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 [3, 5, 13], 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 [3, 13, 17], as follows (below end denotes protocol termination):

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

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:

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

one from `Alice`'s point of view, $!\langle\texttt{Bob}, \mathsf{nat}\rangle$ (output to `Bob` with nat-type), and another from `Bob`'s point of view, $?\langle\texttt{Alice}, \mathsf{nat}\rangle$ (input from `Alice` with nat-type), against which the respective `Alice` and `Bob` programs are checked to be compliant.

The first step towards generalised type structures for multiparty sessions is to allow modular specifications of protocols using arbitrary compositions and repetitions of interaction units (this is a standard requirement in multiparty contracts [21]). Consider the type $G_2 = \texttt{Bob} \to \texttt{Carol}\colon\langle\mathsf{nat}\rangle.\mathsf{end}$. The designer may wish to compose sequentially G_1 and G_2 together to build a larger protocol:

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

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

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

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 `foreach`, see § 2):

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

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 a decidable projection and static type-checking by which we can preserve the main properties of the session types such as progress and communication-safety in parameterised process topologies? The key technique proposed in this paper is a projection method from a dependent global type onto a *generic end-point generator* which exactly captures the interaction structures of parameterised end-points and which can represent the class of all possible end-point types.

The main contributions of this paper follow:

- *A new expressive framework to globally specify and program* a wide range of parametric communication protocols (§ 2). We achieve this result by combining dependent type theories derived from Gödel's System $\mathcal{T}$ [18] (for expressiveness) and indexed dependent types from [22] (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).

$\mathtt{i} ::= i \mid \mathrm{n} \mid \mathtt{i}\ \mathrm{op}\ \mathtt{i}'$	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
$\mathrm{K} ::= \{\mathrm{n}_0, \ldots, \mathrm{n}_k\}$	Finite integer set	$\mid\ \mathsf{end}$	Null

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

Fig. 1. Global types and type reduction

– *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 (§ 2.3, § 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 [19] (§ 3.6).

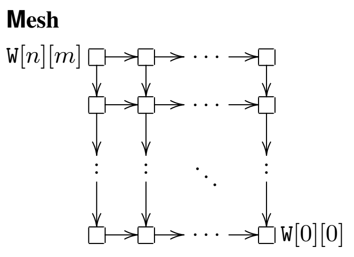
The complete formal definition of our system, including proofs and additional material for examples and implementations can be found in [11].

2 Types and processes for parameterised multiparty sessions

2.1 Global types

Global types allow the description of the parameterised conversations of multiparty sessions as a type signature. Our type syntax integrates three different formulations: (1) global types from [3]; (2) dependent types with primitive recursive combinators based on [18]; and (3) parameterised dependent types from a simplified Dependent ML [1, 22].

The grammar of global types ($G, G', \ldots$) is given in figure 1. *Parameterised principals* $\mathtt{p}, \mathtt{p}', \mathtt{q}, \ldots$ can be indexed by one or more parameters, e.g. $\mathtt{Worker}[5][i+1]$. Index $\mathtt{i}$ ranges over index variables i, j, n, naturals n or arithmetic operations. A global interaction can be a message exchange ($\mathtt{p} \to \mathtt{p}' : \langle U \rangle.G$), where $\mathtt{p}, \mathtt{p}'$ denote the sending and receiving principals, U the payload type of the message and G the subsequent interaction. Payload types U are either value types S (which contain base type nat and session channel types $\langle G \rangle$), or *end-point types* T (which correspond to the behaviour of one of the session participants and will be explained in § 3) for delegation. Branching ($\mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K}$) allows the session to follow one of the different G_k paths in



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

Fig. 2. Parameterised multiparty protocol on a mesh topology

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 $\mathcal{T}$ [12] whose reduction semantics is given in figure 1. Its parameters are a global type G, an index variable i with range I, a type variable for recursion $\mathbf{x}$ and a recursion body G'.[1] When applied to an index $\mathtt{i}$, its semantics corresponds to the repetition $\mathtt{i}$-times of the body G', with the index variable i value going down by one at each iteration, from $\mathtt{i}-1$ to 0. The final behaviour is given by G when the index reaches 0. The index sorts comprise the set of natural numbers and its restrictions by predicates $(\mathtt{P}, \mathtt{P}', ..)$ that are, in our case, conjunctions of inequalities. $\mathtt{op}$ represents first-order indices operators (such as $+$, $-$, $*$,...). We often omit I and end in our examples. Using $\mathbf{R}$, we define the product, composition, repetition and test operators (seen in § 1):

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

where we assume that $\mathbf{x}$ is not free in G and G_1, and that the leaves of the syntax trees of G_1 and G are end. These definitions rely on a special substitution of each end by $\mathbf{x}$ (for example, $\mathtt{p} \to \mathtt{p}'\{l_1{:}!\langle \mathsf{nat} \rangle; \mathsf{end}, l_2{:}\mathsf{end}\}\{\mathbf{x}/\mathsf{end}\} = \mathtt{p} \to \mathtt{p}'\{l_1{:}!\langle \mathsf{nat} \rangle; \mathbf{x}, l_2{:}\mathbf{x}\}$). The composition operator appends the execution of G_2 to G_1; the repetition operator above repeats G j-times[2]; the boolean values are integers 0 (false) and 1 (true). Similar syntactic sugar is defined for local types and processes. Note that composition and repetition do not necessarily impose sequentiality: only the order of the asynchronous messages and the possible dependency [13] between receivers and subsequent senders controls the sequentiality. A parallel version of the sequence example of (§ 1 (2)) can be written: $\Pi n.(\mathtt{foreach}(i < n)\{\mathtt{W}[n-i] \to \mathtt{W}[n-i-1] : \langle \mathsf{nat} \rangle\})$.

Mesh example The session presented in figure 2 describes a particular protocol over a standard mesh topology [15]. In this two dimensional array of workers $\mathtt{W}$, each worker receives messages from his left and top neighbours (if they exist) before sending messages to his right and bottom (if they exist). Our session takes two parameters n and m which represent the number of rows and the number of columns. Then we have two iterators that repeat $\mathtt{W}[i{+}1][j{+}1] \to \mathtt{W}[i][j{+}1] : \langle \mathsf{nat} \rangle$ and $\mathtt{W}[i{+}1][j{+}1] \to \mathtt{W}[i{+}1][j] : \langle \mathsf{nat} \rangle$

[1] We distinguish recursion and primitive recursion in order to get decidability results, see § 3.4.
[2] This version of `foreach` uses decreasing indices. One can write an increasing version [11].

$c ::= y \mid s[\mathtt{p}]$	Channels	$\hat{\mathtt{p}}, \hat{\mathtt{q}} ::= \hat{\mathtt{p}}[\mathrm{n}] \mid \mathcal{P}$	Principal values
$u ::= x \mid a$	Identifiers	$m ::= (\hat{\mathtt{q}},\hat{\mathtt{p}},v) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},s[\hat{\mathtt{p}}']) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},l)$	Messages in transit
$v ::= a \mid \mathrm{n}$	Values	$h ::= \epsilon \mid m \cdot h$	Queue types
$e ::= \mathtt{i} \mid v \mid x \mid s[\mathtt{p}] \mid e \operatorname{op} e'$	Expressions		

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

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

for all i and j. The communication flow goes from the top-left $\mathtt{W}[n][m]$ and converges towards the bottom-right $\mathtt{W}[0][0]$ in $n + m$ parallel message exchanges.

2.2 Process syntax and semantics

Syntax The syntax of expressions and processes is given in figure 3, extended from [3], adding the primitive recursion operator and a new request process. Identifiers u can be variables x or channel names a. Values v are either channels a or natural numbers n. Expressions e are built out of indices $\mathtt{i}$, values v, variables x, session end points (for delegation) and operations over expressions. In processes, sessions are asynchronously initiated by $\bar{u}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y).P$. It spawns, for each of the $\{\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}\}$, a request that is accepted by the participant through $u[\mathtt{p}](y).P$. Messages are sent by $c!\langle \mathtt{p}, e\rangle; P$ to the participant $\mathtt{p}$ and received by $c?\langle \mathtt{q}, x\rangle; P$ from the participant $\mathtt{q}$. Selection $c \oplus \langle \mathtt{p}, l\rangle; P$, and branching $c\&\langle \mathtt{q}, \{l_k : P_k\}_{k\in K}\rangle$, 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 $\mathtt{p}$ 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 e.g. $\mathtt{W}[3 + 1]$ to be reduced to $\mathtt{W}[4]$) is 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 identical to their global type counterparts. The rule [Init] describes the initialisation of a session by its first participant $\bar{a}[\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}](y_0).P_0$. It spawns asynchronous requests $\bar{a}[\hat{\mathtt{p}}_k] : s$ that allow delayed acceptance by the other session participants (rule [Join]). After the connection, the participants share the private session name s, and the queue associated to s (which

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

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

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

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

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

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

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

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

Fig. 4. Reduction rules

is initially empty by rule [Init]). The variables y_p in each participant p are then replaced with the corresponding session channel, $s[\mathrm{p}]$. A more verbose, but symmetric, version of [Init] (where any participant can start the session, not only p_0) could also be used [11].

The rest of the session reductions are standard [3, 13]. 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 $\mu X.P \equiv P\{\mu X.P/X\}$).

2.3 Processes for parameterised multiparty protocols

We give here the processes corresponding to the interactions described in § 1 and § 2.1, then introduce a parallel implementation of the Fast Fourier Transform algorithm.

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

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

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.

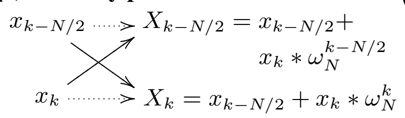
As an illustration of the semantics, we show the reduction of the above process for $n = 2$. After several applications of the [SuccR] and [ZeroR] rules, we have:

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

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

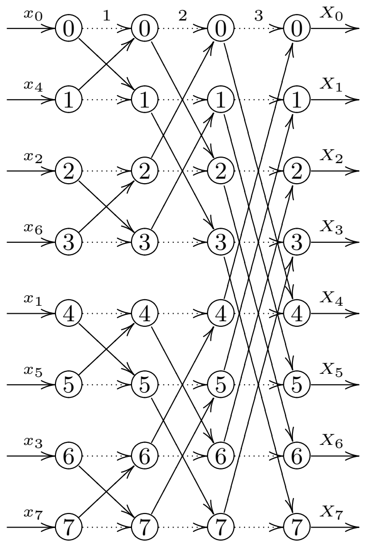
$$\begin{array}{ll} \longrightarrow & (\nu s)(s : \epsilon \mid s[\mathtt{W}[2]]!\langle \mathtt{W}[1], v\rangle; \mid \bar{a}[\mathtt{W}[1]] : s \mid \bar{a}[\mathtt{W}[0]] : s \mid \\ & \quad a[\mathtt{W}[0]](y).y?(\mathtt{W}[1], z); \mid a[\mathtt{W}[1]](y).y?(\mathtt{W}[2], z); y!\langle \mathtt{W}[0],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

(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 $P(n, \mathtt{p}, x_{\overline{\mathtt{p}}}, y, r_{\mathtt{p}}) =$

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

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. Below, $f(i, j)$ represents the expression computed at the (i, j)-th element.

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

FFT 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

$T ::=$	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	$\mid$ end	End

Fig. 6. End-point types

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

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

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 p initially receives $x_{\overline{\mathtt{p}}}$ where $\overline{\mathtt{p}}$ denotes the bit-reversal of 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

The syntax of end-point types is given in figure 6. Output expresses the sending to p of a value or channel of type U, followed by the interactions T. Selection represents the transmission to p of a label l_k chosen in $\{l_k\}_{k\in K}$ followed by T_k. Input and branching are their dual counterparts. The other types are similar to their global versions.

End-point projection: a generic projection The relation between end-point types and global types is formalised by the projection relation. Since the actual participant characteristics might only be determined at runtime, we cannot straightforwardly use the definition from [3, 13]. Instead, we rely on the expressive power of the primitive recursive operator: *a generic end-point projection of G onto* q, written $G \restriction \mathtt{q}$, represents the family of all the possible end-point types that a principal q can satisfy at run-time.

The general endpoint generator is defined in figure 7 using the derived construct if _ then _ else _. The projection $\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G \restriction \mathtt{q}$ leads to a case analysis: if the participant $\mathtt{q}$ is equal to $\mathtt{p}$, then the end-point type of $\mathtt{q}$ is an output of type U to $\mathtt{p}'$; if participant $\mathtt{q}$ is $\mathtt{p}'$ then $\mathtt{q}$ inputs U from $\mathtt{p}'$; else we skip the prefix. The first case corresponds to the possibility for the sender and receiver to be identical. Projecting the branching global type is similarly defined, but for the operator $\sqcup$ explained below. For 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 [13], which explains that naïve branching projection leads to inconsistent end-point types.

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

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

In our framework, this restriction is too strong since each branch may contain different parametric interaction patterns. To overcome this, 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:[3] 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:

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

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:

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

Then $\mathtt{W}[2]$ can have the type $\&\langle \mathtt{W}[1],\ \{\mathsf{ok} : \langle \mathtt{W}[1], \mathsf{bool} \rangle,\ \mathsf{quit} : \langle \mathtt{W}[1], \mathsf{nat} \rangle\}\rangle$ which could not be obtained through the original projection rule in [3, 13]. 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.

[3] The idea of mergeability is introduced informally in the tutorial paper [8].

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

Following [22], we assume given in the typing rules two semantically defined judgements: $\Gamma \models \mathtt{P}$ (predicate $\mathtt{P}$ is a consequence of Γ) and $\Gamma \models \mathtt{i} : I$ ($\mathtt{i} : I$ follows from the assumptions of Γ). We also inductively define well-formed types using a kind system [11]. The judgement $\Gamma \vdash U \blacktriangleright \kappa$ 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 $\Pi i : I.\kappa$. Well-formedness of a term $\mathtt{i}$ and $\mathtt{P}$ in Γ and environments is defined in the standard way [1].

3.3 Typing processes

We explain here (Figure 8) a selection of the process typing rules. Rules [TNAT] and [TVAR] are standard ($\Gamma \vdash \mathsf{Env}$ means that Γ is well-formed). For participants, we check their typing by [TID] and [TP] in a similar way as [22] where $\Gamma \vdash \kappa$ means kinding κ is well-formed. In [TPREC], 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 deals with the changed index range within the recursor body. More precisely, we first check τ's kind. Then we verify for the base case ($j = 0$) that P has type $\tau\{0/j\}$. Last, we check the more complex inductive case: Q should have type $\tau\{i + 1/j\}$ under the environment $\Gamma, i{:}I^-, X{:}\tau\{i/j\}$ where $\tau\{i/j\}$ of X means that X satisfies the predecessor's type (induction hypothesis). [TAPP] is the elimination rule for dependent types.

Since our types include dependent types and recursors, we need a notion of type equivalence. We extend the standard method of [1, §2] with the recursor: [WF] 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 and [PREC] says two recursors are equated if either (1) each subgraph is equated by $\equiv$, or (2) they reduce to the same normal forms when applied to a finite number of indices. Rule [TEQ] allows to type processes up-to type equivalence.

[TINIT] types a session initialisation on shared channel u, binding channel y and requiring participants $\{\mathtt{p}_0, .., \mathtt{p}_n\}$. The premise verifies that the type of y is the first projection of the global type G of u and that the participants in G (denoted by $\mathtt{pid}(G)$) can be semantically derived as $\{\mathtt{p}_0, .., \mathtt{p}_\mathrm{n}\}$. [TACC] allows to type the $\mathtt{p}$-th participant to the session initiated on u. The typing rule checks that the type of y is the $\mathtt{p}$-th projection of the global type G of u and that G is fully instantiated. The kind rule ensures that G is fully instantiated (i.e. G''s kind is Type). [TREQ] types the process that waits for an accept from a participant: its type corresponds to the end-point projection of G.

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

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

$$\frac{\begin{cases}\Gamma \vdash G_1 \equiv G_2 \quad \Gamma \vdash G_1' \equiv G_2' \qquad \text{or} \\ \Gamma \vdash \mathbf{R}\, G_1\ \lambda i{:}I.\lambda \mathbf{x}.G_1'\ \mathrm{n} \equiv \mathbf{R}\, G_2\ \lambda i{:}I.\lambda \mathbf{x}.G_2' \mathrm{n} \text{ with } \Gamma \models I = [0..\mathrm{m}], 0 \le \mathrm{n} \le \mathrm{m}\end{cases}}{\Gamma \vdash \mathbf{R}\, G_1\ \lambda i{:}I.\lambda \mathbf{x}.G_1' \equiv_{\mathrm{wf}} \mathbf{R}\, G_2\ \lambda i{:}I.\lambda \mathbf{x}.G_2'}\,[\textsc{PRec}]$$

$$\frac{\Gamma \vdash \mathrm{whnf}(G_1) \equiv_{\mathrm{wf}} \mathrm{whnf}(G_2)}{\Gamma \vdash G_1 \equiv G_2}\,[\textsc{WF}] \quad \frac{\Gamma \vdash P \triangleright \tau \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash P \triangleright \tau'}\,[\textsc{TEq}] \quad \frac{\Gamma \vdash P \triangleright \tau \quad \Gamma \vdash \tau \le \tau'}{\Gamma \vdash P \triangleright \tau'}\,[\textsc{TSub}]$$

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

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

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

Fig. 8. Process typing

Recursion [TRec], variable ([TVar]), output ([TOut]), input, delegation, inaction, branching/selection and the expression typing rules as well as the typing rules for queues are similar to those in [3, 13].

3.4 Properties of typing

Ensuring termination of type-checking with dependent types is not an easy task since type equivalences are often derived from term equivalences. We rely on the strong normalisation of System $\mathcal{T}$ [12] 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. $\Gamma \vdash P \rhd \emptyset$ terminates.*

Proof. (Outline) By the standard argument from indexed dependent types [1, 22], 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 $\lfloor\text{TEQ}\rfloor$, we include the type equality check into $\lfloor\text{TINIT,TREQ,TACC}\rfloor$ (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. $\square$

To ensure the termination of $\Gamma \models \mathtt{P}$, several solutions include the restriction of predicates to linear equalities over natural numbers without multiplication (or to other decidable arithmetic subsets) or the restriction of indices to finite domains, cf. [22].

3.5 Type soundness and progress

The following lemma states that mergeability is sound with respect to the branching subtyping $\leq$. By this, we can safely replace the third clause $\sqcup_{k\in K} G_k \upharpoonright \mathtt{q}$ of the branching case from the projection definition by $\sqcap\{T \mid \forall k \in K. T \leq (G_k \upharpoonright \mathtt{q})\}$. This allows us to prove subject reduction by including subsumption as done in [13].

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 $\leq$.*

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 $\Gamma \vdash P \rhd \tau$ and $P \longrightarrow^* P'$, then $\Gamma \vdash P' \rhd \tau'$ for some τ' such that $\tau \Rightarrow^* \tau'$.*

Note that communication safety [13, Theorem 5.5] and session fidelity [13, 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 [13, 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 $\Gamma \vdash^\star P \rhd \Delta$ 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 [13, Theorem 5.12].[4]

Theorem 3.6 (Progress) *If P is well-linked and without any element from the runtime syntax and $\Gamma \vdash^\star P \triangleright \emptyset$. Then for all $P \longrightarrow^* Q$, either $Q \equiv \mathbf{0}$ or $Q \longrightarrow R$ for some R.*

3.6 Typing examples

Repetition example - § 1 (1) This example illustrates the repetition of a message pattern. Let $G(n) = \texttt{foreach}(i < n)\{\texttt{Alice} \to \texttt{Bob}\colon \langle \mathsf{nat}\rangle.\texttt{Bob} \to \texttt{Carol}\colon \langle \mathsf{nat}\rangle\}$. Following the projection from figure 7, `Alice`'s end-point projection of $G(n)$ is[5]:

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

Let $\texttt{Alice}(n) = \bar{a}[\texttt{Alice}, \texttt{Bob}, \texttt{Carol}](y).(\mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X\ \ n)$ and let $\Delta(n) = \{y : (G(n) \upharpoonright \texttt{Alice})\}$ and $\Gamma = n : \mathsf{nat}, a : \langle G\rangle$. We can prove that $\Gamma \vdash \texttt{Alice}(n) \triangleright \emptyset$ from [TINIT] if we have $\Gamma \vdash \mathbf{R}\ \mathbf{0}\ \lambda i.\lambda X.y!\langle \texttt{Bob}, e[i]\rangle; X\ n \triangleright \Delta(n)$. This, in turn, is given by [TPREC] and [TAPP] from $\Gamma, i : I^-, X : \Delta(i) \vdash y!\langle \texttt{Bob}, e[i]\rangle; X \triangleright \Delta(i+1)$ and the trivial $\Gamma \vdash \mathbf{0} \triangleright y : \mathsf{end}$. From [TVAR], we have $\Gamma, i : I^-, X : \Delta(i) \vdash X \triangleright \Delta(i)$. We conclude by [TOUT] and weak head normal form equivalence [WF] of the types $\Delta(i+1)$ and $y\ :!\langle \texttt{Bob}, \mathsf{nat}\rangle; (\mathbf{R}\ \mathsf{end}\ \lambda j.\lambda \mathbf{x}.!\langle \texttt{Bob}, \mathsf{nat}\rangle; \mathbf{x}\ \ i)$. $\texttt{Bob}(n)$ and $\texttt{Carol}(n)$ can be similarly typed.

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

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

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 [WF] 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.

[4] We believe a stronger progress property for interleaved multiparty sessions ensured by the interaction typing in [3] can be obtained in this framework, too (since our typing system is an extension from the communication system in [3]).

[5] For readability, in the following examples, we omit from the nested conditionals the impossible cases.

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

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

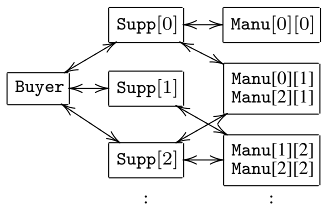
As we reasoned above, each $P(n, \mathtt{p}, x_{\overline{\mathtt{p}}}, y, r_{\mathtt{p}})$ is straightforwardly typable by an end-point type which is equivalent with the one projected from the global type G from figure 5(c). Automatically checking the equivalence for all n is not easy though: we need to rely on the finite domain restriction using [WF,PREC]. The following theorem says once P_{fft} is applied to a natural number m, its evaluation always terminates with the answer at $r_{\mathtt{p}}$. The proof is by the progress (Theorem 3.5), noting $P_{\text{fft}}\ \mathrm{m}$ is typable by a single, multiparty dependent session (except the answering channel 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 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* $r_{\mathtt{p}}$.

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 [19], the public document authored by the W3C Choreography Description Language Working Group [21]. 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 [19] include the ability to *reference a global description from within a global description* to support *recursive behaviour* as denoted in STEP 4(b, d): this 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 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, 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 $G = \mathit{\Pi} i.\mathit{\Pi}\tilde{J}.(G_1\,;\,\mu\mathbf{t}.(G_2\,;\,G_3))$ 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 $\lfloor$TEQ$\rfloor$, we can obtain the projection written in Figure 9. To tell the other suppliers whether the



1. A buyer requests a quote from a set of suppliers.
   $G_1 = \mathtt{foreach}(i < n)\{\mathtt{Buyer} \to \mathtt{Supp}[i] : \langle \mathsf{Quote} \rangle\}$
2. All suppliers receive the request to ask their respective manufacturers for a bill of material items. The suppliers interact with their manufacturers to build their quotes for the buyer.
   $G_2(i) = \mathtt{foreach}(j : J_i)\{\mathtt{Supp}[i] \to \mathtt{Manu}[i][j] : \langle \mathsf{Item} \rangle . \ \mathtt{Manu}[i][j] \to \mathtt{Supp}[i] : \langle \mathsf{Quote} \rangle\}$
   The eventual quote is sent back to the buyer.
   $G_2 = \mathtt{foreach}(i : I)\{G_2(i); \mathtt{Supp}[i] \to \mathtt{Buyer} : \langle \mathsf{Quote} \rangle\}$
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. 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.

$$\begin{array}{lll}
\multicolumn{3}{l}{G_3 = \mathbf{R}\ \mathbf{t}\ \lambda i.\lambda \mathbf{y}.\mathtt{Buyer} \to \mathtt{Supp}[i] : \{} \\
\mathsf{ok} : & \mathsf{end} & \\
\mathsf{modify} : & \multicolumn{2}{l}{\mathtt{Buyer} \to \mathtt{Supp}[i] : \langle \mathsf{Quote} \rangle .} \\
 & \mathtt{Supp}[i] \to \mathtt{Buyer} : \{\mathsf{ok} : & \mathsf{end} \\
 & \qquad\qquad \mathsf{retryStep3} : & \mathbf{y} \\
 & \qquad\qquad \mathsf{reject} : & \mathsf{end}\}\}\ i
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{G_3 \upharpoonright \mathtt{Supp}[\mathrm{n}] = \&\langle \mathtt{Buyer}, \{} \\
\mathsf{ok} : & \mathsf{end} & \\
\mathsf{modify} : & \multicolumn{2}{l}{?\langle \mathtt{Buyer}, \mathsf{Quote} \rangle ; \oplus \langle \mathtt{Buyer}, \{} \\
 & \qquad \mathsf{ok} : & \mathsf{end} \\
 & \qquad \mathsf{retryStep3} : & \mathbf{y} \\
 & \qquad \mathsf{reject} : & \mathsf{end}\}\rangle\}\rangle
\end{array}$$

Fig. 9. The Quote Request use case (C-U-002) [19] with the corresponding global types

loop is being reiterated or if it is finished, we can simply insert the following closing notification $\mathtt{foreach}(j \in I \setminus i)\{\mathtt{Buyer} \to \mathtt{Supp}[j] : \{\mathsf{close} :\}\}$ before each end, and a similar retry notification (with label 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 $\mathtt{Supp}[i]$).

We have explored the impact of parameterised type structures for communications through implementations of the above use case and of several parallel algorithms in Java with session types [14], including the Jacobi method (with sequence and mesh topologies) and the FFT. We observe (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; and (2) a performance benefit against the original binary session version, reducing the overhead of multiple binary session establishments (see [11]). Full implementation and integration of our theory into [4, 14] is on-going work.

4 Related work

Dependent types The first use of primitive recursive functionals for dependent types is in Nelson's $\mathcal{T}^\pi$ [18] for the λ-calculus, which is a finite representation of $\mathcal{T}^\infty$ by Tait and Martin Löf [16, 20]. $\mathcal{T}^\pi$ can type functions previously untypable in ML, and the finite representability of dependent types makes it possible to have a type-reconstruction algorithm. We also use the ideas from the DML's dependent typing system in [1, 22] 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 The work [7] presented an *executable global processes* for Web interactions based on the binary session types. Our work provides flexible, programmable global descriptions as *types*, offering a progress for parameterised multiparty session, which is not ensured in [7]. Recent formalisms for typing multiparty interactions include [6, 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 (interleaving, choices) of a protocol. The most complex example of [9, § 3] (a group key agreement protocol from [2]), which is typed as π-processes with delegations, can be specified and articulated by a single parameterised global session type as:

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

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

Conversation Calculus [6] supports the dynamic joining and leaving of participants. Though the formalism in § 2.2 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.

Acknowledgements. We thank the reviewers, Lasse Nielsen and Dimitris Mostrous for their useful comments for the paper and Kohei Honda for discussions.

References

1. D. Aspinall and M. Hofmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT, 2005.
2. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 17–26, New York, NY, USA, 1998. ACM.
3. L. Bettini et al. Global progress in dynamically interfered multiparty sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433, 2008.
4. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.
5. E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.
6. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
7. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17, 2007.
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. Online Appendix. `http://www.doc.ic.ac.uk/~yoshida/dependent/`.
12. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1989.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
14. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
15. F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1991.
16. P. Martin-Lőf. Infinite terms and a system of natural deduction. In *Compositio Mathematica*, pages 93–103. Wolters-Noordhoof, 1972.
17. 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.
18. N. Nelson. Primitive recursive functionals with dependent types. In *MFPS*, volume 598 of *LNCS*, pages 125–143, 1991.
19. Web Services Choreography Requirements (No. 11). `http://www.w3.org/TR/ws-chor-reqs`.
20. W. W. Tait. Infinitely long terms of transfinite type. In *Formal Systems and Recursive Functions*, pages 177–185. North Holland, 1965.
21. Web Services Choreography Working Group. Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
22. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.