Parameterised Multiparty Session Types

Nobuko Yoshida, Pierre-Malo Deniélou, 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. 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 [5, 6, 12], 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 paper introduces a robust dependent type theory which can statically ensure communication-safe, deadlock-free process interactions which follow parameterised multiparty protocols.

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 [5, 12, 16], as follows (below end denotes protocol termination):

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

Upon agreement on G_1 as a specification for `Alice` and `Bob`, each program can be implemented separately. For type-checking, G_1 is *projected* into end-point session types: one from `Alice`'s point of view, $!\langle \mathtt{Bob}, \mathsf{nat} \rangle$ (output to `Bob` with nat-type), and another from `Bob`'s point of view, $?\langle \mathtt{Alice}, \mathsf{nat} \rangle$ (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 [19]). Consider the type $G_2 \;=\; \mathtt{Bob} \to \mathtt{Carol}\colon \langle \mathsf{nat} \rangle.\mathsf{end}$. The designer may wish to compose G_1 and G_2 together to build a larger protocol:

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

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 dependant product to build a *family of global specifications*, as in:

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

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 run-time 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 $\texttt{foreach}$, see § 2):

$$\Pi n.(\texttt{foreach}(i < n)\{\texttt{W}[i+1] \to \texttt{W}[i] : \langle \mathsf{nat} \rangle\}) \qquad \begin{tikzcd} \boxed{\text{n}} \arrow[r, maps to] & \boxed{\text{n-1}} \arrow[r, maps to] & \cdots \arrow[r] & \boxed{0} \end{tikzcd} \tag{2}$$

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 (2), if $\mathrm{n} \geq 2$, there are three distinct *communication patterns* inhabiting this specification: the initiator (send only), the $\mathrm{n} - 1$ middle workers (receive and send), and the last worker (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). Can we provide an 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.

Contributions of this work

- *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 $\mathcal{T}$ [17] (for expressiveness) and indexed dependent types from [20] (for tractability to control parameters), 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. The resulting static typing system allows decidable type-checking and guarantees type-safety and deadlock-freedom for well-typed processes involved in parameterised multiparty communication protocols (§ 3).
- *Applications* featuring various process topologies, including the complex butterfly network for the parallel FFT algorithm (§ 2.4,3.6). 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 join primitive for session initialisation, applied to Web services use cases [2] (§ 3.6).

The complete formal definition of our system, including proofs and additional material for examples and implementations can be found in the appendices and in [1].

2 Types and processes for parameterised multiparty sessions

2.1 Global types

Global types allow the description of the parameterised conversation scenarios of multiparty sessions as a type signature. Our type syntax integrates three different formulations: (1) global types from [5]; (2) dependent types with primitive recursive combinators based on [17] and (3) parameterised dependent types from a simplified Dependent ML [3, 20].

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

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

Fig. 1. Global types and 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 to follow 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.

The interesting addition is the primitive recursion operator $\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G'$ from Gödel's System $\mathcal{T}$ [11]. Its reduction semantics is given in figure 1. The primitive recursive operator takes as parameters 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 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 sets of predicates $(\mathtt{P}, \mathtt{P}', ..)$. In our case, these are conjunctions of inequalities. op represents first-order operators on indices (such as $+$, $-$, $*$,...). We often omit I and end in our examples. Using $\mathbf{R}$, we can define the product, composition, repetition and test operators (used in § 1):

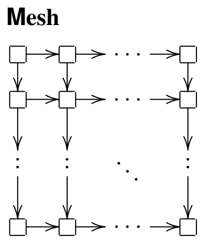
$\Pi i.G \ = \mathbf{R}\ \mathsf{end}\ \lambda i.\lambda \mathbf{x}.G\{i+1/i\}$	$\mathtt{foreach}(i{<}j)\{G\} = \mathbf{R}\ \mathsf{end}\ \lambda i.\lambda \mathbf{x}.G\{\mathbf{x}/\mathsf{end}\}\ j$
$G_1; G_2 = \mathbf{R}\ G_2\ \lambda i.\lambda \mathbf{x}.G_1\{\mathbf{x}/\mathsf{end}\}\ 1$	$\mathsf{if}\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

where we assume that $\mathbf{x}$ is not free in G and G_1, and that G_1 and G terminate with end. The encoding substitutes $\mathbf{x}$ for end. The composition operator executes G_1 and G_2 sequentially; the repetition operator above repeats G j-times[2]; the boolean values are integers 0 (false) and 1 (true). These definitions are similar for other types and processes.

[1] A separation between the recursion and the recursor is essential for decidability results, see § 3.4.

[2] It is useful to write the parameterised communication in the increasing order, see Appendix B.

2.2 Example of a parameterised multiparty protocol



$\Pi n.\Pi m.$
$\texttt{foreach}(i < n)\{$
  $\texttt{foreach}(j < m)\{$
    $\texttt{W}[i+1][j+1] \to \texttt{W}[i][j+1] : \langle \mathsf{nat} \rangle.$
    $\texttt{W}[i+1][j+1] \to \texttt{W}[i+1][j] : \langle \mathsf{nat} \rangle\};$
  $\texttt{W}[i+1][0] \to \texttt{W}[i][0] : \langle \mathsf{nat} \rangle\};$
$\texttt{foreach}(k < m)\{\texttt{W}[0][k+1] \to \texttt{W}[0][k] : \langle \mathsf{nat} \rangle\}$

Fig. 2. Parameterised multiparty protocol on a mesh topology

The session from figure 2 describes a particular protocol over a standard mesh topology [14]. In this two dimensional example, each worker has four neighbours, except for the ones located on the first and last rows and columns. Our session takes two parameters n and m which represent the number of rows and the number of columns. Then we have two iterators that repeat $\texttt{W}[i+1][j+1] \to \texttt{W}[i][j+1] : \langle \mathsf{nat} \rangle$ and $\texttt{W}[i+1][j+1] \to \texttt{W}[i+1][j] : \langle \mathsf{nat} \rangle$ for all i and j. These two messages specify that each worker not situated on the last row or last column sends a message to his neighbours situated below and on its right.[3] The types $\texttt{W}[i+1][0] \to \texttt{W}[i][0] : \langle \mathsf{nat} \rangle$ and $\texttt{foreach}(k < m)\{\texttt{W}[0][k+1] \to \texttt{W}[0][k] : \langle \mathsf{nat} \rangle\}$ deal with, respectively, the last column and the last row. The flow of messages comes from $\texttt{W}[n][m]$ and converges towards $\texttt{W}[0][0]$.

2.3 Process syntax and semantics

Syntax The syntax of expressions and processes is given in figure 3, extended from [5], 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. In processes, sessions are asynchronously initiated by $\bar{u}[\mathrm{p}_0, .., \mathrm{p}_\mathrm{n}](y).P$. It spawns, for each of the $\{\mathrm{p}_0, .., \mathrm{p}_\mathrm{n}\}$[4], a request that is accepted by the participant through $u[\mathrm{p}](y).P$. Messages are sent by $c!\langle \mathrm{p}, e\rangle; P$ to the participant p and received by $c?\langle \mathsf{q}, x\rangle; P$ from the participant q. Selection $c \oplus \langle \mathrm{p}, l\rangle; P$, and branching $c\&\langle \mathsf{q}, \{l_k : P_k\}_{k\in K}\rangle$, allow a participant to choose a branch from those supported by another. Standard language constructs include recursive processes $\mu X.P$, restriction $(\nu s)P$ 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 as in e.g. $(\nu a : \langle G\rangle)Q$ or $s?(x : \langle G\rangle)Q$ or $\mathbf{R}\ Q\ \lambda i : I.\lambda X.Q'$. We omit the annotations unless needed. We often omit $\mathbf{0}$ and the participant p from the session primitives. Requests, session hiding and channel queues appear only at runtime, as explained below.

Semantics The semantics is defined by the reduction relation $\longrightarrow$ presented in figure 4. The standard definition of evaluation contexts (that allow $\texttt{W}[3+1]$ to be reduced to $\texttt{W}[4]$) is

[3] Processes can execute asynchronously and in parallel as long as the communication actions specified in their types are not causally dependant [12]. Hence these messages are multicasted.
[4] Since the set of principals is parameterised, we allow some syntactic sugar to express ranges of participants.

$$
\begin{array}{lll|lll}
c ::= y \mid s[\mathtt{p}] & \text{Channels} & & \hat{\mathtt{p}}, \hat{\mathtt{q}} ::= \hat{\mathtt{p}}[\mathrm{n}] \mid \mathcal{P} & & \text{Principal values} \\
u ::= x \mid a & \text{Identifiers} & & m ::= (\hat{\mathtt{q}},\hat{\mathtt{p}},v) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},s[\hat{\mathtt{p}}']) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},l) & & \text{Messages in transit} \\
v ::= a \mid \mathrm{n} & \text{Values} & & h ::= \epsilon \mid m \cdot h & & \text{Queue types} \\
\multicolumn{6}{l}{e ::= \mathtt{i} \mid v \mid x \mid s[\mathtt{p}] \mid e \;\mathrm{op}\; e' \quad \text{Expressions}}
\end{array}
$$

$$
\begin{array}{lll|ll}
P ::= & & \text{Processes} & \mu X.P & \text{Recursion} \\
 & \mid \bar{u}[\mathtt{p}_0,..,\mathtt{p}_\mathrm{n}](y).P & \text{Init} & \mathbf{0} & \text{Inaction} \\
 & \mid u[\mathtt{p}](y).P & \text{Accept} & P \mid Q & \text{Parallel} \\
 & \mid \bar{a}[\mathtt{p}] : s & \text{Request} & \mathbf{R}\; P\; \lambda i.\lambda X.Q & \text{Primitive recursion} \\
 & \mid c!\langle \mathtt{p}, e\rangle; P & \text{Value sending} & X & \text{Process variable} \\
 & \mid c?\langle \mathtt{p}, x\rangle; P & \text{Value reception} & (P\ \mathtt{i}) & \text{Application} \\
 & \mid c \oplus \langle \mathtt{p}, l\rangle; P & \text{Selection} & (\nu s)P & \text{Session restriction} \\
 & \mid c\&\langle \mathtt{p}, \{l_k : P_k\}_{k\in K}\rangle & \text{Branching} & s{:}h & \text{Queues}
\end{array}
$$

Fig. 3. Syntax for user-defined and run-time processes

$$
\begin{array}{rr}
\mathbf{R}\; P\; \lambda i.\lambda X.Q\; 0 \longrightarrow P & \text{[ZeroR]} \\
\mathbf{R}\; P\; \lambda i.\lambda X.Q\; \mathrm{n}+1 \longrightarrow Q\{\mathrm{n}/i\}\{\mathbf{R}\; P\; \lambda i.\lambda X.Q\; \mathrm{n}/X\} & \text{[SuccR]} \\
\bar{a}[\hat{\mathtt{p}}_0,..,\hat{\mathtt{p}}_\mathrm{n}](y).P \longrightarrow (\nu s)(P\{s[\hat{\mathtt{p}}_0]/y\} \mid s : \emptyset \mid \bar{a}[\hat{\mathtt{p}}_1] : s \mid ... \mid \bar{a}[\hat{\mathtt{p}}_\mathrm{n}] : s) & \text{[Init]} \\
\bar{a}[\hat{\mathtt{p}}_k] : s \mid a[\hat{\mathtt{p}}_k](y_k).P_k \longrightarrow P_k\{s[\hat{\mathtt{p}}_k]/y_k\} & \text{[Join]} \\
s[\hat{\mathtt{p}}]!\langle \hat{\mathtt{q}}, v\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{\mathtt{p}}, \hat{\mathtt{q}}, v) & \text{[Send]} \\
s[\hat{\mathtt{p}}] \oplus \langle \hat{\mathtt{q}}, l\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\hat{\mathtt{p}}, \hat{\mathtt{q}}, l) & \text{[Label]} \\
s[\hat{\mathtt{p}}]?(\hat{\mathtt{q}}, x); P \mid s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, v) \cdot h \longrightarrow P\{v/x\} \mid s : h & \text{[Recv]} \\
s[\hat{\mathtt{p}}]\&(\hat{\mathtt{q}}, \{l_k : P_k\}_{k\in K}) \mid s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, l_{k_0}) \cdot h \longrightarrow P_{k_0} \mid s : h \quad (k_0 \in K) & \text{[Branch]}
\end{array}
$$

Fig. 4. Reduction rules

omitted. The metavariables $\hat{\mathtt{p}}, \hat{\mathtt{q}}$, .. range over principal values (where all indices have been evaluated). [ZeroR] and [SuccR] are standard and the same as for global types. The rule [Init] describes the initialisation of a session by its first participant $\bar{a}[\mathtt{p}_0,..,\mathtt{p}_\mathrm{n}](y_0).P_0$. Asynchronous requests $\bar{a}[\hat{\mathtt{p}}_k] : s$ are spawned to allow delayed acceptance by the other participants (rule [Join]). After the connection, the participants share the private session name s, and the queue associated to s, which is initialised as empty. The variables $y_\mathtt{p}$ in each participant p are then replaced with the corresponding session channel, $s[\mathtt{p}]$.

The rest of the session reductions are standard [5, 12]. The output rules [Send] and [Label] push values, channels and labels into the queue of the session s. The rules [Recv] and [Branch] perform the complementary operations. Note that these operations check that the sender and receiver match. Processes are considered modulo structural equivalence, denoted by $\equiv$ (in particular, we note $\mu X.P \equiv P\{\mu X.P/X\}$).

2.4 Processes for parameterised multiparty protocols

We give here the processes corresponding to the interactions described in § 1 and § 2.2, then introduce a parallel implementation of the Fast Fourier Transform algorithm.

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

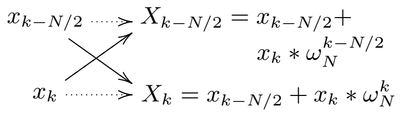
$$
\begin{array}{lll}
\mathit{\Pi} n.(\mathsf{if}\ n = 0 & \mathsf{then} & \mathsf{end} \\
 & \mathsf{else} & (\mathbf{R}\ (\bar{a}[\mathtt{W}[n],..,\mathtt{W}[0]](y).y!\langle \mathtt{W}[n-1], v\rangle; \mathsf{end} \\
 & & \quad \mid a[\mathtt{W}[0]](y).y?(\mathtt{W}[1], z); \mathsf{end}) \\
 & & \quad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

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.

Mesh from figure 2 The mesh example is more complex: when n and m are bigger than 2, there are 9 distinct roles that each have a different pattern of communication. We only list processes for (1) the centre workers $\mathtt{W}[i][j]$ $(0 < i < n,\ 0 < j < m)$ who are connected in all four directions, (2) the initiator $\mathtt{W}[n][m]$ from the top-left corner and (3) the workers $\mathtt{W}[0][j]$ $(1 < j < m)$ from the middle of the bottom row. Below, $f(i, j)$ represents the expression computed at the (i, j)-th element.

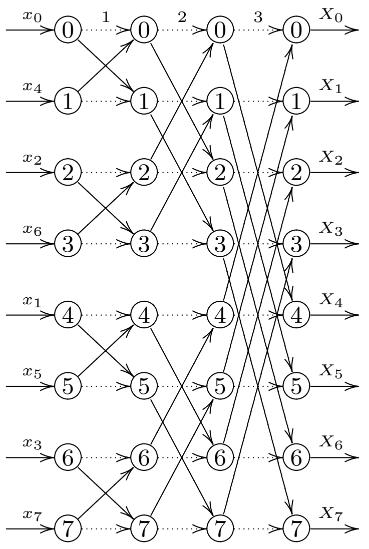
$$\begin{aligned}
P_{\text{centre}}(i,j) &= a[\mathtt{W}[i][j]](y).y?(\mathtt{W}[i+1][j], z_1); y?(\mathtt{W}[i][j+1], z_2);\\
&\qquad y!\langle \mathtt{W}[i-1][j], f(i-1,j)\rangle; y!\langle \mathtt{W}[i][j-1], f(i,j-1)\rangle; \mathbf{0}\\
P_{\text{start}}(n,m) &= \bar{a}[\mathtt{W}[0][0]..\mathtt{W}[n][m]](y).y!\langle \mathtt{W}[n-1][m], f(n-1,m)\rangle;\\
&\qquad y!\langle \mathtt{W}[n][m-1], f(n,m-1)\rangle; \mathbf{0}\\
P_{\text{bot\_middle}}(n,m) &= a[\mathtt{W}[0][j]](y).y?(\mathtt{W}[1][j], x); y?(\mathtt{W}[0][j+1], z); y!\langle \mathtt{W}[0][j-1], f(0,j)\rangle; \mathbf{0}
\end{aligned}$$

(a) Butterfly pattern



$x_{k-N/2}$
$X_{k-N/2} = x_{k-N/2} + x_k * \omega_N^{k-N/2}$
x_k
$X_k = x_{k-N/2} + x_k * \omega_N^k$

(b) FFT diagram



(c) Global type $G =$

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

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

```
  $y!\langle \mathtt{p}, x_{\bar{\mathtt{p}}} \rangle$;
  foreach($l < n$){
    if $\mathsf{bit}_{n-l}(\mathtt{p}) = 0$
     then $y?\langle \mathtt{p}, x \rangle; y!\langle \mathtt{p} + 2^{n-l-1}, x \rangle;$
          $y?\langle \mathtt{p} + 2^{n-l-1}, z \rangle; y!\langle \mathtt{p}, x + z\,\omega_N^{g(l,\mathtt{p})} \rangle;$
     else $y?\langle \mathtt{p}, x \rangle; y!\langle \mathtt{p} - 2^{n-l-1}, x \rangle;$
          $y?\langle \mathtt{p} - 2^{n-l-1}, z \rangle; y!\langle \mathtt{p}, z + x\,\omega_N^{g(l,\mathtt{p})} \rangle;$ };
  $y?\langle \mathtt{p}, x \rangle; r_{\mathtt{p}}!\langle 0, x \rangle;$
where $g(l, \mathtt{p}) = \mathtt{p} \mod 2^l$
```

Fig. 5. Fast Fourier Transform on a butterfly network topology

FFT - Figure 5 We describe a parallel implementation of the Fast Fourier Transform algorithm (more precisely the radix-2 variant of the Cooley-Tukey algorithm [10]).

Figure 5(a) illustrates the recursive principle of the algorithm, called *butterfly*, where two different outputs can be computed in constant time from the results of the same two recursive calls. The complete algorithm is illustrated by the diagram from figure 5(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 $\mathtt{p}$. Figure 5(c) gives the global session type describing the interactions between 2^n machines. The first iterator is the initialisation step. Then we have an iteration over variable l for the n successive steps of the algorithm. Figure 5(d) defines the processes that each of the machines runs. Each process returns the final answer at $r_{\mathtt{p}}$.

3 Typing parameterised multiparty interactions

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
$\mid \oplus\langle \mathtt{p}, \{l_k : T_i\}_{k\in K}\rangle$	Selection	$\mid T\ \mathtt{i}$	Application
$\mid \&\langle \mathtt{p}, \{l_k : T_i\}_{k\in K}\rangle$	Branching	$\mid \mathsf{end}$	End

Fig. 6. End-point types

The syntax of end-point types is given in figure 6. 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 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 [5, 12]. 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.

$$\begin{array}{rcl|rcl}
\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G \upharpoonright \mathtt{q} & = & \text{if } \mathtt{q}\text{=}\mathtt{p}\text{=}\mathtt{p}' \text{ then } !\langle \mathtt{p}, U\rangle; ?\langle \mathtt{p}, U\rangle; G \upharpoonright \mathtt{q} & (\mu\mathbf{t}.G)\upharpoonright \mathtt{p} & = & \mu\mathbf{t}.G \upharpoonright \mathtt{p} \\
 & & \text{else if } \mathtt{q}\text{=}\mathtt{p} \text{ then } !\langle \mathtt{p}', U\rangle; G \upharpoonright \mathtt{q} & \mathbf{x}\upharpoonright \mathtt{p} & = & \mathbf{x} \\
 & & \text{else if } \mathtt{q}\text{=}\mathtt{p}' \text{ then } ?\langle \mathtt{p}, U\rangle; G \upharpoonright \mathtt{q} & (G\ \mathtt{i}) \upharpoonright \mathtt{p} & = & (G\upharpoonright \mathtt{p})\ \mathtt{i} \\
 & & \text{else } G\upharpoonright \mathtt{q} & \mathsf{end} \upharpoonright \mathtt{p} & = & \mathsf{end} \\
\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k\in K} \upharpoonright \mathtt{q} & = & \text{if } \mathtt{q}\text{=}\mathtt{p} \text{ then } \oplus\langle \mathtt{p}', \{l_k : G_k \upharpoonright \mathtt{q}\}_{k\in K}\rangle & & & \\
 & & \text{else if } \mathtt{q}\text{=}\mathtt{p}' \text{ then } \&\langle \mathtt{p}, \{l_k : G_k \upharpoonright \mathtt{q}\}_{k\in K}\rangle & & & \\
 & & \text{else } \sqcup_{k\in K} G_k \upharpoonright \mathtt{q} & & & \\
\mathbf{R}\, G\, \lambda i\!:\!I.\lambda\mathbf{x}.G' \upharpoonright \mathtt{q} & = & \mathbf{R}\, G \upharpoonright \mathtt{q}\, \lambda i\!:\!I.\lambda\mathbf{x}.G' \upharpoonright \mathtt{q} & & &
\end{array}$$

Fig. 7. Projection of global types to end-point types

The general endpoint generator is defined in figure 7 using the derived construct $\text{if \_ then \_ else \_}$. 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 fourth case corresponds to the possibility for the sender and receiver to be identical. Projecting the branching global type is similarly defined, but for the operator $\sqcup$ explained below. For the other cases (as well as for our derived operators), the projection is homomorphic.

Mergeability and injection of branching types We first recall the example from [12], 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, regardless of the choice made by $\mathtt{W}[0]$, both branches fail to behave in the same way, as $\mathtt{W}[2]$ is not aware of the chosen branch and cannot know the type of the expected value. The projection would only be defined if we changed the above nat to bool. This illustrates the fact that the projection of all branches is required to be identical.

In our framework, this restriction is too strong since each branch may contain different parametric interaction patterns. To solve this problem, we propose two methods called *mergeability* and *injection* of branching types. Formally, the mergeability relation $\bowtie$ is the smallest congruence relation over end-point types such that:[5] if $\forall i \in (K \cap J).T_k \bowtie T'_j$ and $\forall i \in (K \setminus J) \cup (J \setminus K).l_k \neq l_j$, then $\&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \bowtie \&\langle \mathtt{p}, \{l_j : T'_j\}_{j\in J}\rangle$. When $T_1 \bowtie T_2$ is defined, we define the injection $\sqcup$ as a partial commutative operator over two types such that $T \sqcup T = T$ for all types and that:

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

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:

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

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 [5, 12]. This projection is sound up to branching subtyping (cf. Lemma 3.4).

3.2 Type system

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

Judgements and environments One of the main differences with previous session type systems is that session environments 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 environments* 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 $u \notin dom(\Gamma)$. We use the same convention for other variables.

[5] The idea of meargeablity is introduced informally in the tutorial paper [8].

Following [20], we assume given in the typing rules two semantically defined judgements: $\mathit{\Gamma} \models \mathtt{P}$ (predicate $\mathtt{P}$ is a consequence of $\mathit{\Gamma}$) and $\mathit{\Gamma} \models \mathtt{i} : I$ ($\mathtt{i} : I$ follows from the assumptions of $\mathit{\Gamma}$). We also inductively define well-formed types using a kind systems (Appendix A). The judgement $\mathit{\Gamma} \vdash U \blacktriangleright \kappa$ 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 $\mathit{\Pi} i : I.\kappa$. Well-formedness of a term $\mathtt{i}$ and $\mathtt{P}$ in $\mathit{\Gamma}$ and environments is defined in the standard way [3].

3.3 Typing processes

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

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

$$\frac{\mathit{\Gamma} \vdash \mathrm{whnf}(G_1) \equiv_{\mathrm{wf}} \mathrm{whnf}(G_2)}{\mathit{\Gamma} \vdash G_1 \equiv G_2}\,[\text{WF}] \qquad \frac{\mathit{\Gamma} \vdash P \rhd \tau \quad \mathit{\Gamma} \vdash \tau \equiv \tau'}{\mathit{\Gamma} \vdash P \rhd \tau'}\,[\text{TEq}] \qquad \frac{\mathit{\Gamma} \vdash P \rhd \tau \quad \mathit{\Gamma} \vdash \tau \leq \tau'}{\mathit{\Gamma} \vdash P \rhd \tau'}\,[\text{TSub}]$$

$$\frac{\mathit{\Gamma}, X : \tau \vdash P \rhd \tau}{\mathit{\Gamma} \vdash \mu X.P \rhd \tau}\,[\text{TRec}] \qquad \frac{\mathit{\Gamma}, X : \tau \vdash \mathsf{Env}}{\mathit{\Gamma}, X : \tau \vdash X \rhd \tau}\,[\text{TVar}] \qquad \frac{\mathit{\Gamma} \vdash P \rhd \mathit{\Pi} i{:}I.\tau \quad \mathit{\Gamma} \models \mathtt{i} \in I}{\mathit{\Gamma} \vdash P\ \mathtt{i} \rhd \tau\{\mathtt{i}/i\}}\,[\text{TApp}]$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash u : \langle G \rangle \quad \mathit{\Gamma} \vdash P \rhd \mathit{\Delta}, y : G \upharpoonright \mathrm{p}_0 \\ \mathit{\Gamma} \vdash \mathrm{p}_i \rhd \mathsf{nat} \quad \mathit{\Gamma} \models \mathsf{pid}(G) = \{\mathrm{p}_0..\mathrm{p}_\mathrm{n}\}\end{array}}{\mathit{\Gamma} \vdash \bar{u}[\mathrm{p}_0, .., \mathrm{p}_\mathrm{n}](y).P \rhd \mathit{\Delta}}\,[\text{TInit}] \qquad \frac{\begin{array}{c}\mathit{\Gamma} \vdash u : \langle G \rangle \quad \mathit{\Gamma} \vdash P \rhd \mathit{\Delta}, y : G \upharpoonright \mathtt{p} \\ \mathit{\Gamma} \vdash \mathtt{p} \rhd \mathsf{nat} \quad \mathit{\Gamma} \models \mathtt{p} \in \mathsf{pid}(G)\end{array}}{\mathit{\Gamma} \vdash u[\mathrm{p}](y).P \rhd \mathit{\Delta}}\,[\text{TAcc}]$$

$$\frac{\mathit{\Gamma} \vdash a : \langle G \rangle \quad \mathit{\Gamma} \vdash \mathtt{p} \rhd \mathsf{nat} \quad \mathit{\Gamma} \models \mathtt{p} \in \mathsf{pid}(G)}{\mathit{\Gamma} \vdash \bar{a}[\mathrm{p}] : s \rhd s[\mathrm{p}] : G \upharpoonright \mathtt{p}}\,[\text{TReq}] \qquad \frac{\mathit{\Gamma} \vdash e \rhd S \quad \mathit{\Gamma} \vdash P \rhd \mathit{\Delta}, c : T}{\mathit{\Gamma} \vdash c!\langle \mathrm{p}, e \rangle; P \rhd \mathit{\Delta}, c :!\langle \mathrm{p}, S \rangle; T}\,[\text{TOut}]$$

Fig. 8. Process typing

We explain here (Figure 8) a selection of the process typing rules of our system. Rules $\lfloor$TNat$\rfloor$ and $\lfloor$TVar$\rfloor$ are standard ($\mathit{\Gamma} \vdash \mathsf{Env}$ means that $\mathit{\Gamma}$ is well-formed). For participants, we check their typing by $\lfloor$TId$\rfloor$ and $\lfloor$TP$\rfloor$ in a similar way as [20] where $\mathit{\Gamma} \vdash \kappa$ means kinding κ is well-formed. In $\lfloor$TPRec$\rfloor$, we use the abbreviation $[0..\mathtt{j}] = \{i : \mathsf{nat} \mid i \leq \mathtt{j}\}$. Then we define I^{-} by $[0..0]^{-} = \emptyset$ and $[0..\mathtt{i}]^{-} = [0..\mathtt{i} - 1]$. This rule needs to deal 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 $\mathit{\Gamma}, i{:}I^{-}, X{:}\tau\{i/j\}$ where $\tau\{i/j\}$ of X means that X satisfies the predecessor's type (induction hypothesis). The rule $\lfloor$TApp$\rfloor$ is the elimination rule of dependent types. Since our types include dependent types and recursors, we need a notion of type equivalence $\lfloor$TEq$\rfloor$ to type processes up-to type reductions. The rule $\lfloor$WF$\rfloor$ 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 (we extend the standard method from [3, §2] with the recursor).

$\lfloor$TInit$\rfloor$ types a session initialisation on shared channel u, binding channel y and requiring participants $\{\mathrm{p}_0, .., \mathrm{p}_n\}$. 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 $\{p_0,..,p_n\}$. $\lfloor$TACC$\rfloor$ allows to type the p-th participant to the session initiated on u. The typing rule checks that the type of y is the p-th projection of the global type G of u and that G is fully instantiated. The kind rule ensures that G is fully instantiated (i.e. G''s kind is Type). $\lfloor$TREQ$\rfloor$ types the process that waits for an accept from a participant: its type corresponds to the end-point projection of G.

Recursion $\lfloor$TREC$\rfloor$, variable ($\lfloor$TVAR$\rfloor$), output ($\lfloor$TOUT$\rfloor$), input, delegation, inaction, branching/selection and the expression typing rules as well as the typing rules for queues are similar to those in [5, 12].

3.4 Properties of typing

Ensuring termination of type-checking with dependent types is not an easy task since type equivalences are often defined from term equivalences. We rely here on the strong normalisation of System $\mathcal{T}$ [11] for the termination proof.

Proposition 3.1 (Termination and Confluence) *The head 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.*

The following lemma is proved by defining the weight of the equality and showing the weight of any premise of a rule is always less than the weight of the conclusion (the weight for a recursor needs to be extended to allow the inductive equality rule).

Proposition 3.2 (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.*

Proposition 3.3 (Termination for 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. (Outline) By the standard argument from indexed dependent types [3, 20], for the dependent λ-applications, we do not require equality of terms (i.e. we only need the equality of the indices by the semantic consequence judgements). Hence to eliminate the type equality rule $\lfloor$TEQ$\rfloor$, we include the type equality check into $\lfloor$TINIT,TREQ,TACC$\rfloor$ (between the global type and its projected session type), and the input rule (between the session type and the type annotating x). Similarly for recursive agents. Since $\alpha \equiv \beta$ (for any type α and β) terminates, these checks always terminate. □

Notice that the projection of G on p is always decidable if the equality on principals is decidable. To ensure the termination of $\Gamma \models \mathtt{P}$, several solutions include the restriction of predicates to linear equalities over the naturals without multiplications (or to other decidable arithmetic subsets) or the restriction of indices to finite domains, cf. [20].

3.5 Subject reduction

The following lemma states that mergeability is sound with respect to the branching subtyping $\leq$ (see figure 25 in the Appendix).

Lemma 3.4 (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 $\leq$.*

By this lemma, 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 in the runtime typing as done in [12].

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 $\equiv$.

- $\{s[\hat{\mathtt{p}}] :!\langle \hat{\mathtt{q}}, U\rangle; T, s[\hat{\mathtt{q}}] :?\langle \hat{\mathtt{p}}, U\rangle; T'\} \Rightarrow \{s[\hat{\mathtt{p}}] : T, s[\hat{\mathtt{q}}] : T'\}$
- $\{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, \{l_k : T_k\}_{k\in K}\rangle\} \Rightarrow \{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, l_j\rangle; T_j\}$
- $\{s[\hat{\mathtt{p}}] : \oplus\langle \hat{\mathtt{q}}, l_j\rangle; T, s[\hat{\mathtt{q}}] : \&(\mathtt{p}, \{l_k : T_k\}_{k\in K})\} \Rightarrow \{s[\hat{\mathtt{p}}] : T, s[\hat{\mathtt{q}}] : T_j\}$
- $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta''$ 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 treats the case of the choice of label l_j while the third rule propagate these choices to the receiver (participant $\hat{\mathtt{q}}$). Using the above notion we can state type preservation under reductions as follows:

Theorem 3.5 (Subject Congruence and Reduction)
- *If $\Gamma \vdash P \triangleright \Delta$ and $P \equiv P'$, then $\Gamma \vdash P' \triangleright \Delta$.*
- *If $\Gamma \vdash P \triangleright \tau$ and $P \longrightarrow^* P'$, then $\Gamma \vdash P' \triangleright \tau'$ for some τ' such that $\tau \Rightarrow^* \tau'$.*

Note that communication safety [12, Theorem 5.5] and session fidelity [12, Corollary 5.6] are corollaries of the above theorem. A notable fact is, in the presence of the asynchronous join primitive, we can still obtain *progress* in a single multiparty session as in [12, Theorem 5.12], i.e. if a program P starts from one session, the reductions at session channels do not get a stuck. Formally we write $\Gamma \vdash^\star P \triangleright \Delta$ if P is typable and with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to at most a singleton. Another element which can hinder progress is when interactions at shared channels cannot proceed. 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. The proof is similar to [12, Theorem 5.12].[6]

Theorem 3.6 (Progress) *If P is well-linked and does not contain 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.*

3.6 Typing examples

Repetition example - § 1 (1) This example illustrates the repetition of a message pattern. Let $G(n) = \mathtt{foreach}(i < n)\{\mathtt{Alice} \to \mathtt{Bob}\colon\langle\mathsf{nat}\rangle.\mathtt{Bob} \to \mathtt{Carol}\colon\langle\mathsf{nat}\rangle\}$. Following the projection definition from Figure 7, `Alice`'s end-point projection of $G(n)$ is:

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

Let $\mathtt{Alice}(n) = \bar{a}[\mathtt{Alice},\mathtt{Bob},\mathtt{Carol}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle\mathtt{Bob}, e[i]\rangle; X\ n)$ and $\Delta(n) = \{y : (G(n) \upharpoonright \mathtt{Alice})\}$ and $\Gamma = n\colon\mathsf{nat}, a\colon\langle G\rangle$. We can prove that $\Gamma \vdash \mathtt{Alice}(n) \triangleright \emptyset$ from $\lfloor\text{TI}\small{\text{NIT}}\rfloor$ if we have $\Gamma \vdash \mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle\mathtt{Bob}, e[i]\rangle; X\ n \triangleright \Delta(n)$. This, in turn, is given by $\lfloor\text{TPR}\small{\text{EC}}\rfloor$ and $\lfloor\text{TA}\small{\text{PP}}\rfloor$ from $\Gamma, i : I^-, X : \Delta(i) \vdash y!\langle\mathtt{Bob}, e[i]\rangle; X \triangleright \Delta(i{+}1)$ and the trivial $\Gamma \vdash \mathbf{0} \triangleright y : \mathsf{end}$. From $\lfloor\text{TV}\small{\text{AR}}\rfloor$, we have $\Gamma, i : I^-, X : \Delta(i) \vdash X \triangleright \Delta(i)$. We conclude by $\lfloor\text{TO}\small{\text{UT}}\rfloor$ and weak head normal form equivalence $\lfloor\text{WF}\rfloor$ of the types $\Delta(i+1)$ and $y :!\langle\mathtt{Bob}, \mathsf{nat}\rangle; (\mathbf{R}\ \mathsf{end}\ \lambda j.\lambda\mathbf{x}.!\langle\mathtt{Bob}, \mathsf{nat}\rangle; \mathbf{x}\ i)$. $\mathtt{Bob}(n)$ and $\mathtt{Carol}(n)$ can be similarly typed.

[6] A stronger progress property for interleaved multiparty sessions ensured by the interaction typing in [5] can be obtained in this framework, too (since our typing system is an extension from the communication system in [5]).

Sequence example - § 1 (2) The sequence example consists of three roles (when $n \geq 2$): the starter $\mathtt{W}[n]$ sends the first message, the final worker $\mathtt{W}[0]$ receives the final message and the middle workers first receive a message and then send another to the next worker. We write below the generic projection for participant $\mathtt{W}[\mathtt{p}]$ (left) and the end-point type that naturally types the processes (right):

$$\begin{array}{l|l}
\mathbf{R}\ \textsf{end}\ \lambda i.\lambda \mathbf{x}. & \\
\ \textsf{if}\ \ \mathtt{p} = \mathtt{W}[i+1]\ \textsf{then}\ !\langle \mathtt{W}[i], \textsf{nat}\rangle; \mathbf{x} & \textsf{if}\ \ \mathtt{p} = \mathtt{W}[n]\ \textsf{then}\ !\langle \mathtt{W}[n-1], \textsf{nat}\rangle; \textsf{else} \\
\ \textsf{else if}\ \ \mathtt{p} = \mathtt{W}[i]\ \textsf{then}\ ?\langle \mathtt{W}[i+1], \textsf{nat}\rangle; \mathbf{x} & \textsf{if}\ \ \mathtt{p} = \mathtt{W}[0]\ \textsf{then}\ ?\langle \mathtt{W}[1], \textsf{nat}\rangle;\textsf{else} \\
\ \textsf{else if}\ \ \mathbf{x} \qquad\qquad\qquad\qquad\qquad n & \textsf{if}\ \ \mathtt{p} = \mathtt{W}[i]\ \textsf{then}\ ?\langle \mathtt{W}[i+1], \textsf{nat}\rangle;!\langle \mathtt{W}[i-1], \textsf{nat}\rangle;
\end{array}$$

For readability we omitted in the projected type the impossible case $\mathtt{p} = \mathtt{W}[i+1] = \mathtt{W}[i]$. In order to type this example, we need to prove the equivalence of these two types. For any instantiation of $\mathtt{p}$ and n, the standard weak head normal form equivalence rule $\lfloor \mathrm{WF} \rfloor$ is sufficient. Proving the equivalence for all $\mathtt{p}$ and n requires either (a) to bound the domain I in which they live, and check all instantiations within this finite domain; or (b) to prove the equivalence through a meta-logic case analysis. In case (a), type checking terminates, while case (b) allows to easily prove strong properties about a protocol's implementation.

FFT example - Figure 5 We prove type-safety and deadlock-freedom for the FFT processes. Let P_{fft} be the following process:

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

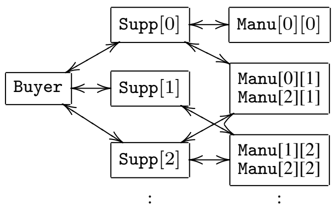
As we reasoned above, each $P(n, \mathtt{p}, x_{\overline{\mathtt{p}}}, y, r_{\mathtt{p}})$ is straightforwardly typable by an end-point type which is equivalent with the one projected from the global type G from figure 5(c). Automatically checking the equivalence for all n is not easy though: we need to rely on the finite domain restriction using $\lfloor \mathrm{WF} \rfloor$. 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 3.7 (Type safety and deadlock-freedom of FFT) *For all* m, $\emptyset \vdash P_{\mathit{fft}}\ \mathrm{m} \triangleright \emptyset$*; and for all Q such that 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. By the progress property from Theorem 3.5, noting $P_{\text{fft}}\ \mathrm{m}$ is automatically typable by a single, multiparty dependent session. □

Web Service example - Figure 9 We program and type a real-world Web service use case: Quote Request (C-U-002) is the most complex scenario described in [2], the public document authored by the W3C Choreography Description Language Working Group [19]. As described in Figure 9, a buyer interacts with multiple suppliers who in turn interact with multiple manufacturers in order to obtain quotes for some goods or services. The Requirements from Section 3.1.2.2 of [2] include the ability to *reference a global description from within a global description* to support *recursive behaviour* as denoted in STEP 4(b, d): it can be achieved by parameterised multiparty session types.

We write the specification of the usecase program modularly, starting from the first steps of the informal description above. Here, `Buyer` stands for the buyer, $\mathtt{Supp}[i]$ for a supplier, and $\mathtt{Manu}[j]$ for a manufacturer. We alias manufacturers by $\mathtt{Manu}[i][j]$ to express the fact that $\mathtt{Manu}[j]$ is connected to $\mathtt{Supp}[i]$ (a single $\mathtt{Manu}[j]$ can have multiple aliases $\mathtt{Manu}[i'][j]$, see figure 9). Then, we can write global types for each of the steps. STEP 1



1. **1.** A buyer requests a quote from a set of suppliers.
   $G_1 = \mathtt{foreach}(i < n)\{\mathtt{Buyer} \rightarrow \mathtt{Supp}[i] : \langle\mathsf{Quote}\rangle\}$
2. **2.** All suppliers receive the request ask their respective manufacturers for a bill of material items. The suppliers interact with their manufacturers to build their quotes for the buyer.
   $$G_2(i) = \mathtt{foreach}(j : J_i)\{\ \mathtt{Supp}[i] \rightarrow \mathtt{Manu}[i][j] : \langle\mathsf{Item}\rangle.\ \mathtt{Manu}[i][j] \rightarrow \mathtt{Supp}[i] : \langle\mathsf{Quote}\rangle\}$$
   The eventual quote is sent back to the buyer.
   $G_2 = \mathtt{foreach}(i : I)\{G_2(i); \mathtt{Supp}[i] \rightarrow \mathtt{Buyer} : \langle\mathsf{Quote}\rangle\}$
3. **3.** EITHER
   (a) The buyer agrees with one or more of the quotes and places the order(s). OR
   (b) The buyer responds to one or more of the quotes by modifying and sending them back to the relevant suppliers.
4. **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.

$$\begin{array}{l}
G_3 = \mathbf{R}\ \mathbf{t}\ \lambda i.\lambda \mathbf{y}.\mathtt{Buyer} \rightarrow \mathtt{Supp}[i] : \{\\
\ \mathsf{ok} : \quad \mathsf{end}\\
\ \mathsf{modify} : \mathtt{Buyer} \rightarrow \mathtt{Supp}[i] : \langle\mathsf{Quote}\rangle.\\
\qquad \mathtt{Supp}[i] \rightarrow \mathtt{Buyer} : \{\mathsf{ok} : \quad \mathsf{end}\\
\qquad\qquad \mathsf{retryStep3} : \mathbf{y}\\
\qquad\qquad \mathsf{reject} : \quad \mathsf{end}\}\}\ i
\end{array}
\quad\Bigg|\quad
\begin{array}{l}
G_3 \upharpoonright \mathtt{Supp}[\mathrm{n}] = \&\langle\mathtt{Buyer}, \{\\
\ \mathsf{ok} : \quad \mathsf{end}\\
\ \mathsf{modify} : ?\langle\mathtt{Buyer}, \mathsf{Quote}\rangle; \oplus\langle\mathtt{Buyer}, \{\\
\qquad\qquad \mathsf{ok} : \quad \mathsf{end}\\
\qquad\qquad \mathsf{retryStep3} : \mathbf{y}\\
\qquad\qquad \mathsf{reject} : \quad \mathsf{end}\}\rangle\}\rangle
\end{array}$$

Fig. 9. The Quote Request use case (C-U-002) [2] with the corresponding global types

is a simple *multicast* (type G_1). For STEP 2, we write first $G_2(i)$, the nested interaction loop between the i-th supplier and its manufacturers (J_i gives all $\mathtt{Manu}[j]$ connected to $\mathtt{Supp}[i]$). Then G_2 can describe the subsequent action within the main loop. For STEP 3, for simplicity we assume the preference is given by the (reverse) ordering of I. The first choice of G_3 corresponds to the two cases of STEP 3. In the innermost branch of G_3, the branches ok, $\mathsf{retryStep3}$ and reject correspond to STEP 4(a), (b) and (c) respectively, while the type variable **t** models STEP 4(d). We can now compose these subprotocols together. The full global type is then $G = \Pi i.\Pi \tilde{J}.(G_1\ ;\ \mu\mathbf{t}.(G_2\ ;\ G_3))$ where we have i suppliers, and $\tilde{J}$ gives the index sets J_i of the $\mathtt{Manu}[j]$s connected with each $\mathtt{Supp}[i]$.

For the end-point projection, we focus on the suppliers' case. The projections of G_1 and G_2 are straightforward. For $G_3 \upharpoonright \mathtt{Supp}[\mathrm{n}]$, 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 projection written in Figure 9. To tell the other suppliers whether the loop is being reiterated or if it is finished, we can simply insert the following closing notification $\mathtt{foreach}(j \in I \setminus i)\{\mathtt{Buyer} \rightarrow \mathtt{Supp}[j] : \{\mathsf{close} :\}\}$ before each end, and a similar retry notification (with label $\mathsf{retryStep3}$) before $\mathbf{x}$. Finally, each end-point type is formed by $(G_1 \upharpoonright \mathtt{Supp}[\mathrm{n}]\ ;\ \mu\mathbf{x}.G_2 \upharpoonright \mathtt{Supp}[\mathrm{n}]\ ;\ \ G_3 \upharpoonright \mathtt{Supp}[\mathrm{n}])$. While the global types look sequential, actual typed processes can asynchronously join a session and be executed in parallel (e.g., at STEP 1-2, no synchronisation is needed between $\mathtt{Supp}[i]$), cf. footnote 3.

4 Extensions and related work

Programming experiments We have explored the impact of the parametrised type structures for communications through implementation of the above use case as well as a few parallel algorithms with parameterised topologies in Java with session types [13], including the Jacobi method (with a sequence and a mesh) and the FFT (a butterfly network on an hypercube). We observe two immediate benefits: (1) a clear coordination of the communication behaviour of each party with the construction of the whole multiparty protocol, thus reducing the programming errors and ensuring deadlock-freedom; (2) a performance benefit against the original binary session version, reducing the overhead of multiple binary session establishments (see [1]).

Dependent types The first use of primitive recursive functionals for dependent types is in Nelson's $\mathcal{T}^{\pi}$ [17] for the λ-calculus, which is a finite representation of $\mathcal{T}^{\infty}$ by Tait and Martin Löf [15, 18]. $\mathcal{T}^{\pi}$ 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 [3, 20] 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. None of these works investigate families of global specifications using dependent types.

Types and contracts for multiparty interactions Recent formalisms for typing multiparty interactions include [7, 9]. These works treat different aspects of dynamic session structures. *Contracts* [9] can type more processes than session types, thanks to the flexibility of process syntax for describing protocols. However, typable processes themselves in [9] 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 (interleavings, choices) of a protocol. The most complex example of [9, § 3] (a group key agreement protocol taken from [4]), which is typed as π-processes with delegations, can be specified and articulated by a single parameterised global session type as follows:

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

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

Conversation Calculus [7] supports the dynamic joining and leaving of participants. Though the formalism in § 2.3 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. Online version of this paper. `http://www.doc.ic.ac.uk/~yoshida/dependent/`.
2. Web Services Choreography Requirements (No. 11). `http://www.w3.org/TR/ws-chor-reqs`.
3. D. Aspinall and M. Hofmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT, 2005.
4. 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.
5. L. Bettini et al. Global progress in dynamically interfered multiparty sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433, 2008.
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. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
8. 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.
9. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, number 5710 in LNCS, pages 211–228, 2009.
10. 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.
11. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1989.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
13. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
14. F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1991.
15. P. Martin-Lőf. Infinite terms and a system of natural deduction. In *Compositio Mathematica*, pages 93–103. Wolters-Noordhoof, 1972.
16. 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.
17. N. Nelson. Primitive recursive functionals with dependent types. In *MFPS*, volume 598 of *LNCS*, pages 125–143, 1991.
18. W. W. Tait. Infinitely long terms of transfinite type. In *Formal Systems and Recursive Functions*, pages 177–185. North Holland, 1965.
19. Web Services Choreography Working Group. Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
20. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

A Syntax, typing rules and semantics

In this Appendix section, we give the elements of our syntax, type system and semantics that were omitted in the main sections.

Evaluation contexts (Figure 10)

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

Fig. 10. Evaluation contexts

Structural equivalence (Figure 11)

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

$$(\nu r)\, \mathbf{0} \equiv \mathbf{0} \quad (\nu s)\, s : \emptyset \equiv \mathbf{0}$$

$$(\nu r)\, P \mid Q \equiv (\nu r)\, (P \mid Q) \qquad \text{if } r \notin \text{fn}(Q)$$

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

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

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

Fig. 11. Structural equivalence

Reduction rules (Figure 12)

$$(\lambda i.P)\ \mathrm{n} \longrightarrow P\{\mathrm{n}/i\} \qquad \text{[Beta]}$$

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

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

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

$$e_0 \longrightarrow e_0' \quad \Rightarrow \quad \mathcal{E}[e_0, \ldots, e_i] \longrightarrow \mathcal{E}[e_0', \ldots, e_i] \qquad \text{[Context]}$$

Fig. 12. Reduction rules (2)

Judgements (Figure 13)

Kinding The definition of kinds is given in figure 14.

The kinding rules are defined in Figures 15, 16, 17, 18, 19, 20 and 21. We only explain the global type kinding rules from Figure 19.

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$\rfloor$ and abstractions in $\lfloor$KPI$\rfloor$. 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 $[0..\mathrm{j}] = \{i{:}\mathsf{nat} \mid i \leq \mathrm{j}\}$:

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

Fig. 13. Judgements ($\alpha, \beta, ...$ range over any types)

$U_p ::= \mathsf{nat} \mid \Pi i\!:\!I.U_p$
$\kappa ::= \Pi j : I.\kappa \mid \mathsf{Type} \mid \mathsf{SType} \mid \mathsf{PAType} \mid \mathsf{LType} \mid \mathsf{PType}$

Fig. 14. Kinds

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

Note that the second argument ($\lambda i\!:\!I^-.\lambda \mathrm{x}.G'$) 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 weather 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 PType).

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{Type}}\lfloor\text{KBASE}\rfloor \qquad \frac{\Gamma \vdash I \quad \Gamma, i\!:\!I \vdash \kappa}{\Gamma \vdash \Pi i\!:\!I.\kappa}\lfloor\text{KSEQ}\rfloor$$

[KBase] works similarly for $\mathsf{PAType}, \mathsf{SType}, \mathsf{LType}, \mathsf{PType}$.

Fig. 15. Kind correctness

Type equivalence The rules are found in figures 22 and 23. We only define the rules for G while the same set of rules can be applied to T and τ. Since the dependent abstraction $\Pi i : I.G$ is used many times in the examples and τ includes $\Pi i : I.\tau'$ in its syntax, we include $\Pi i\!:\!I.G$ in the definitions.

To check $G_1 \equiv G_2$, we use the following algorithm:

1. We first reduce each types to the weak head normal forms $G'_i = \text{whnf}(G_i)$ by $\lfloor$WFBASE$\rfloor$.
2. Then we check $G'_1 \equiv_{\text{wf}} G'_2$.
   (a) If G'_i is not in the form of the application, then we check they are equal or not by induction on the structures of types.
   (b) If $G'_i = G''_i\mathtt{i}_i$, since $G''_i\mathtt{i}_i$ is in the weak head normal form ($\lfloor$WFAPP$\rfloor$), there are two possibilities:
      i. If G''_i is an either recursor or abstraction with $\mathtt{i}_i$ is a variable: then we check G''_1 and G''_2 are equal or not by induction on the structures of types ($\lfloor$WFPROD,WFREC$\rfloor$); if it fails and in the case of the index is finite ($I = [0..\mathrm{n}]$), then we check they are mathematically equal or not in ($\lfloor$WFRECF$\rfloor$, $\lfloor$WFRL$\rfloor$, $\lfloor$WFLR$\rfloor$ and $\lfloor$WFLL$\rfloor$).
      ii. If G''_i is an application, we check they are equal or not by induction, by applying $\lfloor$WFAPP$\rfloor$ again.

Typing rules (Figure 24). Δ end only means $\forall c \in dom(\Delta).\Delta(c) = \mathsf{end}$.

$$\frac{\Gamma \vdash G \blacktriangleright \mathsf{Type} \quad \mathsf{ftv}(G) = \emptyset}{\Gamma \vdash \langle G \rangle \blacktriangleright \mathsf{SType}} \lfloor\text{KMAR}\rfloor$$

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat} \blacktriangleright \mathsf{SType}} \lfloor\text{KNAT}\rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{bool} \blacktriangleright \mathsf{SType}} \lfloor\text{KBOOL}\rfloor$$

Fig. 16. Kinding rules for value types

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat} \blacktriangleright \mathsf{PAType}} \lfloor\text{KPABASE}\rfloor \qquad \frac{\Gamma, i\!:\!I \vdash U_p \blacktriangleright \kappa}{\Gamma \vdash U_p \blacktriangleright \Pi i\!:\!I.\kappa} \lfloor\text{KPABASE}\rfloor$$

Fig. 17. Kinding rules for principals

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{nat}} \lfloor\text{ENVN}\rfloor \qquad \frac{\Gamma, i\!:\!I \models \mathtt{P} \wedge 0 \leq i}{\Gamma \vdash \{i\!:\!I \mid \mathtt{P} \wedge 0 \leq i\}} \lfloor\text{ENVI}\rfloor$$

Fig. 18. Kinding for Index Sets

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

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

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

$$\frac{\Gamma \vdash G \blacktriangleright \mathsf{Type}}{\Gamma \vdash \mu \mathbf{x}.G \blacktriangleright \mathsf{Type}} \lfloor\text{KREC}\rfloor \qquad \frac{\Gamma \vdash \kappa}{\Gamma \vdash \mathbf{x} \blacktriangleright \kappa} \lfloor\text{KVAR}\rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \blacktriangleright \mathsf{Type}} \lfloor\text{KEND}\rfloor$$

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

Fig. 19. Kinding rules for global types

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \mathsf{nat} \quad \Gamma \vdash T \blacktriangleright \mathsf{LType} \quad \Gamma \vdash U \blacktriangleright \mathsf{SType}\text{ or }\mathsf{LType}}{\Gamma \vdash !\langle \mathtt{p}, U\rangle; T \blacktriangleright \mathsf{LType}} \lfloor\text{KLOUT}\rfloor$$

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \mathsf{nat} \quad \Gamma \vdash T \blacktriangleright \mathsf{LType} \quad \Gamma \vdash U \blacktriangleright \mathsf{SType}\text{ or }\mathsf{LType}}{\Gamma \vdash ?\langle \mathtt{p}, U\rangle; T \blacktriangleright \mathsf{LType}} \lfloor\text{KLIN}\rfloor$$

$$\frac{\Gamma \vdash T \blacktriangleright \Pi i{:}I.\kappa \quad \Gamma \models \mathtt{i} : I}{\Gamma \vdash T\ \mathtt{i} \blacktriangleright \kappa\{\mathtt{i}/i\}} \lfloor\text{KLPROJ}\rfloor$$

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \mathsf{nat} \quad \forall k \in K, \Gamma \vdash T_k \blacktriangleright \mathsf{LType}}{\Gamma \vdash \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \blacktriangleright \mathsf{LType}} \lfloor\text{KLSEL}\rfloor$$

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \mathsf{nat} \quad \forall k \in K, \Gamma \vdash T_k \blacktriangleright \mathsf{LType}}{\Gamma \vdash \&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle \blacktriangleright \mathsf{LType}} \lfloor\text{KLBRANCH}\rfloor$$

$$\frac{\Gamma \vdash T \blacktriangleright \kappa\{0/j\} \quad \Gamma, i : I^- \vdash T' \blacktriangleright \kappa\{i+1/j\}}{\Gamma \vdash \mathbf{R}\, T\, \lambda i{:}I^-.\lambda \mathbf{x}.T' \blacktriangleright \Pi j{:}I.\kappa} \lfloor\text{KLRECSEQ}\rfloor$$

$$\frac{\Gamma \vdash \kappa}{\Gamma \vdash \mathbf{x} \blacktriangleright \kappa} \lfloor\text{KVAR}\rfloor \qquad \frac{\Gamma \vdash T \blacktriangleright \mathsf{LType}}{\Gamma \vdash \mu\mathbf{t}.T \blacktriangleright \mathsf{LType}} \lfloor\text{KLREC}\rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \blacktriangleright \mathsf{LType}} \lfloor\text{KLTVAR}\rfloor$$

Fig. 20. Kinding rules for local types

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \emptyset \blacktriangleright \mathsf{PType}} \lfloor\text{KPNULL}\rfloor \qquad \frac{\Gamma \vdash \Delta \blacktriangleright \mathsf{PType} \quad \Gamma \vdash T \blacktriangleright \mathsf{LType}}{\Gamma \vdash \Delta, c : T \blacktriangleright \mathsf{PType}} \lfloor\text{KPCHAN}\rfloor$$

$$\frac{\Gamma, i : I \vdash \tau \blacktriangleright \kappa}{\Gamma \vdash \Pi i{:}I.\tau \blacktriangleright \Pi i{:}I.\kappa} \lfloor\text{KPPROD}\rfloor$$

Fig. 21. Kinding rules for process types

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

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

$$\frac{\Gamma \vdash G_1 \equiv G_2}{\Gamma \vdash \mu \mathbf{x}.G_1 \equiv_{\mathrm{wf}} \mu \mathbf{x}.G_2} \lfloor \textsc{WfPRec} \rfloor$$

$$\frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathbf{x} \equiv_{\mathrm{wf}} \mathbf{x}} \lfloor \textsc{WfRVar} \rfloor \qquad \frac{\Gamma \vdash \mathsf{Env}}{\Gamma \vdash \mathsf{end} \equiv_{\mathrm{wf}} \mathsf{end}} \lfloor \textsc{WfEnd} \rfloor$$

$$\frac{\Gamma, i \colon I \vdash G_1 \equiv G_2}{\Gamma \vdash \Pi i \colon I.G_1 \equiv_{\mathrm{wf}} \Pi i \colon I.G_2} \lfloor \textsc{WfProd} \rfloor$$

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

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

Fig. 22. Global type weak head-normal form equivalence: Context rules

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

$$\frac{\Gamma \vdash \mathbf{R}\, G_1\ \lambda i \colon I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \mathbf{R}\, G_2\ \lambda i \colon I.\lambda \mathbf{x}.G_2'\ \mathrm{n} \quad \Gamma \models I = [0..\mathrm{m}] \quad 0 \le \mathrm{n} \le \mathrm{m}}{\Gamma \vdash \mathbf{R}\, G_1\ \lambda i \colon I.\lambda \mathbf{x}.G_1' \equiv_{\mathrm{wf}} \mathbf{R}\, G_2\ \lambda i \colon I.\lambda \mathbf{x}.G_2'} \lfloor \textsc{WfRecF} \rfloor$$

$$\frac{\Gamma \vdash \mathbf{R}\, G_1\ \lambda i \colon I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \Pi i \colon I.G_2\ \mathrm{n} \quad \Gamma \models I = [0..\mathrm{m}] \quad 0 \le \mathrm{n} \le \mathrm{m}}{\Gamma \vdash \mathbf{R}\, G_0\ \lambda i \colon I.\lambda \mathbf{x}.G_1 \equiv_{\mathrm{wf}} \Pi i \colon I.G_2} \lfloor \textsc{WfRL} \rfloor$$

$$\frac{\Gamma \vdash \mathbf{R}\, G_1\ \lambda i \colon I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \Pi i \colon I.G_2\ \mathrm{n} \quad \Gamma \models I = [0..\mathrm{m}] \quad 0 \le \mathrm{n} \le \mathrm{m}}{\Gamma \vdash \Pi i \colon I.G_2 \equiv_{\mathrm{wf}} \mathbf{R}\, G_0\ \lambda i \colon I.\lambda \mathbf{x}.G_1} \lfloor \textsc{WfLR} \rfloor$$

$$\frac{\Gamma \vdash \Pi i \colon I.G_1\ \mathrm{n} \equiv \Pi i \colon I.G_2\ \mathrm{n} \quad \Gamma \models I = [0..\mathrm{m}] \quad 0 \le \mathrm{n} \le \mathrm{m}}{\Gamma \vdash \Pi i \colon I.G_1 \equiv_{\mathrm{wf}} \Pi i \colon I.G_2} \lfloor \textsc{WfLL} \rfloor$$

Fig. 23. Global type weak head-normal form equivalence: Mathematical Induction rules

$$\frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?\langle \mathtt{p}, x\rangle; P \triangleright \Delta, c : ?\langle \mathtt{p}, S\rangle; T}\ \lfloor\text{TIn}\rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in K}{\Gamma \vdash c \oplus \langle \mathtt{p}, l_j\rangle; P \triangleright \Delta, c : \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k \in K}\rangle}\ \lfloor\text{TSel}\rfloor$$

$$\frac{\forall k \in K, \Gamma \vdash P_k \triangleright \Delta, c : T_k}{\Gamma \vdash c\&\langle \mathtt{p}, \{l_k : P_k\}_{k \in K}\rangle \triangleright \Delta, c : \&\langle \mathtt{p}, \{l_k : T_k\}_{k \in K}\rangle}\ \lfloor\text{TBra}\rfloor$$

$$\frac{\Gamma, a : U \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta}\ \lfloor\text{TNu}\rfloor \qquad \frac{\Gamma \vdash \Delta \quad \Delta\ \mathsf{end}\text{ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}\ \lfloor\text{TNull}\rfloor \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'}\ \lfloor\text{TPar}\rfloor$$

Fig. 24. Process typing (Part 2)

Subtyping and runtime typing rules Figure 25 presents the subtyping rules. Figure 26 presents the runtime typing rules omitted from the main sections. They are identical with [5].

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

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

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

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

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

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

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

Fig. 25. Subtyping

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\emptyset} P \triangleright \Delta} \lfloor\text{GInit}\rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Delta' \mathsf{end}\ \text{only}}{\Gamma \vdash_{\Sigma} P \triangleright \Delta * \Delta'} \lfloor\text{Weak}\rfloor$$

$$\frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Gamma \vdash_{\Sigma'} Q \triangleright \Delta' \quad \Sigma \cap \Sigma' = \emptyset}{\Gamma \vdash_{\Sigma\cup\Sigma'} P \mid Q \triangleright \Delta * \Delta'} \lfloor\text{GPar}\rfloor$$

$$\frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \mathsf{co}(\Delta, s)}{\Gamma \vdash_{\Sigma\setminus s} (\nu s)P \triangleright \Delta \setminus s} \lfloor\text{GSRes}\rfloor \qquad \frac{\Gamma, a : \langle G\rangle \vdash_{\Sigma} P \triangleright \Delta}{\Gamma \vdash_{\Sigma} (\nu a)P \triangleright \Delta} \lfloor\text{GNRes}\rfloor$$

Fig. 26. Run-time process typing

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

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

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

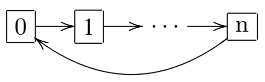
$$\frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta \quad j \in K}{\Gamma \vdash_{\{s\}} s : h \cdot (\mathtt{q}, \mathtt{p}, l_j) \triangleright \Delta; \{s[\mathtt{q}] : \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\}} \lfloor\text{QSel}\rfloor$$

Fig. 27. Queue typing

B Ring and mesh communication patterns

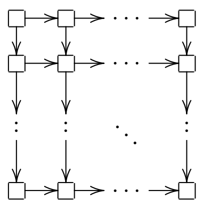
We present some programming examples of global types which make use of typical communication patterns that can be found in classical parallel algorithms textbooks.

(a) Ring pattern



$$
\begin{array}{l}
\mathit{\Pi} n : I. \\
\quad (\mathbf{R}\ \mathtt{W}[n] \to \mathtt{W}[0] : \langle U \rangle.\mathsf{end} \\
\qquad \lambda i.\lambda \mathbf{x}.\mathtt{W}[n-i-1] \to \mathtt{W}[n-i] : \langle U \rangle.\mathbf{x} \\
n)
\end{array}
$$

(b) Mesh pattern



$$
\begin{array}{l}
\mathit{\Pi} n.\mathit{\Pi} m. \\
\ (\mathbf{R} \\
\ \ (\mathbf{R}\ \mathsf{end}\ \lambda k.\lambda \mathbf{z}.\mathtt{W}[0][k+1] \to \mathtt{W}[0][k] : \langle \mathsf{nat} \rangle.\mathbf{z}\ m) \\
\ \ \lambda i.\lambda \mathbf{x}. \\
\quad (\mathbf{R}\ (\mathtt{W}[i+1][0] \to \mathtt{W}[i][0] : \langle \mathsf{nat} \rangle.\mathbf{x}) \\
\qquad \lambda j.\lambda \mathbf{y}. \\
\qquad\quad \mathtt{W}[i+1][j+1] \to \mathtt{W}[i][j+1] : \langle \mathsf{nat} \rangle. \\
\qquad\quad \mathtt{W}[i+1][j+1] \to \mathtt{W}[i+1][j] : \langle \mathsf{nat} \rangle.\mathbf{y} \\
\quad m) \\
\ n)
\end{array}
$$

Ring The ring pattern consists of $\mathrm{n}+1$ workers (named by W) that each has exactly two neighbours: the worker $\mathtt{W}[i]$ communicates with the worker $\mathtt{W}[i-1]$ and $\mathtt{W}[i+1]$ ($1 \leq i \leq \mathrm{n}-1$), with the exception of $\mathtt{W}[0]$ and $\mathtt{W}[\mathrm{n}]$ who share a direct link. The non-uniformity of linking between the workers in the ring gives three distinct roles: Starter, represented by $\mathtt{W}[0]$, Middle, represented by $\mathtt{W}[i]$ for $1 \leq i \leq \mathrm{n}-1$, and Last, represented by $\mathtt{W}[\mathrm{n}]$. The type specifies that the first message is sent by $\mathtt{W}[0]$ to $\mathtt{W}[1]$, and the last one is sent from $\mathtt{W}[\mathrm{n}]$ 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 $\mathrm{n} \geq 2$. The process that generates all the roles using a recursor is as follows:

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

Mesh The session from figure 2(c) describes communication over a two dimensional mesh pattern. The participants in the first and last rows and columns, except the corners which have two neighbors, have three neighbors. The other participants have four neighbors. The roles of the mesh are defined by the communication behavior of each participant and by the links the participants have with their neighbors. From the figure, it is easy to identify the nine roles present in the pattern. 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. These two messages specify the fact that each worker not situated on the last row or last column sends a message to his neighbors situated below and on its right. The types $\mathtt{W}[i+1][0] \to \mathtt{W}[i][0] : \langle \mathsf{nat} \rangle$ and $\mathbf{R}\ \mathsf{end}\ \lambda k.\lambda \mathbf{z}.\mathtt{W}[0][k+1] \to \mathtt{W}[0][k] : \langle \mathsf{nat} \rangle.\mathbf{z}\ m$ deal with, respectively, the last column and the last row. Variants of this pattern include toric meshes and hypercubes. To ensure the presence of all three roles in the workers of this topology, the parameter domain is set to $\mathrm{n}, \mathrm{m} \geq 2$.

The process that implements a parameterised mesh communication pattern is defined as follows. The processes having as second name "middle" define the elements in the middle of rows and columns, which place is defined by the first name.

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

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

B.1 Typing

Ring pattern - figure 2(a) The typing of the ring pattern is similar to the one of sequence. The general generator of this pattern is

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

and

the user would design the end-point type as follows:

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

The first case denotes the protocol of the initiator; the second one corresponds to the last worker, while the third one to one of the middle workers.

The type equality is the same as the sequence: proved by the case analysis by the induction of the recursor using the inductive rules in a version of the end-point types following in figure 23 ($\lfloor$WFBASE,WFLL,WFRECF$\rfloor$), combining the β-reductions over two end-point types.

From these types, the ring processes are straightforwardly typable using $\mathbf{R}$.

Let us define:

$$\begin{array}{ll}P_0 & = \bar{a}[\mathtt{W}[0],..,\mathtt{W}[\mathrm{n}]](y).y!\langle \mathtt{W}[1], v\rangle; y?(\mathtt{W}[n], z); P\\ P_n & = a[\mathtt{W}[n]](y).y?(\mathtt{W}[n-1], z); y!\langle \mathtt{W}[0], z\rangle; Q\\ P_{i+1} & = a[\mathtt{W}[i+1]](y).y?(\mathtt{W}[i], z); y!\langle \mathtt{W}[i+2], z\rangle; \qquad 1 \leq i+1 \leq n-1\end{array}$$

Assume

$$\emptyset \vdash P \triangleright \Delta_0$$

By $\lfloor$TOUT, TIN$\rfloor$, we have:

$$a : \langle G\rangle \vdash y!\langle \mathtt{W}[1], v\rangle; y?(\mathtt{W}[n], z); P \triangleright \Delta_n, y : G \upharpoonright \mathtt{p}_0$$

where $G \upharpoonright \mathtt{p}_0$ is obtained from the role type above. Hence, by $\lfloor$TINIT$\rfloor$, noting $\mathtt{p}_0 = \mathtt{W}[0]$,

$$a : \langle G\rangle \vdash P_0 \triangleright \Delta_0$$

Similarly, using $\lfloor$TACC$\rfloor$ with $\mathtt{p}_n = \mathtt{W}[n]$, we have:

$$a : \langle G\rangle \vdash P_n \triangleright \Delta_n$$

Hence we have

$$a : \langle G\rangle \vdash P_0 \mid P_n \triangleright \Delta$$

with $\Delta = \Delta_0, \Delta_n$. Similarly, we can type:

$$a : \langle G\rangle, i : I^-, 1 \leq i+1 \vdash P_{i+1} \triangleright \emptyset$$

with $I = [0..n-1]$. Also we have:

$$a : \langle G\rangle, i : I^-, X : \tau\{i/j\} \vdash X \triangleright \tau\{i+1/j\}$$

where $\tau\{i/j\} = \Delta$ for all i. Hence by strengthing and $\lfloor$TPAR$\rfloor$, we have:

$$a : \langle G\rangle, i : I^-, X : \tau\{i/j\} \vdash P_{i+1}|X \triangleright \tau\{i+1/j\}$$

Hence applying $\lfloor$TPREC$\rfloor$, we can type the ring process as:

$$a : \langle G\rangle \vdash P_{\text{ring}} \triangleright \Delta$$

as required.

Mesh pattern - figure 2(b) The highlight of this example is a type-equality between the general end-point generator and a *role-based* end-point type (called *role-types*). The role-type groups the participants who inhabit in the same parameterised protocol as a single role. We observe the mesh pattern consists of the nine roles (when $n, m \geq 2$). The participants in the first and last rows and columns, except the corners which have two neighbors, have three neighbors. The other participants have four neighbors. The roles of the mesh are defined by the communication behavior of each participant and by the links the participants have with their neighbors. The transformation starts from the general generator of this topology given below.

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

From the figure 2(b), the user would design the end-point type as follows:

```
if      p $=$ W[$n$][$m$] then $!\langle$W[$n-1$][$m$], nat$\rangle; !\langle$W[$n$][$m-1$], nat$\rangle$;
elseif p $=$ W[$n$][0] then $?\langle$W[$n$][1], nat$\rangle; !\langle$W[$n-1$][0], nat$\rangle$;
elseif p $=$ W[0][$m$] then $?\langle$W[1][$m$], nat$\rangle; !\langle$W[0][$m-1$], nat$\rangle$;
elseif p $=$ W[0][0] then $?\langle$W[1][0], nat$\rangle; ?\langle$W[0][1], nat$\rangle$;
elseif $1 \leq k+1 \leq m-1$ and p $=$ W[$n$][$k+1$]
        then $?\langle$W[$n$][$k+2$], nat$\rangle; !\langle$W[$n-1$][$k+1$], nat$\rangle; !\langle$W[$n$][$k$], nat$\rangle$;
elseif $1 \leq k+1 \leq m-1$ and p $=$ W[0][$k+1$]
        then $?\langle$W[1][$k+1$], nat$\rangle; ?\langle$W[0][$k+2$], nat$\rangle; !\langle$W[0][$k$], nat$\rangle$;
elseif $1 \leq i+1 \leq n-1$ and p $=$ W[$i+1$][$m$]
        then $?\langle$W[$i+2$][$m$], nat$\rangle; !\langle$W[$i$][$m$], nat$\rangle; !\langle$W[$i+1$][$m-1$], nat$\rangle$;
elseif $1 \leq i+1 \leq n-1$ and p $=$ W[$i+1$][0]
        then $?\langle$W[$i+2$][0], nat$\rangle; ?\langle$W[$i+1$][1], nat$\rangle; !\langle$W[$i$][0], nat$\rangle$;
elseif $1 \leq i+1 \leq n-1$ and $1 \leq j+1 \leq m-1$ and p $=$ W[$i+1$][$j+1$]
        then $?\langle$W[$i+2$][$j+1$], nat$\rangle; ?\langle$W[$i+1$][$j+2$], nat$\rangle; !\langle$W[$i$][$j+1$], nat$\rangle; !\langle$W[$i+1$][$j$], nat$\rangle$;
```

Each case denotes a role in the mesh pattern. Role names are given in process definition. We will present the typing of the two roles left-top-corner and bottom-row to give an idea of how our system types the mesh pattern. The other cases are left to the reader.

Let $T[\mathtt{p}][n][m]$ for the first original type and $T'[\mathtt{p}][n][m]$ for the second role type. We can check 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 left top corner role, the generator type reduces to one step and gives the end-point type $!\langle \mathtt{W}[\mathrm{n}-1][\mathrm{m}], \mathsf{nat}\rangle; !\langle \mathtt{W}[\mathrm{n}][\mathrm{m}-1], \mathsf{nat}\rangle;$, which is the same to the one returned by the case analysis of the role-type built by the programmer. For $\mathtt{p} = \mathtt{W}[0][k+1]$, we analyze the case when $\mathrm{n} = 2$ and $\mathrm{m} = 2$, where $1 \leq k + 1 \leq \mathrm{m} - 1$. The generator type returns the end-point type $?\langle \mathtt{W}[1][k+2], \mathsf{nat}\rangle; ?\langle \mathtt{W}[0][k+2], \mathsf{nat}\rangle; !\langle \mathtt{W}[0][k], \mathsf{nat}\rangle;$ where the first action comes from the case in line number 13, the second action comes from the case in line 2 and the third action comes from case in line 1. One can observe that the end-point type returned for $\mathtt{p} = \mathtt{W}[0][k+1]$ in the role-type of the programmer is the same as the one returned by the generator. Similarly for all the other cases. Note that for all n, m, the β-reduction terminates, so that the equality checking terminates.

By $\lfloor$TOUT, TIN$\rfloor$, we have:

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

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

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

The primitive recursive process is typed similarly as for the one of the sequence pattern.

C Proofs of the lemmas and theorems

C.1 Basic properties

We prove here a series of consistency lemmas concerning permutations and weakening of judgements follow. 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 C.1 *1. (Permutation and Weakening) Suppose $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash$ 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 $\Gamma \vdash$ 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) $\Gamma \vdash P \triangleright \tau$ 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* $\Gamma, \mathtt{P}', \Gamma' \vdash J$.

The next lemma will be important in the proof of Subject Reduction. 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.

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

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

C.2 Proofs for Propositions, Lemmas and Theorems in the main sections

Proposition C.3 (Termination and Confluence) *(restatement of 3.1) The head 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. Similarly the relation* $\longrightarrow$ *on global and end-point types (i.e.* $G \longrightarrow G'$ *and* $T \longrightarrow T'$ *for open types which reduce under recursive prefixes, abstractions and recursors) are strong normalising and confluent on well-formed kindings.*

Proof. By strong normalisation of System $\mathcal{T}$ [11]. For the confluence, we first prove that the relation $\longrightarrow$ 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. □

Proposition C.4 (Termination for Type-Equality Checking) *(cf. 3.2) Assuming that proving the judgements* $\Gamma \models J$ *appearing in type equality derivations is decidable (e.g. in* $\lfloor$KPROJ$\rfloor$*), then type-equality checking of* $\Gamma \vdash \alpha \equiv \beta$ *terminates.*

To prove termination of the second equality requires a careful analysis since the premises of the mathematical induction rules compare the two types whose sizes are lager than those of the conclusions. These rules correspond to the contextual congruence. The size of the judgements are carefully defined using the following four functions (figures 28 and 29).

1. $|G|$ is the size of (the structure of) G
2. $\mu^\circ(G)$ is the number of the head reductions from G.
3. $\mu(G)$ is the number of the head reductions from G and its subterms.
4. $\mu^\star(G)$ is the maximum number of the head reductions from G and its subterms. This includes the mathematical induction case where the dependent applications with variables as the arguments.

The termination then is proved by the following lemma.

Judgements	$w(\Gamma \vdash G_1 \equiv_{\mathrm{wf}} G_2) = \omega \cdot \mu(G_1) + \omega \cdot \mu(G_2) + \lvert G_1\rvert + \lvert G_2\rvert + 1$
	$w(\Gamma \vdash G_1 \equiv G_2) = w(\Gamma \vdash G_1 \equiv_{\mathrm{wf}} G_2) + 1$
Types	
Value	$\lvert\mathsf{bool}\rvert = \mathsf{nat} = 1, \lvert\langle G\rangle\rvert = \lvert G\rvert + 1$
Global	$\lvert \mathtt{p} \to \mathtt{p}' : \langle U\rangle.G\rvert = 2 + \lvert U\rvert + \lvert G\rvert, \quad \lvert \mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k\in K}\rvert = 2 + \Sigma_{k\in K}(1 + \lvert G_k\rvert)$
	$\lvert \mu \mathbf{t}.G\rvert = \lvert G\rvert + 1, \quad \lvert\mathbf{t}\rvert = \lvert\mathbf{x}\rvert = \lvert\mathsf{end}\rvert = 1$
	$\lvert \Pi i : I.G\rvert = 1 + \lvert G\rvert, \quad \lvert G\ i\rvert = \lvert G\rvert + 1$
	$\lvert \mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G'\rvert = 4 + \lvert G\rvert + \lvert G'\rvert$
Local	$\lvert !\langle \mathtt{p}, U\rangle; T\rvert = 3 + \lvert U\rvert + \lvert T\rvert, \quad \lvert ?\langle \mathtt{p}, U\rangle; T\rvert = 3 + \lvert U\rvert + \lvert T\rvert$
	$\lvert \oplus\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\rvert = \lvert \&\langle \mathtt{p}, \{l_k : T_k\}_{k\in K}\rangle\rvert = 2 + \Sigma_{k\in K}(1 + \lvert T_k\rvert)$
	$\lvert \mu \mathbf{t}.T\rvert = \lvert T\rvert + 1, \quad \lvert\mathbf{t}\rvert = \lvert\mathbf{x}\rvert = \lvert\mathsf{end}\rvert = 1$
	$\lvert \Pi i : I.T\rvert = 1 + \lvert T\rvert, \quad \lvert T\ \mathtt{i}\rvert = \lvert T\rvert + 1$
	$\lvert \mathbf{R}\ T\ \lambda i : I.\lambda \mathbf{x}.T'\rvert = 4 + \lvert T\rvert + \lvert T'\rvert$
Principals	$\lvert \Pi i : I.U_p\rvert = 1 + \lvert U_p\rvert$
Processes	$\lvert\emptyset\rvert = 0, \lvert \Delta, c : T\rvert = \lvert\Delta\rvert + \lvert T\rvert + 1$
	$\lvert \Pi i : I.\tau\rvert = 1 + \lvert\tau\rvert, \ \lvert \tau\ \mathtt{i}\rvert = 1 + \lvert\tau\rvert$

Fig. 28. Size of types and judgements

$$
\begin{array}{l}
\mu(\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G) = \mu(U) + \mu(G), \quad \mu(\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k\in K}) = \Sigma_{k\in K}\mu(G_k) \\
\mu(\mu \mathbf{t}.G) = \mu(G), \quad \mu(\mathbf{t}) = \mu(\mathbf{x}) = \mu(\mathsf{end}) = 0 \\
\mu(\Pi i : I.G) = \mu(G) \\
\mu(\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G') = \mu(G) + \mu(G') \\
\mu(G\ \mathrm{m}) = \mu^{\circ}(G\ \mathrm{m}) + \mu(\mathrm{whnf}(G\ \mathrm{m})) \\
\mu(G\ i) = \mu^{\star}(G) \\
\mu^{\star}(\Pi i : I.G) = \Sigma_{0\leq \mathrm{j} \leq \mathrm{n}+1}\ (\mu(G\ \mathtt{j}) + 1) + 1 \quad (I = [0..\mathrm{n}]) \\
\mu^{\star}(\Pi i : I.G) = \mu(\Pi i : I.G) \quad (I = \mathsf{nat}) \\
\mu^{\star}(\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G') = \Sigma_{0\leq \mathrm{j} \leq \mathrm{n}+1}\ (\mu(\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G'\mathtt{j}) + 1) + 1 \quad (I = [0..\mathrm{n}]) \\
\mu^{\star}(\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G') = \mu(\mathbf{R}\ G\ \lambda i : I.\lambda \mathbf{x}.G') \quad (I = \mathsf{nat}) \\
\mu^{\circ}(G) = n \quad \text{if } G \longrightarrow^{n} G' \not\longrightarrow
\end{array}
$$

Fig. 29. Number of reductions

Lemma C.5 (Size of Equality Judgements) *The weight of any premise of a rule is always less than the weight of the conclusion.*

We argue by induction of the length of reduction sequences and the size of terms. This is obvious for the rules in figure 22. The rule $\lfloor \text{WFBASE} \rfloor$ is by definition. Other rules use the definition of $\mu^\star$ directly.

Lemma C.6 (Soundness of mergeability) *(restatement of 3.4) 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 $G_1 \upharpoonright \mathtt{p}$ 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. $\square$

Theorem C.7 (Subject Congruence and Reduction) *(cf 3.5)*

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

Proof. We only list the crucial cases of the proof of subject reduction: the recursor (where mathematical induction is required) and the initialisation. 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 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

$$\Gamma, i : I^-, X : \tau\{i/j\} \vdash Q \triangleright \tau'\{i+1/j\} \tag{3}$$
$$\Gamma \vdash P \triangleright \tau'\{0/i\} \tag{4}$$
$$\Gamma \vdash \Pi j : I.\tau\ \blacktriangleright\ \Pi j : I.\kappa \tag{5}$$

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 C.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

$$\Gamma, X : \tau\{\mathrm{n}/j\} \vdash Q\{\mathrm{n}/i\} \triangleright \tau'\{\mathrm{n}+1/j\} \tag{6}$$

Then there are two cases.

Base Case $\mathrm{n} = 0$*:* By applying Substitution Lemma (Lemma C.2 (2)) to (6) with (4), 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 C.2) to (6) 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]

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

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

$$\forall i \neq 0, \Gamma \vdash \mathrm{p}_i \triangleright \mathsf{nat} \quad (7)$$

$$\Gamma \vdash a : \langle G \rangle \quad (8)$$

$$\Gamma \models \mathsf{pid}(G) = \{\mathrm{p}_0..\mathrm{p}_\mathrm{n}\} \quad (9)$$

$$\Gamma \vdash P \triangleright \Delta', y : G \upharpoonright \mathrm{p}_0 \quad (10)$$

From (10) and Lemma C.2 (4), $\Gamma \vdash P\{s[\mathrm{p}_0]/y\} \triangleright \Delta, s[\mathrm{p}_0] : G \upharpoonright \mathrm{p}_0$ (11)

From Lemma C.1 (3a) and $\lfloor$QINIT$\rfloor$, $\Gamma \vdash_s s : \epsilon \triangleright \emptyset$ (12)

From (7), (8), (9) and $\lfloor$TREQ$\rfloor$, $\forall i \neq 0, \Gamma \vdash \bar{a}[\mathrm{p}_i] : s \triangleright s[\mathrm{p}_i] : G \upharpoonright \mathrm{p}_i$ (13)

Then $\lfloor$TPAR$\rfloor$ on (11), (12) and (13) gives:

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

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

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

From Lemma 3.4 we know that $\mathsf{co}((s[\mathrm{p}_0] : G \upharpoonright \mathrm{p}_0, ..., s[\mathrm{p}_n] : G \upharpoonright \mathrm{p}_n), s)$. We can then use $\lfloor$GSRES$\rfloor$ to get:

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

We conclude from $\lfloor$TSUB$\rfloor$.

Case [Join]

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

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

$$\Gamma \vdash \bar{a}[\mathrm{p}] : s \triangleright s[\mathrm{p}] : G \upharpoonright \mathrm{p} \quad (14)$$

$$T \geq G \upharpoonright \mathrm{p} \quad (15)$$

$$\Gamma \vdash u[\mathrm{p}](y).P \triangleright \Delta' \quad (16)$$

By inversion of $\lfloor$TACC$\rfloor$ from (16) $\Gamma \vdash P \triangleright \Delta', y : G \upharpoonright \mathrm{p}$ (17)

From (17) and Lemma C.2 (4), $\Gamma \vdash P\{s[\mathrm{p}]/y\} \triangleright \Delta', s[\mathrm{p}] : G \upharpoonright \mathrm{p}$ (18)

We conclude by $\lfloor$TSUB$\rfloor$ from (18) and (15).

The proof of the progress is a corollary from this theorem. The detailed definitions (simple and well-linked) can be found in a full version of [12].

D The Fast Fourier Transform algorithm and proofs of Theorem 3.7

We first give more detailed explanations of the FFT algorithm, its global type and end-point processes in the FFT example in § 2.4. Then we prove Theorem 3.7 – the processes are typable against the given global type. Their termination (i.e. deadlock-freedom) is then obtained as a corollary.

D.1 More on the FFT

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

FFT and the butterfly network We present here the radix-2 variant of the Cooley-Tukey FFT algorithm [10]. Assuming that N is a power of 2, this FFT algorithm uses a divide-and-conquer strategy based on the following equation (we use the fact that $\omega_N^{2k} = \omega_{N/2}^k$):

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

Each of the two separate sums are DFT of half of the original vector members, separated into even and odd indices. 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 5(a) illustrates this recursive principle, called *butterfly*, where two different outputs can be computed in constant time from the results of the same two recursive calls.

The complete algorithm is illustrated by the diagram from figure 5(b). It features the application of the FFT on a network of $N = 2^3 = 8$ machines computing the DFT 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. When reading the diagram from right to left, each step consists in merging the results from half of the inputs, following in this the butterfly pattern: each machine is successively involved in a butterfly with a machine whose number differs by only one bit. Note that the recursive partition over the value of a different bit at each step requires a particular bit-reversed ordering of the input vector: the machine number p initially receives $x_{\overline{\mathsf{p}}}$ where $\overline{\mathsf{p}}$ denotes the bit-reversal of p.

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

Global Types Figure 5(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 iteration 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). 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 5(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 $\mathsf{bit}_{n-l}(\mathtt{p})$). 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: $r_{\mathtt{p}}!\langle 0, x\rangle$.

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

D.2 Proofs of Theorem 3.7

We assume index n to be a parameter as in figure 5. 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:

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

A simple induction gives us the equivalent type:

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

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

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

An induction over p and some simple arithmetic over binary numbers gives us the only two branches that can be taken:

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

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 $?\langle \mathrm{p}, U\rangle;$ from the initialisation to the beginning of the loop iteration at the price of adding the last self-receive to the end: $?\langle \mathrm{p}, U\rangle; \mathsf{end}$. The resulting equivalent type up to $\equiv$ is:

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

From this end-point type, it is straightforward to type and implement the processes defined in figure 5(d) in § 2.4.