q}] : T \tag{31}$$

and then using $\lfloor$GPAR$\rfloor$ on (31) and (30), we conclude

$$\Gamma \vdash_{\{s\}} P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, v) \triangleright (\Delta_2; \{s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle\}) * (\Delta_1', s[\mathtt{q}] : T).$$

Note that $(\Delta_2; \{s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle\}) * (\Delta_1', s[\mathtt{q}] : T) = \Delta_2 * (\Delta_1', s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle; T)$.

Case [Recv].

$$s[\mathtt{p}]?(\mathtt{q}, x); P \mid s : (\mathtt{q}, \{\mathtt{p}\}, v) \cdot h \longrightarrow P\{v/x\} \mid s : h$$

By inductive hypothesis, $\Gamma \vdash_{\Sigma} s[\mathtt{p}]?(\mathtt{q}, x); P \mid s : (\mathtt{q}, \{\mathtt{p}\}, v) \cdot h \triangleright \Delta$. Then we have $\Sigma = \{s\}$ and

$$\Gamma \vdash s[\mathtt{p}]?(\mathtt{q}, x); P \triangleright \Delta_1 \tag{32}$$
$$\Gamma \vdash_{\{s\}} s : (\mathtt{q}, \mathtt{p}, v) \cdot h \triangleright \Delta_2 \tag{33}$$

where $\Delta = \Delta_2 * \Delta_1$. From (32) we have

$$\Delta_1 = \Delta_1', s[\mathtt{p}] :?\langle \mathtt{q}, S\rangle; T$$
$$\Gamma, x : S \vdash P \triangleright \Delta_1', s[\mathtt{p}] : T \tag{34}$$

From (33) we have

$$\Delta_2 = \{s[\mathtt{q}] : !\langle \mathtt{p}, S'\rangle\} * \Delta_2'$$
$$\Gamma \vdash_{\{s\}} s : h \triangleright \Delta_2' \tag{35}$$
$$\Gamma \vdash v : S'. \tag{36}$$

The coherence of Δ implies $S = S'$. From (34) and (36), together with Substitution lemma, we obtain $\Gamma \vdash P\{v/x\} \triangleright \Delta_1', s[\mathtt{p}] : T$, which implies by rule $\lfloor$GINIT$\rfloor$

$$\Gamma \vdash_{\emptyset} P\{v/x\} \triangleright \Delta_1', s[\mathtt{p}] : T. \tag{37}$$

Using rule $\lfloor$GPAR$\rfloor$ on (37) and (35) we conclude

$$\Gamma \vdash_{\{s\}} P\{v/x\} \mid s : h \triangleright \Delta_2' * (\Delta_1', s[\mathtt{p}] : T).$$

Note that $(\{s[\mathtt{q}] : !\langle \mathtt{p}, S\rangle\} * \Delta_2') * (\Delta_1', s[\mathtt{p}] :?\langle \mathtt{q}, S\rangle; T) \Rightarrow \Delta_2' * (\Delta_1', s[\mathtt{p}] : T)$. □

Note that communication safety [23, Theorem 5.5] and session fidelity [23, Corollary 5.6] are corollaries of the above theorem.

A notable fact is, in the presence of the asynchronous initiation primitive, we can still obtain *progress* in a single multiparty session as in [23, Theorem 5.12], i.e. if a program P starts from one session, the reductions at session channels do not get a stuck. Formally

1. We say P is *simple* if P is typable and derived by $\Gamma \vdash^{\star} P \triangleright \Delta$ where the session typing in the premise and the conclusion of each prefix rule is restricted to at most a singleton. More concretely, (1) we eliminate Δ from [TInit], [TAcc], [TOut], [TIn], [TSel] and [TBra], (2) we delete [TDeleg] and [TRecep], (3) we restrict τ and Δ in [TPRec], [TEq], [TApp], [TRec] and [TVar] contain at most only one session typing, and (4) we set $\Delta = \emptyset$ and Δ' contains at most only one session typing; or vice-versa in [TPar].
2. We say P is *well-linked* when for each $P \longrightarrow^{*} Q$, whenever Q has an active prefix whose subject is a (free or bound) shared channels, then it is always reducible. This condition eliminates the element which can hinder progress is when interactions at shared channels cannot proceed. See [23, § 5] more detailed definitions.

The proof of the following theorem is essentially identical with [23, Theorem 5.12].

Theorem 4.9 (Progress) *If P is well-linked and without any element from the runtime syntax and $\Gamma \vdash^{\star} P \triangleright \emptyset$. Then for all $P \longrightarrow^{*} Q$, either $Q \equiv \mathbf{0}$ or $Q \longrightarrow R$ for some R.*

5 Typing examples

In this section, we give examples of typing derivations for the protocols mentioned in § 1 and § 2.1.

5.1 Repetition example - § 1 (1)

This example illustrates the repetition of a message pattern. The global type for this protocol is $G(n) = \mathtt{foreach}(i < n)\{\mathtt{Alice} \to \mathtt{Bob}\colon\langle\mathsf{nat}\rangle.\mathtt{Bob} \to \mathtt{Carol}\colon\langle\mathsf{nat}\rangle\}$. Following the projection from Figure 10, `Alice`'s end-point projection of $G(n)$ has the following form:

$$
\begin{array}{l}
G(n) \upharpoonright \mathtt{Alice} = \mathbf{R}\ \ \mathsf{end} \\
\qquad\qquad\qquad\quad \lambda i.\lambda \mathbf{x}.\mathsf{if}\ \mathtt{Alice} = \mathtt{Alice} = \mathtt{Bob}\ \mathsf{then}\ (\ldots) \\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ \mathtt{Alice} = \mathtt{Alice}\ \mathsf{then}\ (!\langle \mathtt{Bob}, \mathsf{nat}\rangle; \mathsf{if}\ \mathtt{Alice} = \mathtt{Bob} = \mathtt{Carol}\ \mathsf{then}\ \ldots) \\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ \mathtt{Alice} = \mathtt{Bob}\ \mathsf{then}\ \ldots \\
\qquad\qquad\qquad\quad \mathsf{else}\ \ldots\ n
\end{array}
$$

For readability, we omit from our examples the impossible cases created by the projection algorithm. The number of cases can be automatically trimmed to only keep the ones whose resolutions depend on free variables.

In this case, the projection yields the following local type:

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

Before typing, we first define some abbreviations:

$$\begin{aligned}\texttt{Alice}(n) &= \bar{a}[\texttt{Alice},\texttt{Bob},\texttt{Carol}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X\ n)\\ \Delta(n) &= \{y : (G(n) \upharpoonright \texttt{Alice})\}\\ \Gamma &= n{:}\mathsf{nat}, a{:}\langle G(n)\rangle\end{aligned}$$

Our goal is to prove the typing judgement

$$\Gamma \vdash \texttt{Alice}(n) \rhd \emptyset$$

We start from the leafs.

$$\dfrac{\dfrac{\dfrac{\Gamma, i : I^-, X : \Delta(i) \vdash \mathsf{Env}}{\Gamma, i : I^-, X : \Delta(i) \vdash X \rhd y : \Delta(i)}\,[\textsc{TVar}]}{\Gamma, i : I^-, X : \Delta(i) \vdash y!\langle \texttt{Bob}, e[i]\rangle; X \rhd y :!\langle \texttt{Bob}, \mathsf{nat}\rangle; \Delta(i)}\,[\textsc{TOut}]}{\Gamma, i : I^-, X : \Delta(i) \vdash y!\langle \texttt{Bob}, e[i]\rangle; X \rhd \Delta(i+1)}\,[\textsc{TEq}]$$

The application of the [TEQ] rule is justified by the fact that the types $\Delta(i+1)$ and $y : !\langle \texttt{Bob}, \mathsf{nat}\rangle; (\mathbf{R}\ \mathsf{end}\ \lambda j.\lambda \mathbf{x}.!\langle \texttt{Bob}, \mathsf{nat}\rangle; \mathbf{x}\ i)$ are equivalent: they have the same weak-head normal form (we use the rule $\lfloor$WFBASE$\rfloor$).

Since we have the trivial $\Gamma \vdash \mathbf{0} \rhd \Delta(0)$, we can apply the rules [TAPP] and [TPREC].

$$\dfrac{\dfrac{\Gamma, i : I^-, X : \Delta(i) \vdash y!\langle \texttt{Bob}, e[i]\rangle; X \rhd \Delta(i+1) \quad \Gamma \vdash \mathbf{0} \rhd \Delta(0) \quad \Gamma, i{:}I \vdash \Delta(i) \blacktriangleright \kappa}{\Gamma \vdash (\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X) \rhd \Pi i{:}I.\Delta(i)}\,[\textsc{TPRec}]}{\Gamma \vdash (\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X\ n) \rhd \Delta(n)}\,[\textsc{TApp}]$$

We conclude with [TINIT].

$$\dfrac{\Gamma \vdash a : \langle G(n)\rangle \quad \Gamma \vdash (\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X\ n) \rhd \Delta(n)}{\Gamma \vdash \texttt{Alice}(n) \rhd \emptyset}\,[\textsc{TInit}]$$

$\texttt{Bob}(n)$ and $\texttt{Carol}(n)$ can be similarly typed.

5.2 Sequence example - § 1 (2)

The sequence example consists of n participants organised in the following way (when $n \geq 2$): the starter $\texttt{W}[n]$ sends the first message, the final worker $\texttt{W}[0]$ receives the final message and the middle workers first receive a message and then send another to the next worker. We write below the result of the projection for a participant $\texttt{W}[\texttt{p}]$ (left) and the end-point type that naturally types the processes (right):

$\mathbf{R}$ end $\lambda i.\lambda \mathbf{x}.$ if $\texttt{p} = \texttt{W}[i+1]$ then $!\langle \texttt{W}[i], \mathsf{nat}\rangle; \mathbf{x}$ else if $\texttt{p} = \texttt{W}[i]$ then $?\langle \texttt{W}[i+1], \mathsf{nat}\rangle; \mathbf{x}$ else $\mathbf{x}$ n	if $\texttt{p} = \texttt{W}[n]$ then $!\langle \texttt{W}[n-1], \mathsf{nat}\rangle;$ else if $\texttt{p} = \texttt{W}[0]$ then $?\langle \texttt{W}[1], \mathsf{nat}\rangle;$else if $\texttt{p} = \texttt{W}[i]$ then $?\langle \texttt{W}[i+1], \mathsf{nat}\rangle;!\langle \texttt{W}[i-1], \mathsf{nat}\rangle;$

This example illustrates the main challenge faced by the type checking algorithm. In order to type this example, we need to prove the equivalence of these two types. For any concrete instantiation of p and n, the standard weak head normal form equivalence rule ⌊WFBASE⌋ is sufficient. Proving the equivalence for all p and n requires either (a) to bound the domain I in which they live, and check all instantiations within this finite domain using rule ⌊WFRECF⌋; or (b) to prove the equivalence through the meta-logic rule ⌊WFRECEXT⌋. In case (a), type checking terminates, while case (b) allows to prove stronger properties about a protocol's implementation.

5.3 Ring - Figure 3(a)

The typing of the ring pattern is similar to the one of the sequence. The projection of this global session type for W[p] gives the following local type:

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

On the other hand, user processes can be easily type-checked with an end-point type of the following form:

$$
\begin{array}{ll}
\mathsf{if} & \mathtt{p} = \mathtt{W}[0]\ \mathsf{then}\ !\langle \mathtt{W}[1], \mathsf{nat}\rangle ; ?\langle \mathtt{W}[n], \mathsf{nat}\rangle ; \\
\mathsf{elseif} & \mathtt{p} = \mathtt{W}[n]\ \mathsf{then}\ ?\langle \mathtt{W}[n-1], \mathsf{nat}\rangle ; !\langle \mathtt{W}[0], \mathsf{nat}\rangle ; \\
\mathsf{elseif} & 1 \leq i+1 \leq n-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[i+1] \\
& \mathsf{then}\ ?\langle \mathtt{W}[i], \mathsf{nat}\rangle ; !\langle \mathtt{W}[i+2], \mathsf{nat}\rangle ;
\end{array}
$$

Proving the equivalence between these types is similar as the one the sequence: we rely on rules ⌊WFBASE⌋ and ⌊WFRECF⌋ when the domain of n is bounded, or on the meta-logic rule ⌊WFRECEXT⌋.

5.4 Mesh pattern - Figure 3(b)

The mesh example describes nine different participants behaviours (when $n, m \geq 2$). The participants in the first and last rows and columns, except the corners which have two neighbours, have three neighbours. The other participants have four neighbours. The specifications of the mesh are defined by the communication behaviour of each participant and by the links the participants have with their neighbours. The term below is the result of the projection of the global type for participant p

$$
\begin{array}{l}
\mathbf{R}\ \ (\mathbf{R}\ \mathsf{end} \upharpoonright \mathtt{p}\ \lambda k.\lambda \mathbf{z}.\mathsf{if}\ \mathtt{p} = \mathtt{W}[0][k+1]\ \mathsf{then}\ !\langle \mathtt{W}[0][k], \mathsf{nat}\rangle; \mathbf{z} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[0][k]\ \mathsf{then}\ ?\langle \mathtt{W}[0][k+1], \mathsf{nat}\rangle; \mathbf{z} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathbf{z}\ )m \\
\quad \lambda i.\lambda \mathbf{x}. \\
\quad\ (\mathbf{R}\ \ (\mathsf{if}\ \mathtt{p} = \mathtt{W}[i+1][0]\ \mathsf{then}\ !\langle \mathtt{W}[i][0], \mathsf{nat}\rangle; \mathbf{x} \\
\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i][0]\ \mathsf{then}\ ?\langle \mathtt{W}[i+1][0], \mathsf{nat}\rangle; \mathbf{x} \\
\qquad\quad \mathsf{else}\ \mathbf{x}\ ) \\
\qquad \lambda j.\lambda \mathbf{y}. \\
\qquad\quad \mathsf{if}\ \mathtt{p} = \mathtt{W}[i+1][j+1]\ \mathsf{then}\ !\langle \mathtt{W}[i][j+1], \mathsf{nat}\rangle; \\
\qquad\qquad\quad \mathsf{if}\ \mathtt{p} = \mathtt{W}[i+1][j+1]\ \mathsf{then}\ !\langle \mathtt{W}[i+1][j], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i+1][j]\ \mathsf{then}\ ?\langle \mathtt{W}[i+1][j+1], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\qquad\quad \mathsf{else}\ \mathbf{y} \\
\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i][j+1]\ \mathsf{then}\ ?\langle \mathtt{W}[i+1][j+1], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\qquad\quad \mathsf{if}\ \mathtt{p} = \mathtt{W}[i+1][j+1]\ \mathsf{then}\ !\langle \mathtt{W}[i+1][j], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i+1][j]\ \mathsf{then}\ ?\langle \mathtt{W}[i+1][j+1], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\qquad\quad \mathsf{else}\ \mathbf{y} \\
\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i+1][j+1]\ \mathsf{then}\ !\langle \mathtt{W}[i+1][j], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\quad \mathsf{elseif}\ \mathtt{p} = \mathtt{W}[i+1][j]\ \mathsf{then}\ ?\langle \mathtt{W}[i+1][j+1], \mathsf{nat}\rangle; \mathbf{y} \\
\qquad\quad \mathsf{else}\ \mathbf{y} \\
\quad\ m) \\
n
\end{array}
$$

From Figure 3(b), the user would design the end-point type as follows:

$$
\begin{array}{l}
\mathsf{if}\quad \mathtt{p} = \mathtt{W}[n][m]\ \mathsf{then}\ !\langle \mathtt{W}[n-1][m], \mathsf{nat}\rangle; !\langle \mathtt{W}[n][m-1], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ \mathtt{p} = \mathtt{W}[n][0]\ \mathsf{then}\ ?\langle \mathtt{W}[n][1], \mathsf{nat}\rangle; !\langle \mathtt{W}[n-1][0], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ \mathtt{p} = \mathtt{W}[0][m]\ \mathsf{then}\ ?\langle \mathtt{W}[1][m], \mathsf{nat}\rangle; !\langle \mathtt{W}[0][m-1], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ \mathtt{p} = \mathtt{W}[0][0]\ \mathsf{then}\ ?\langle \mathtt{W}[1][0], \mathsf{nat}\rangle; ?\langle \mathtt{W}[0][1], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ 1 \leq k+1 \leq m-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[n][k+1] \\
\qquad \mathsf{then}\ ?\langle \mathtt{W}[n][k+2], \mathsf{nat}\rangle; !\langle \mathtt{W}[n-1][k+1], \mathsf{nat}\rangle; !\langle \mathtt{W}[n][k], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ 1 \leq k+1 \leq m-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[0][k+1] \\
\qquad \mathsf{then}\ ?\langle \mathtt{W}[1][k+1], \mathsf{nat}\rangle; ?\langle \mathtt{W}[0][k+2], \mathsf{nat}\rangle; !\langle \mathtt{W}[0][k], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ 1 \leq i+1 \leq n-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[i+1][m] \\
\qquad \mathsf{then}\ ?\langle \mathtt{W}[i+2][m], \mathsf{nat}\rangle; !\langle \mathtt{W}[i][m], \mathsf{nat}\rangle; !\langle \mathtt{W}[i+1][m-1], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ 1 \leq i+1 \leq n-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[i+1][0] \\
\qquad \mathsf{then}\ ?\langle \mathtt{W}[i+2][0], \mathsf{nat}\rangle; ?\langle \mathtt{W}[i+1][1], \mathsf{nat}\rangle; !\langle \mathtt{W}[i][0], \mathsf{nat}\rangle; \\
\mathsf{elseif}\ 1 \leq i+1 \leq n-1\ \mathsf{and}\ 1 \leq j+1 \leq m-1\ \mathsf{and}\ \mathtt{p} = \mathtt{W}[i+1][j+1] \\
\qquad \mathsf{then}\ ?\langle \mathtt{W}[i+2][j+1], \mathsf{nat}\rangle; ?\langle \mathtt{W}[i+1][j+2], \mathsf{nat}\rangle; !\langle \mathtt{W}[i][j+1], \mathsf{nat}\rangle; !\langle \mathtt{W}[i+1][j], \mathsf{nat}\rangle;
\end{array}
$$

Each case denotes a different local behaviour in the mesh pattern. We present the following meta-logic proof of the typing equivalence through $\lfloor$WFRECEXT$\rfloor$ in the two cases of the top-left corner and bottom row, in order to demonstrate how our system types the mesh session. The other cases are left to the reader.

Let $T[\mathtt{p}][n][m]$ designate the first original type and $T'[\mathtt{p}][n][m]$ the second type. To prove the type equivalence, we want to check that for all $\mathrm{n}, \mathrm{m} \geq 2$ and $\mathtt{p}$, we have:

$$(\prod n.\prod m.T[\mathtt{p}][n][m])\mathrm{nm} \longrightarrow^* T_{\mathrm{n,m}} \not\longrightarrow \text{ iff } (\prod n.\prod m.T'[\mathtt{p}][n][m])\mathrm{nm} \longrightarrow^* T_{\mathrm{n,m}} \not\longrightarrow.$$

For $\mathtt{p} = \mathtt{W}[n][m]$, which implements the top-left corner, the generator type reduces several steps and gives the end-point type $!\langle \mathtt{W}[\mathrm{n}-1][\mathrm{m}], \mathsf{nat}\rangle; !\langle \mathtt{W}[\mathrm{n}][\mathrm{m}-1], \mathsf{nat}\rangle; \mathbf{0}$,

which is the same to the one returned in one step by the case analysis of the type built by the programmer. For $\mathtt{p} = \mathtt{W}[0][k+1]$, we analyse the case where $1 \leq k+1 \leq \mathrm{m}-1$. The generator type returns the end-point type $?\langle \mathtt{W}[1][k+2], \mathsf{nat}\rangle; ?\langle \mathtt{W}[0][k+2], \mathsf{nat}\rangle; !\langle \mathtt{W}[0][k], \mathsf{nat}\rangle;$.One can observe that the end-point type returned for $\mathtt{p} = \mathtt{W}[0][k+1]$ in the type of the programmer is the same as the one returned by the generator. Similarly for all the other cases.

By [TOUT, TIN], we have:

$$a : \langle G\rangle \vdash y!\langle \mathtt{W}[n-1][m], f(n-1,m)\rangle; y!\langle \mathtt{W}[n][m-1], f(n,m-1)\rangle; \mathbf{0} \rhd \Delta, y : G \upharpoonright \mathtt{p}_{\text{top-left}}$$

$$a : \langle G\rangle \vdash y?(\mathtt{W}[1][k+1], z_1); y?(\mathtt{W}[0][k+2], z_2); y!\langle \mathtt{W}[0][k], f(0,k)\rangle; \mathbf{0} \rhd \Delta', y : G \upharpoonright \mathtt{p}_{\text{bottom}}$$

where $G \upharpoonright \mathtt{p}$ is obtained from the type above.

5.5 FFT example - Figure 8

We prove type-safety and deadlock-freedom for the FFT processes. Let P_{fft} be the following process:

$$\begin{array}{l} \Pi n.(\nu a)(\mathbf{R}\ \bar{a}[\mathtt{p}_0..\mathtt{p}_{2^n-1}](y).P(2^n-1, \mathtt{p}_0, x_{\overline{\mathtt{p}_0}}, y, r_{\mathtt{p}_0}) \\ \quad \lambda i.\lambda Y.(\bar{a}[\mathtt{p}_{i+1}](y).P(i+1, \mathtt{p}_{i+1}, x_{\overline{\mathtt{p}_{i+1}}}, y, r_{\mathtt{p}_{i+1}}) \mid Y)\ 2^n - 1) \end{array}$$

As we reasoned above, each $P(n, \mathtt{p}, x_{\overline{\mathtt{p}}}, y, r_{\mathtt{p}})$ is straightforwardly typable by an end-point type which can be proven to be equivalent with the one projected from the global type G from Figure 8(c). Automatically checking the equivalence for all n is not easy though: we need to rely on the finite domain restriction using $\lfloor$WFRECF$\rfloor$ or to rely on a meta-logic proof through $\lfloor$WFRECEXT$\rfloor$. The following theorem says once P_{fft} is applied to a natural number m, its evaluation always terminates with the answer at $r_{\mathtt{p}}$.

Theorem 5.1 (Type safety and deadlock-freedom of FFT) *For all* m, $\emptyset \vdash P_{\mathit{fft}}\ \mathrm{m} \rhd \emptyset$*; and if* $P_{\mathit{fft}}\ \mathrm{m} \longrightarrow^* Q$*, then* $Q \longrightarrow^* (r_0!\langle 0, X_0\rangle \mid \ldots \mid r_{2^{\mathrm{m}}-1}!\langle 0, X_{2^{\mathrm{m}}-1}\rangle)$ *where the* $r_{\mathtt{p}}!\langle 0, X_{\mathtt{p}}\rangle$ *are the actions sending the final values* $X_{\mathtt{p}}$ *on external channels* $r_{\mathtt{p}}$.

Proof. For the proof, we first show $P_{\text{fft}}\ \mathrm{m}$ is typable by a single, multiparty dependent session (except the answering channel at $r_{\mathtt{p}}$). Then the result is immediate as a corollary of progress (Theorem 4.9).

To prove that the processes are typable against the given global type, we start from the end-point projection.

We assume index n to be a parameter as in Figure 8. The main loop is an iteration over the n steps of the algorithm. Forgetting for now the content of the main loop, the generic projection for machine $\mathtt{p}$ has the following skeleton:

$$\begin{array}{l} \Pi n.(\mathbf{R}\ (\mathbf{R}\ \mathsf{end}\ \lambda l.\lambda \mathbf{x}.(\ldots)\ n) \\ \lambda k.\lambda \mathbf{u}. \\ \quad \mathsf{if}\ \mathtt{p} = k\ \mathsf{then}\ !\langle k, U\rangle; ?\langle k, U\rangle; \mathbf{u}\ \mathsf{else}\ \mathbf{u}) \\ 2^n \end{array}$$

A simple induction gives us through $\lfloor$WFRECEXT$\rfloor$ the equivalent type:

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

We now consider the inner loops. The generic projection gives:

$\ldots$
$(\mathbf{R}\ \mathbf{x}\ \lambda i.\lambda \mathbf{y}.$
$(\mathbf{R}\ \mathbf{y}\ \lambda j.\lambda \mathbf{z}.$
$\mathsf{if}\ \mathtt{p} = i * 2^{n-l} + 2^{n-l-1} + j = i * 2^{n-l} + j\ \mathsf{then}\ \ldots$
$\mathsf{else\ if}\ \mathtt{p} = i * 2^{n-l} + 2^{n-l-1} + j\ \mathsf{then}\ !\langle i * 2^{n-l} + j, U\rangle; \ldots$
$\mathsf{else\ if}\ \mathtt{p} = i * 2^{n-l} + j\ \mathsf{then}\ ?\langle i * 2^{n-l} + 2^{n-l-1} + j, U\rangle; \ldots$
$\mathsf{else\ if}\ \ldots\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$
$)\ 2^{n-l-1}$
$)\ 2^{l}$
$\ldots$

An induction over $\mathtt{p}$ and some simple arithmetic over binary numbers gives us through $\lfloor$WFRECEXT$\rfloor$ the only two branches that can be taken:

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

The first branch corresponds to the upper part of the butterfly while the second one corresponds to the lower part. For programming reasons (as seen in the processes, the natural implementation include sending a first initialisation message with the x_k value), we want to shift the self-receive $?\langle \mathtt{p}, U\rangle;$ from the initialisation to the beginning of the loop iteration at the price of adding the last self-receive to the end: $?\langle \mathtt{p}, U\rangle; \mathsf{end}$. The resulting equivalent type up to $\equiv$ is:

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

From this end-point type, it is straightforward to type and implement the processes defined in Figure 8(d) in § 2.6. Hence we conclude the proof. □

5.6 Web Service

This section demonstrates the expressiveness of our type theory. We program and type a real-world Web service usecase: Quote Request (C-U-002) is the most complex scenario described in [36], the public document authored by the W3C Choreography Description Language Working Group [39].

Quote Request usecase The usecase is described below (as published in [36]). A buyer interacts with multiple suppliers who in turn 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.

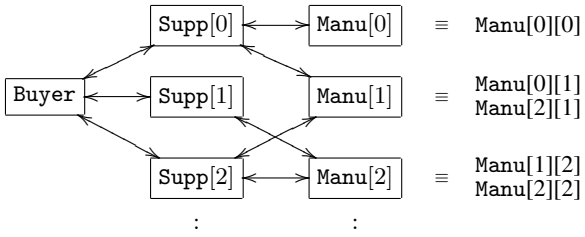


Fig. 19. The Quote Request usecase (C-U-002) [36]

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 19, may seem simple, but it contains many challenges. The Requirements in Section 3.1.2.2 of [36] 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, $\texttt{Buyer}$ stands for the buyer, $\texttt{Supp}[i]$ for a supplier, and $\texttt{Manu}[j]$ for a manufacturer. Then we alias manufacturers by $\texttt{Manu}[i][j]$ to identify that $\texttt{Manu}[j]$ is connected to $\texttt{Supp}[i]$ (so a single $\texttt{Manu}[j]$ can have multiple aliases $\texttt{Manu}[i'][j]$, see figure 19). Then, using the idioms presented in § 1, STEP 1 is defined as:

$$G_1 = \texttt{foreach}(iI)\{\texttt{Buyer} \to \texttt{Supp}[i] : \langle \mathsf{Quote} \rangle.\mathsf{end}\}$$

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

$$G_2 = \texttt{foreach}(i : I)\{G_2[i],\ \texttt{Supp}[i] \to \texttt{Buyer} : \langle \mathsf{Quote} \rangle.\mathsf{end}\}$$
$$G_2[i] = \texttt{foreach}(j : J_i)\{\ \texttt{Supp}[i] \to \texttt{Manu}[i][j] : \langle \mathsf{Item} \rangle.\ \texttt{Manu}[i][j] \to \texttt{Supp}[i] : \langle \mathsf{Quote} \rangle.\mathsf{end}\}$$

$G_2[i]$ 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.

$$\begin{array}{l} G_3 = \mathbf{R}\ \ \mathbf{t}\ \ \lambda i.\lambda \mathbf{y}.\texttt{Buyer} \to \texttt{Supp}[i] : \{ \\ \quad \mathsf{ok} : \quad \mathsf{end} \\ \quad \mathsf{modify} : \texttt{Buyer} \to \texttt{Supp}[i] : \langle \mathsf{Quote} \rangle \\ \qquad \texttt{Supp}[i] \to \texttt{Buyer} : \{\ \mathsf{ok} : \quad \mathsf{end} \\ \qquad\qquad\qquad\qquad\quad \mathsf{retryStep3} : \mathbf{y} \\ \qquad\qquad\qquad\qquad\quad \mathsf{reject} : \quad \mathsf{end}\}\}\ i \end{array}$$

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 $G_{23} = \mu \mathbf{t}.G_2,\ G_3$ and assuming $I = [0..i]$, the full global type is

$$\lambda i.\lambda \tilde{J}.G_1, G_{23}$$

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

End-point types We show the end-point 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:

$$G_1 \upharpoonright \texttt{Supp}[\mathrm{n}] = ?\langle \texttt{Buyer}, \mathsf{Quote} \rangle$$
$$G_2 \upharpoonright \texttt{Supp}[\mathrm{n}] = \texttt{foreach}(j : J_i)\{!\langle \texttt{Manu}[\mathrm{n}][j], \mathsf{Item} \rangle; ?\langle \texttt{Manu}[\mathrm{n}][j], \mathsf{Quote} \rangle\}; !\langle \texttt{Buyer}, \mathsf{Quote} \rangle$$

For $G_3 \upharpoonright \texttt{Supp}[\mathrm{n}]$, 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:

$$\begin{array}{l} \&\langle \texttt{Buyer}, \{\ \mathsf{ok} : \quad \mathsf{end} \\ \quad \mathsf{modify} : ?\langle \texttt{Buyer}, \mathsf{Quote} \rangle; \oplus \langle \texttt{Buyer}, \{ \\ \qquad\qquad\qquad\quad \mathsf{ok} : \quad \mathsf{end} \\ \qquad\qquad\qquad\quad \mathsf{retryStep3} : T \\ \qquad\qquad\qquad\quad \mathsf{reject} : \quad \mathsf{end}\}\rangle\}\rangle \end{array}$$

where T is a type for the invocation from $\texttt{Buyer}$:

$$\begin{array}{l} \mathsf{if}\ \mathrm{n} \leq i\ \mathsf{then}\ \&\langle \texttt{Buyer}, \{\mathsf{closed} : \mathsf{end},\ \mathsf{retryStep3} : \mathbf{t}\}\rangle \\ \mathsf{elseif}\ i = \mathrm{n}\ \mathsf{then}\ \mathbf{t} \end{array}$$

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

$$G_1 \upharpoonright \texttt{Supp}[\mathrm{n}], \mu\mathbf{t}.G_2 \upharpoonright \texttt{Supp}[\mathrm{n}],\ G_3 \upharpoonright \texttt{Supp}[\mathrm{n}])$$

Following this specification, the projections can be implemented in various end-point languages (such as CDL or BPEL).

6 Conclusion and related work

This paper studies a parameterised multiparty session type theory which combines three well-known theories: indexed dependent types [40], dependent types with recursors [31] and multiparty session types [3, 23]. The resulting typing system is decidable (under an appropriate assumption about the arithmetics of indices). It offers great expressive powers to describe complex communication topologies and guarantees safety properties of processes running under such topologies. We have explored the impact of parameterised type structures for communications through implementations of the above web service usecases and of several parallel algorithms in Java with session types [24, 25], including the Jacobi method (with sequence and mesh topologies) and the FFT [26, 32, 34]. We observe (1) a clear coordination of the communication behaviour of each party with the construction of the whole multiparty protocol, thus reducing programming errors and ensuring deadlock-freedom; and (2) a performance benefit against the original binary session version, reducing the overhead of multiple binary session establishments (see also [18, 26, 32]). Full implementation and integration of our theory into [4, 24, 25] is on-going work.

6.1 Related Work

We focus on the works on dependent types and other typed process calculi which are related to multiparty session types; for further comparisons of session types with other service-oriented calculi and behaviour typing systems, see [17] for a wide ranging survey of the related literature.

Dependent types The first use of primitive recursive functionals for dependent types is in Nelson's $\mathcal{T}^\pi$ [31] for the λ-calculus, which is a finite representation of $\mathcal{T}^\infty$ by Tait and Martin Löf [28, 38]. $\mathcal{T}^\pi$ can type functions previously untypable in ML, and the finite representability of dependent types makes it possible to have a type-reconstruction algorithm. We also use the ideas from the DML's dependent typing system in [1, 40] where type dependency is only allowed for index sorts, so that 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 sequences of global and end-point types and processes described by the primitive 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 on the 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. [21]), and channels (e.g. [41]) 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 [7] where effects can appear both in types and processes. None of these works investigate families of 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 [31]).

Types and contracts for multiparty interactions The first papers on multiparty session types were [6] and [23]. 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. Since the first work [23] was proposed, this theory has been used in the different contexts such as distributed protocol implementation and optimisation [37], security [4, 9], design by contract [5], parallel algorithms [32, 42], web services [42], multicore programming [43], an advanced progress guarantee [3], messaging optimisation [30], structured exceptions [10], buffer and channel size analysis for multiparty interactions [15], and medical guidelines [33], some of which initiated industrial collaborations, cf. [22]. Our typing system can be smoothly integrated with other works as no changes to the runtime typing components have been made and the expressiveness has been greatly improved.

The work [11] presented an *executable global processes* for Web interactions based on the binary session types. Our work provides flexible, programmable global descriptions as *types*, offering a progress for parameterised multiparty session, which is not ensured in [11].

Recent formalisms for typing multiparty interactions include [8, 13]. These works treat different aspects of dynamic session structures. *Contracts* [13] can type more processes than session types, thanks to the flexibility of process syntax for describing protocols. However, typable processes themselves in [13] may not always satisfy the properties of session types such as progress: it is proved later by checking whether the type meets a certain form. Hence proving progress with contracts effectively requires an exploration of all possible paths (interleaving, choices) of a protocol. The most complex example of [13, § 3] (a group key agreement protocol from [2]), which is typed as π-processes with delegations, can be specified and articulated by a single parameterised global session type as:

$$\begin{aligned} &\Pi n : I.(\texttt{foreach}(i < n)\{\mathtt{W}[n-i] \to \mathtt{W}[n-i+1] : \langle \mathsf{nat} \rangle\}; \\ &\qquad \texttt{foreach}(i < n)\{\mathtt{W}[n-i] \to \mathtt{W}[n+1] : \langle \mathsf{nat} \rangle.\mathtt{W}[n+1] \to \mathtt{W}[n-i] : \langle \mathsf{nat} \rangle\}) \end{aligned}$$

Once the end-point process conforms to this specification, we can automatically guarantee communication safety and progress.

Conversation Calculus [8] supports the dynamic joining and leaving of participants. We also introduced a dynamic role-based multiparty session type discipline [16] where

an arbitrary number of participants can participate to a running session via a universal polling operator. Though the formalism in § 2.4 can operationally capture such dynamic features, the aim of the present work is *not* the type-abstraction of dynamic interaction patterns. Our purpose is to capture, in a single type description, a family of protocols over arbitrary number of participants, to be instantiated at runtime. The parameterisation gives freedom not possible with previous session types: once typed, a parametric process is ensured that its arbitrary well-typed instantiations, in terms of both topologies and process behaviours, satisfy the safety and progress properties of typed processes. Parameterisation, composition and repetition are common idioms in parallel algorithms and choreographic/conversational interactions, all of which are uniformly treatable in our dependent type theory. Here types offer a rigorous structuring principle which can economically abstract rich interaction structures, including parameterised ones.

References

1. D. Aspinall and M. Hofmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT, 2005.
2. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 17–26, New York, NY, USA, 1998. ACM.
3. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
4. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.
5. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
6. E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.
7. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.
8. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
9. S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session Types for Access and Information Flow Control. In *CONCUR'10*, volume 6269 of *LNCS*, pages 237–252. Springer, 2010.
10. S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. In K. Lodaya and M. Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 338–351, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
11. 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.
12. 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.

13. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, number 5710 in LNCS, pages 211–228, 2009.
14. 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.
15. P.-M. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010. Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/multianalysis`.
16. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*. ACM, 2011. To appear, Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/dynamics`.
17. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and Session Types: an Overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
18. Online Appendix. `http://www.doc.ic.ac.uk/~yoshida/dependent/`.
19. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
20. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1989.
21. M. Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
22. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, LNCS. Springer, 2011. To appear.
23. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008. Full version at `http://www.doc.ic.ac.uk/~yoshida/multiparty`.
24. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-Safe Eventful Sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.
25. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
26. Y. Kryftis. *Session-based Programming for Massage-Passing-based Parallel Algorithms*. Master's thesis, Imperial College London, 2009. `http://www.doc.ic.ac.uk/~yk208/project/project.html`.
27. F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1991.
28. P. Martin-Lőf. Infinite terms and a system of natural deduction. In *Compositio Mathematica*, pages 93–103. Wolters-Noordhoof, 1972.
29. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
30. 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.
31. N. Nelson. Primitive recursive functionals with dependent types. In *MFPS*, volume 598 of *LNCS*, pages 125–143, 1991.
32. N. Ng. *High performance parallel design based on session programming*. Master thesis, Department of Computing, Imperial College London, 2010. `http://www.doc.ic.ac.uk/~cn06/individual-project/`.
33. L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sumtypes. Technical Report 8, Department of Computing, Imperial College London, 2009. Extended abstract will appear in Express'10. See Apims Project for an implementaion at: `http://www.thelas.dk/index.php/apims`.
34. O. Pernet, N. Ng, R. Hu, N. Yoshida, and Y. Kryftis. Safe Parallel Programming with Session Java. Technical Report 14, Department of Computing, Imperial College London, 2010.
35. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

36. Web Services Choreography Requirements (No. 11). `http://www.w3.org/TR/ws-chor-reqs`.
37. K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, volume 6116 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2010.
38. W. W. Tait. Infinitely long terms of transfinite type. In *Formal Systems and Recursive Functions*, pages 177–185. North Holland, 1965.
39. Web Services Choreography Working Group. Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
40. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.
41. N. Yoshida. Channel dependent types for higher-order mobile processes. In *POPL '04*, pages 147–160. ACM Press, 2004.
42. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCs*, volume 6014 of *LNCS*, pages 128–145, 2010.
43. N. Yoshida, V. T. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *FMCO*, volume 5751 of *LNCS*, pages 226–246. Springer, 2009.

A Kinding and typing rules

In this Appendix section, we give the definitions of kinding rules and typing rules that were omitted in the main sections.

A.1 Kinding and subtyping

Figure 20 defines the kinding rules for local types. Figure 21 presents the subtyping rules which are used for typing runtime processes. The rules for the type isomorphism can be given by replacing $\leq$ by $\approx$.

B Typing system for runtime processes

This appendix defines a typing system for runtime processes (which contain queues). Most of the definitions are from [3].

Message	$\mathtt{T}$ ::=	$!\langle \hat{\mathtt{p}}, U\rangle$	*message send*
	$\mid$	$\oplus\langle \hat{\mathtt{p}}, l\rangle$	*message selection*
	$\mid$	$\mathtt{T}; \mathtt{T}'$	*message sequence*
Generalised	T ::=	T	*session*
	$\mid$	$\mathtt{T}$	*message*
	$\mid$	$\mathtt{T}; T$	*continuation*

Message types are the types for queues: they represent the messages contained in the queues. The *message send type* $!\langle \hat{\mathtt{p}}, U\rangle$ expresses the communication to $\hat{\mathtt{p}}$ of a value or of a channel of type U. The *message selection type* $\oplus\langle \hat{\mathtt{p}}, l\rangle$ represents the communication to participant $\hat{\mathtt{p}}$ of the label l and $\mathtt{T}; \mathtt{T}'$ represents sequencing of message types (we

$$\frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \Gamma \vdash T \blacktriangleright \mathsf{Type} \quad \Gamma \vdash U \blacktriangleright \mathsf{Type} \text{ or } \mathsf{Type}}{\Gamma \vdash !\langle \mathtt{p}, U\rangle; T \blacktriangleright \mathsf{Type}}\ [\textsc{KLOut}]$$

$$\frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \Gamma \vdash T \blacktriangleright \mathsf{Type} \quad \Gamma \vdash U \blacktriangleright \mathsf{Type} \text{ or } \mathsf{Type}}{\Gamma \vdash ?\langle \mathtt{p}, U\rangle; T \blacktriangleright \mathsf{Type}}\ [\textsc{KLIn}]$$

$$\frac{\Gamma \vdash T \blacktriangleright \Pi i{:}I.\kappa \quad \Gamma \models \mathtt{i} : I}{\Gamma \vdash T\ \mathtt{i} \blacktriangleright \kappa\{\mathtt{i}/i\}}\ [\textsc{KLApp}] \qquad \frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \forall k \in K, \Gamma \vdash T_k \blacktriangleright \mathsf{Type}}{\Gamma \vdash \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \blacktriangleright \mathsf{Type}}\ [\textsc{KLSel}]$$

$$\frac{\Gamma \vdash \mathtt{p} \rhd \mathsf{nat} \quad \forall k \in K, \Gamma \vdash T_k \blacktriangleright \mathsf{Type}}{\Gamma \vdash \&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \blacktriangleright \mathsf{Type}}\ [\textsc{KLBra}]$$

$$\frac{\Gamma \vdash T \blacktriangleright \kappa\{0/j\} \quad \Gamma, i : I^- \vdash T' \blacktriangleright \kappa\{i+1/j\}}{\Gamma \vdash \mathbf{R}\ T\ \lambda i{:}I^-.\lambda \mathbf{x}.T' \blacktriangleright \Pi j{:}I.\kappa}\ [\textsc{KLRcr}]$$

$$\frac{\Gamma \vdash \kappa}{\Gamma \vdash \mathbf{x} \blacktriangleright \kappa}\ [\textsc{KVar}] \qquad \frac{\Gamma \vdash T \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mu \mathbf{x}.T \blacktriangleright \mathsf{Type}}\ [\textsc{KLRec}] \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \blacktriangleright \mathsf{Type}}\ [\textsc{KLEnd}]$$

Fig. 20. Kinding rules for local types

$$\frac{\Gamma \vdash T \leq T'}{\Gamma \vdash !\langle \mathtt{p}, U\rangle; T \leq !\langle \mathtt{p}, U\rangle; T'}\ \lfloor \textsc{TSubOut} \rfloor \qquad \frac{\Gamma \vdash T \leq T'}{\Gamma \vdash ?\langle \mathtt{p}, U\rangle; T \leq ?\langle \mathtt{p}, U\rangle; T'}\ \lfloor \textsc{TSubIn} \rfloor$$

$$\frac{\forall k \in K \subseteq J,\ \Gamma \vdash T_k \leq T'_k}{\Gamma \vdash \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \leq \oplus\langle \mathtt{p}, \{l_j : T'_j\}_{j\in J}\rangle}\ \lfloor \textsc{TSSel}_{\leq} \rfloor$$

$$\frac{\forall k \in J \subseteq K,\ \Gamma \vdash T_k \leq T'_k}{\Gamma \vdash \&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \leq \&\langle \mathtt{p}, \{l_j : T'_j\}_{j\in J}\rangle}\ \lfloor \textsc{TBra}_{\leq} \rfloor$$

$$\frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma, i : I \vdash T'_1 \leq T'_2}{\Gamma \vdash \mathbf{R}\ T_1\ \lambda i{:}I.\lambda \mathbf{x}.T'_1 \leq \mathbf{R}\ T_2\ \lambda i{:}I.\lambda \mathbf{x}.T'_2}\ \lfloor \textsc{TSubPRec} \rfloor$$

$$\frac{\Gamma \vdash T\{\mu \mathbf{x}.T/\mathbf{x}\} \leq T'}{\Gamma \vdash \mu \mathbf{x}.T \leq T'}\ \lfloor \textsc{TLSubRec} \rfloor \qquad \frac{\Gamma \vdash T' \leq T\{\mu \mathbf{x}.T/\mathbf{x}\}}{\Gamma \vdash T' \leq \mu \mathbf{x}.T}\ \lfloor \textsc{TRSubRec} \rfloor$$

$$\frac{\Gamma \vdash T \leq T' \quad \Gamma \models \mathtt{i} : I = \mathtt{i}' : I}{\Gamma \vdash T\ \mathtt{i} \leq T'\ \mathtt{i}'}\ \lfloor \textsc{TSubProj} \rfloor$$

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \leq \mathsf{end}}\ \lfloor \textsc{TSubEnd} \rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathbf{x} \leq \mathbf{x}}\ \lfloor \textsc{TSubRVar} \rfloor$$

Fig. 21. Subtyping

assume associativity for ;). For example $\oplus\langle 1, \mathsf{ok}\rangle$ is the message type for the message $(2, 1, \mathsf{ok})$. A *generalised type* is either a session type, or a message type, or a message type followed by a session type. Type $\mathtt{T}; T$ represents the continuation of the type $\mathtt{T}$ associated to a queue with the type T associated to a pure process. An example of generalised type is $\oplus\langle 1, \mathsf{ok}\rangle; !\langle 3, \mathsf{string}\rangle; ?\langle 3, \mathsf{date}\rangle; \mathsf{end}$.

In order to take into account the structural congruence between queues (see Figure 7) we consider message types modulo the equivalence relation $\approx$ induced by the rules shown as follows (with $\natural \in \{!, \oplus\}$ and $Z \in \{U, l\}$):

$$\natural\langle \hat{\mathtt{p}}, Z\rangle;\ \natural'\langle \hat{\mathtt{q}}, Z\rangle; \mathtt{T} \approx\ \natural'\langle \hat{\mathtt{q}}, Z\rangle;\ \natural\langle \hat{\mathtt{p}}, Z\rangle; \mathtt{T} \quad \text{if } \hat{\mathtt{p}} \neq \hat{\mathtt{q}}$$

The equivalence relation on message types extends to generalised types by:

$$\mathtt{T} \approx \mathtt{T}' \text{ implies } \mathtt{T}; \mathsf{T} \approx \mathtt{T}'; \mathsf{T}$$

We say that two session environments Δ and Δ' are equivalent (notation $\Delta \approx \Delta'$) if $c : \mathsf{T} \in \Delta$ and $\mathsf{T} \neq \mathsf{end}$ imply $c : \mathsf{T}' \in \Delta'$ with $\mathsf{T} \approx \mathsf{T}'$ and vice versa. This equivalence relation is used in rule $\lfloor$EQUIV$\rfloor$ (see Figure 22).

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\emptyset} P \triangleright \Delta}\ \lfloor \textsc{GInit} \rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Delta \approx \Delta'}{\Gamma \vdash_{\Sigma} P \triangleright \Delta'}\ \lfloor \textsc{Equiv} \rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash_{\Sigma} P \triangleright \Delta'}\ \lfloor \textsc{Subs} \rfloor$$

$$\frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Gamma \vdash_{\Sigma'} Q \triangleright \Delta' \quad \Sigma \cap \Sigma' = \emptyset}{\Gamma \vdash_{\Sigma \cup \Sigma'} P \mid Q \triangleright \Delta * \Delta'}\ \lfloor \textsc{GPar} \rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \mathsf{co}(\Delta, s)}{\Gamma \vdash_{\Sigma \setminus s} (\nu s) P \triangleright \Delta \setminus s}\ \lfloor \textsc{GSRes} \rfloor$$

Fig. 22. Run-time process typing

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash_{\{s\}} s : \epsilon \triangleright \emptyset}\ \lfloor \textsc{QInit} \rfloor$$

$$\frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta \qquad \Gamma \vdash v : S}{\Gamma \vdash_{\{s\}} s : h \cdot (\hat{\mathtt{q}}, \hat{\mathtt{p}}, v) \triangleright \Delta, \{s[\hat{\mathtt{q}}] : !\langle \hat{\mathtt{p}}, S\rangle\}}\ \lfloor \textsc{QSend} \rfloor$$

$$\frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta}{\Gamma \vdash_{\{s\}} s : h \cdot (\hat{\mathtt{q}}, \hat{\mathtt{p}}, s'[\hat{\mathtt{p}}']) \triangleright \Delta, s'[\hat{\mathtt{p}}'] : T', \{s[\hat{\mathtt{q}}] : !\langle \hat{\mathtt{p}}, T'\rangle\}}\ \lfloor \textsc{QDeleg} \rfloor$$

$$\frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta \quad j \in K}{\Gamma \vdash_{\{s\}} s : h \cdot (\hat{\mathtt{q}}, \hat{\mathtt{p}}, l_j) \triangleright \Delta, \{s[\hat{\mathtt{q}}] : \oplus\langle \hat{\mathtt{p}}, \{l_k : T_k\}_{k \in K}\rangle\}}\ \lfloor \textsc{QSel} \rfloor$$

Fig. 23. Queue typing

We start by defining the typing rules for single queues, in which the turnstile $\vdash$ is decorated with $\{s\}$ (where s is the session name of the current queue) and the session environments are mappings from channels to message types. The empty queue has empty session environment. Each message adds an output type to the current type of the channel which has the role of the message sender. Figure 23 lists the typing rules for queues, where ; is defined by:

$$\Delta; \{s[\hat{\mathtt{q}}] : \mathtt{T}\} = \begin{cases} \Delta', s[\hat{\mathtt{q}}] : \mathtt{T}'; \mathtt{T} & \text{if } \Delta = \Delta', s[\hat{\mathtt{q}}] : \mathtt{T}', \\ \Delta, s[\hat{\mathtt{q}}] : \mathtt{T} & \text{otherwise.} \end{cases}$$

For example we can derive $\vdash_{\{s\}} s : (3, 1, \mathsf{ok}) \triangleright \{s[1] : \oplus\langle 1, \mathsf{ok}\rangle\}$.

In order to type pure processes in parallel with queues, we need to use generalised types in session environments and further typing rules. Figure 22 lists the typing rules for processes containing queues. The judgement $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ means that P contains the queues whose session names are in Σ. Rule $\lfloor$GINIT$\rfloor$ promotes the typing of a pure process to the typing of an arbitrary process, since a pure process does not contain queues. When two arbitrary processes are put in parallel (rule $\lfloor$GPAR$\rfloor$) we need to require that each session name is associated to at most one queue (condition $\Sigma \cap \Sigma' = \emptyset$). In composing the two session environments we want to put in sequence a message type and a session type for the same channel with role. For this reason we define the composition $*$ between generalised types as:

$$\mathsf{T} * \mathsf{T}' = \begin{cases} \mathsf{T}; \mathsf{T}' & \text{if } \mathsf{T} \text{ is a message type,} \\ \mathsf{T}'; \mathsf{T} & \text{if } \mathsf{T}' \text{ is a message type,} \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ represents failure of typing.

We extend $*$ to session environments as expected:

$$\Delta * \Delta' = \Delta \backslash dom(\Delta') \cup \Delta' \backslash dom(\Delta) \cup \{c : \mathsf{T} * \mathsf{T}' \mid c : \mathsf{T} \in \Delta \ \&\ c : \mathsf{T}' \in \Delta'\}.$$

Note that $*$ is commutative, i.e., $\Delta * \Delta' = \Delta' * \Delta$. Also if we can derive message types only for channels with roles, we consider the channel variables in the definition of $*$ for session environments since we want to get for example $\{y : \mathsf{end}\} * \{y : \mathsf{end}\} = \bot$ (message types do not contains end).

In rule $\lfloor$GSRES$\rfloor$ we require the coherence of the session environment Δ with respect to the session name s to be restricted (notation $\mathsf{co}(\Delta, s)$). This coherence is defined in Definition 4.1 using the notions of projection of generalised types and of duality, introduced respectively in Definitions B.1 and B.2.

Definition B.1. The *projection of the generalised local type* T *onto* $\mathtt{q}$, denoted by $\mathsf{T} \upharpoonright \mathtt{q}$, is defined by:

$$(\,!\langle \hat{\mathtt{p}}, U\rangle; \mathsf{T}') \upharpoonright \mathtt{q} = \begin{cases} !U; \mathsf{T}' \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \hat{\mathtt{p}}, \\ \mathsf{T}' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\oplus\langle \hat{\mathtt{p}}, l\rangle; \mathsf{T}') \upharpoonright \mathtt{q} = \begin{cases} \oplus l; \mathsf{T}' \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \hat{\mathtt{p}}, \\ \mathsf{T}' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(?\langle \mathtt{p}, U\rangle; T) \upharpoonright \mathtt{q} = \begin{cases} ?U; T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p}, \\ T \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle) \upharpoonright \mathtt{q} = \begin{cases} \oplus\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p}, \\ \sqcup_{i\in I} T_i \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle) \upharpoonright \mathtt{q} = \begin{cases} \&\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p}, \\ \sqcup_{i\in I} T_i \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\mu \mathbf{x}.T) \upharpoonright \mathtt{q} = \mu \mathbf{x}.(T \upharpoonright \mathtt{q}) \quad \mathbf{x} \upharpoonright \mathtt{q} = \mathbf{x} \quad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}$$

where $\sqcup_{i\in I} T_i \upharpoonright \mathtt{q}$ is defined as $\sqcup_{i\in I} T_i$ in Definition 3.1 replacing by:

$$\begin{aligned} &\&\langle \{l_k : T_k\}_{i\in I}\rangle \sqcup \&\langle \{l_j : T'_j\}_{j\in J}\rangle = \\ &\quad \&\langle \{l_k : T_k \sqcup T'_k\}_{k\in K\cap J} \cup \{l_k : T_k\}_{k\in K\setminus J} \cup \{l_j : T'_j\}_{j\in J\setminus K}\rangle \end{aligned}$$

Definition B.2. *The* duality relation *between projections of generalised types is the minimal symmetric relation which satisfies:*

$$\begin{gathered} \mathsf{end} \bowtie \mathsf{end} \qquad \mathbf{x} \bowtie \mathbf{x} \qquad T \bowtie T' \implies \mu \mathbf{x}.T \bowtie \mu \mathbf{x}.T' \\ \mathsf{T} \bowtie T \implies !U; \mathsf{T} \bowtie ?U; T \\ \forall i \in I \; T_i \bowtie T'_i \implies \oplus\{l_i : T_i\}_{i\in I} \bowtie \&\{l_i : T'_i\}_{i\in I} \\ \exists i \in I \; l = l_i \;\&\; \mathsf{T} \bowtie T_i \implies \oplus l; \mathsf{T} \bowtie \&\{l_i : T_i\}_{i\in I} \end{gathered}$$