Practical Parameterised Session Types

Andi Bejleri

Department of Computing, Imperial College London

Abstract. *Parameterised session types* is a type theory studied in the context of *multiparty session types*, that addresses statically the problem of type-safe, deadlock-free interactions in programs of an arbitrary number of processes. The previous work supporting parameterised session types has several shortfalls that limit their utility in practice. We eliminate the shortfalls by introducing a programming idiom of *roles* and a new type system. Roles have the same design as classes in languages such as Java and C#, while the previous model presents an amorphous syntax without concepts on how to incorporate parameterised session types into a mainstream language. The previous model requires programmers to write processes types, in addition to global types, for type-checking, while this model preserves multiparty's lightweight type annotations and type-checking strategy of simply global types. The previous model requires values of parameters to range over finite sets of natural numbers, while this model allows infinite sets of them.

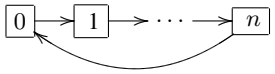
1 Introduction

It is well-known that message-based communication constitutes a prime element in the development of applications, namely web services, business protocols, parallel algorithms, multi-core programming, data centers management systems. This has motivated a vast amount of research into techniques, typically type-systems, for guaranteeing communication-safety. Among them, *multiparty session types* [16] is a type theory that addresses statically the problem of type-safe, deadlock-free interactions among a fixed number of processes. *Intuitive syntax of types* and *efficient type-checking strategy* are the main benefits that make multiparty session types stand out from the other systems. A notion of *global type* is introduced through an intuitive syntax to describe the interaction structure between the processes, that is defined by the "sending-receiving" actions in the presence of conditionals and recursion, from a global point of view. Processes are then efficiently validated by type-checking, through the *projection* of global types onto each participant. Whereby, this theory excels when given the number of participants. Unfortunately, in parallel algorithms and other communication-based applications the number of participants is known only at run-time; e.g. in parallel algorithms, the number of processes assigned to compute the answer of a problem instance is in proportion to its size.

Recently, the idea of *parameterised session types* [21] is studied in the context of multiparty session types. With parameterised session types, a global type can describe the communication pattern of an arbitrary number of participants. Communication patterns describe simple and elegant structured interactions in communication based applications. They are used in many parallel computing architectures of parallel algorithms [14], exchange protocols [4] and web-services [20]. Communication patterns, as design

patterns, help programmers to design more modular and more understandable system architectures. Common communication patterns are Ring, Tree, Mesh and Hypercube.

A global type includes the $\mathbf{R}$ operator from Gödel's theory *T* [1] to iterate over parameterised causalities that abstract the repetitive behavior of a pattern and to compose sequentially global types, where parameterised causalities are defined over parameterised principals. For example, in the Ring pattern,



the causality that abstracts the communications from `0` to `n` is $\texttt{i} \rightarrow \texttt{i+1}{:}\langle U \rangle$ with some abuse of notation, where $0 \leq i \leq n-1$ and $n \geq 1$. Given the number of participants, the $\mathbf{R}$ operator will iterate over the parameterised causality, and then the global type created will be composed with the causality $\texttt{n} \rightarrow \texttt{0} : \langle U \rangle$ to complete an instance of the Ring.

The syntax of processes includes also the $\mathbf{R}$ operator to parameterise participants, to iterate over processes that share the same behavior and to compose in parallel processes. For example, in the Ring, there are three kinds of participants: the first one is `0` which sends to the participant on his left (`1`) and then receives from the last participant (`n`), the second one is `i` for $1 \leq i \leq n-1$ that receives from the participants on his right ($\texttt{i}-1$) and then sends to the one on his left ($\texttt{i}+1$) and finally, the last participant `n`, which receives from the participant on his right ($\texttt{n}-1$) and then sends to the first participant (`0`). The $\mathbf{R}$ operator will iterate over `i`, returning on each iteration processes that share the same behavior, and then will compose them in parallel with the processes of `0` and `n`.

Despite, this strong initial work, parameterised session types have several shortfalls that limit their utility in practice. First, the syntax of processes is amorphous, without concepts to design a library that supports communication of mainstream languages. Second, the type system requires *programmers to write the processes types*, in addition to the global type, for type-checking. The coherence of the process types with respect to the global type is ensured by an *equivalence relation* for every value of each parameter present in the global type. The former aspect of the type system increases the programming effort and the latter diverges from the efficient typing strategy of the fundamental work of global session types [16], where the global type is projected onto participants to obtain the types for typechecking of processes. Third, the type system restricts the computing power of programs, allowing values of parameters to range over finite sets of natural numbers, e.g. parameter $n{:}\{m{:}\mathsf{nat} \mid 0 \leq m \leq 1000\}$ is typed by a bounded set of natural numbers. Finite sets are necessary to provide decidability for the type system, as termination of the equivalence algorithm depends on those sets' cardinality. The upper-bound of the set reflects the maximum capacity of the hardware resources to program date, not matching the purpose of innovative dynamic computing platforms such as clouds where hardware resources flex in response to demand.

This paper eliminates these shortfalls by introducing a programming idiom of *roles* and a new type system. Contributions of this work include:

- *A role defines an abstraction of communication's end-points in mobile processes. It is a blueprint that describes the nature of a communication pattern and the behavior that all run-time processes will share.* The syntactic extension is small but yet

provides a similar concept to classes in class-based languages and so, offers a design on how to incorporate parameterised session types into a mainstream language such as Java and C#.
- A static type system that follows the efficient typing strategy and programming methodology of multiparty session types: programmers first define the global type of the intended pattern and then define each role of it; roles are validated through projection of the global type onto the principals by type-checking. We achieve strict global type annotations in programs and efficient type-checking by extending the multiparty's projection algorithm to parameterised principals.
- Values of parameters range over infinite sets of natural numbers. We use infinite sets to provide full computation power of programs that implement parameterised communication patterns.
- Examples that show how this system can represent various communication patterns and control one of the main sources of programming errors in MPI (a message-passing API to program parallel computers). We illustrate the practical utility of our system through real-world examples from parallel algorithms and key distribution protocol.

Section 2 discusses our syntax of roles, illustrated by the Ring example, and operational semantics[1]. Section 3 gives the syntax of global types for parameterised communication patterns, illustrated by the Ring and Tree patterns. Section 4 defines our typing system, and proofs of decidability and subject reduction. Section 5 describes two real-world examples from parallel algorithms and data exchange protocols. Section 6 surveys related work and section 7 concludes with a discussion of possible future work for this system.

2 Roles

Our system extends that of Bettini *et al.* [7], preserving multiparty's programming methodology and typing strategy. Channels are omitted from the syntax of processes [7], serving a simpler type system than the one introduced by Honda *et al.* [16].

Syntax Figure 1 provides the syntax of our calculus. A program λn.E is a function from naturals (the number of participants) to roles composed in parallel. Roles in our calculus are second-class constructs; they can not be computed by functions. This contrasts the design of the previous work [21], where both functions and processes are considered as runtime entities and of the same programming-idiom class. Each role defines a scope that includes the subsequent behaviors. The prefix $\bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R$ represents the behavior of the first principal in the list (possibly parameterised) of principals $\mathtt{p}_0, \mathtt{p}_1, p$ and the process of that role initiates a session with the acceptor processes of shape $u[\mathtt{p}](y).R$ of principals $\mathtt{p}_1$ and p. The prefixes represent the abstract notion of session establishment of an arbitrary number of principals in minimal syntax. The sending construct $c!\langle \mathtt{p}, e\rangle; R$ denotes the action of sending a value to participant $\mathtt{p}$; the receiving construct $c?\langle \mathtt{p}, x\rangle; R$ denotes the action of receiving a value from $\mathtt{p}$. A similar notation

[1] For space reasons, we present only part of the formal model definition, related with essential features of the system and contributions of this work. The full definition of the formal model can be found in a companion technical report [5].

$\mathrm{E} ::= \lambda n.\mathrm{E} \mid \mathrm{E}\ \mathtt{t} \mid R$ General expressions

$R, S ::=$	Roles	$\mid (\nu w)R$	Hiding
$\mid \bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R$	Multicast request	$\mid \mathbf{0}$	Inaction
$\mid u[\mathtt{p}](y).R$	Accept	$\mid R \mid S$	Parallel
$\mid c!\langle \mathtt{p}, e\rangle; R$	Value sending	$\mid \mathbf{R}\ S\ \lambda i.\lambda X.R$	Primitive recursion
$\mid c?\langle \mathtt{p}, x\rangle; R$	Value reception	$\mid X$	Process variable
$\mid c \oplus \langle \mathtt{p}, l\rangle; R$	Selection	$\mid R\ \mathtt{t}$	Application
$\mid c\&\langle \mathtt{p}, \{l_i : R_i\}_{i\in I}\rangle$	Branching	$\mid s{:}h$	Queues

$u ::= x \mid a$	Identifiers	$c ::= y \mid s[\hat{\mathtt{p}}]$	Channels
$p ::= \mathtt{p}_1..\mathtt{p}_\mathrm{n} \mid \mathbf{R}\ p\ \lambda i.\lambda X.p'\ \mathtt{t} \mid X$	List of prin.	$h ::= \epsilon \mid m \cdot h$	Queues
$e ::= \mathtt{t} \mid v \mid e \circ\mathtt{p}\ e'$	Expressions	$m ::= (\hat{\mathtt{q}},\hat{\mathtt{p}},v) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},l)$	Msg. in transit
$v ::= a \mid s[\hat{\mathtt{p}}] \mid \mathrm{n} \mid \mathsf{true} \mid \mathsf{false}$	Values	$\hat{\mathtt{p}}, \hat{\mathtt{q}} ::= \hat{\mathtt{p}}[\mathrm{n}] \mid \mathcal{N}$	Value principal

Fig. 1. Syntax for roles and run-time processes

is used in selection-branching of a label, $c\oplus\langle \mathtt{p}, l\rangle; R$ and $c\&\langle \mathtt{p}, \{l_i : R_i\}_{i\in I}\rangle$, where the former selects one of the labels enumerated in I and sends it to the later. $\nu w.R$ restricts w to R. Parallel composition is standard.

We use the $\mathbf{R}$ operator from System T as in the previous work, to parameterise principals, to iterate and compose in parallel roles. The operator composes a lambda expression, $\lambda i.\lambda X.R$, with another expression, S. The recursive operator can be used also inside the definition of a role to iterate over a particular end-point behavior. Iteration takes place when a natural number is applied to a primitive recursion term, $R\ \mathtt{t}$.

The message queue $s : h$ represents ordered messages in transit to model TCP-like asynchronous communications. Identifiers u can be variables or shared names. A list of principals can be constant or parameterised using the $\mathbf{R}$ operator. The mathematical definition of principals list is missing from the previous work, weakening the properties of the type system as we shall see later. Expressions include parametric mathematical expressions (see Section 3), values and operations such as $e = e'$, e and e' and not e. Values are defined over shared names, session channels, naturals and boolean values. Channels denote channel variables or session channels. Session channel $s[\hat{\mathtt{p}}]$ denotes the channel of the participant $\hat{\mathtt{p}}$ in the session s. Messages in queues are defined as triples: sender, receiver and data (value or label). Messages are run-time entities, therefore they are defined over value principals. Value principals are the same as in the previous work, including participants (`Bob`, `Alice`, ...) or indexed principals over naturals (`W`[3], `W`[2][4], ...).

Operational Semantics Figure 2 gives the operational semantics via the reduction relation $\longrightarrow$. The interesting features of the rules are how they invoke a program, start a session, instantiate roles, iterate over end-point behavior and exchange messages.

The rule [App] invokes a program by replacing the parameter n with the argument n, as it instantiates roles which are parameterised only by n. Rule [Zero] returns the behavior S and defines the last iteration of the $\mathbf{R}$ operator. Rule [Succ] replaces each occurrence of the index i in R with a predecessor of $\mathrm{n}{+}1$ and replaces X with instances of R returned by the other iterations. When R denotes roles, [Succ] instantiates them

$$(\lambda n.E)\ \mathrm{n} \longrightarrow E\{\mathrm{n}/n\} \quad \text{[App]}$$

$$\mathbf{R}\ S\ \lambda i.\lambda X.R\ 0 \longrightarrow S \quad \text{[Zero]}$$

$$\mathbf{R}\ S\ \lambda i.\lambda X.R\ (\mathrm{n}+1) \longrightarrow R\{\mathrm{n}/i\}\{\mathbf{R}\ S\ \lambda i.\lambda X.R\ \mathrm{n}/X\} \quad \text{[Succ]}$$

$$\begin{array}{c}\bar{a}[\hat{\mathtt{p}}_0..\hat{\mathtt{p}}_\mathrm{n}](y_0).R_0 \mid a[\hat{\mathtt{p}}_1](y_1).R_1 \mid ...a[\hat{\mathtt{p}}_\mathrm{n}](y_\mathrm{n}).R_\mathrm{n} \\ \longrightarrow (\nu s)(R_0\{s[\hat{\mathtt{p}}_0]/y_0\} \mid ... \mid R_\mathrm{n}\{s[\hat{\mathtt{p}}_\mathrm{n}]/y_\mathrm{n}\} \mid s : \emptyset)\end{array} \quad \text{[Link]}$$

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

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

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

$$s[\hat{\mathtt{p}}]\&(\hat{\mathtt{q}}, \{l_i : R_i\}_{i\in I})\ \mid s : (\hat{\mathtt{q}}, \hat{\mathtt{p}}, l_{i_0}) \cdot h \longrightarrow R_{i_0} \mid s : h \quad (i_0 \in I) \quad \text{[Branch]}$$

Fig. 2. Reduction rules

in each iteration and composes them in parallel. Otherwise R denotes an end-point behavior that [Succ] iterates when the session has been established.

A session is established among processes via shared channels (a) that denote public points of communication. At this point, every role has been instantiated into processes and the computation follows over value principals. The rule [Link] invokes a session between n peers by generating a fresh session channel s to perform a series of communications and the associated empty session queue, and substitutes the channel into the processes scope. The identity of each principal within a session is represented by the label $\hat{\mathtt{p}}$ in the session channel $s[\hat{\mathtt{p}}]$. The n-party synchronisation captures a handshake among the n participants to establish a n-party link in real-world protocols. The [Send] and [Label] rules insert a message in the tail of the session queue. The receiving rule [Recv] removes a value message of the same sender, as the one specified in the receiving construct, from the head of the queue, and substitutes it in the process. The [Branch] rule removes from the queue a label message of the same sender, as the ones specified in the branching construct. The result of the rule is the process following the label.

Ring The Ring pattern, described in the introduction, consists of $n+1$ workers (named by $\mathtt{W}$) where each has exactly two neighbours: the worker $\mathtt{W}[j]$ communicates with the workers $\mathtt{W}[j-1]$ and $\mathtt{W}[j+1]$ ($1 \leq j \leq \mathrm{n}-1$), with the exception of $\mathtt{W}[0]$ who communicates with $\mathtt{W}[n]$ and $\mathtt{W}[1]$, and $\mathtt{W}[\mathrm{n}]$ with $\mathtt{W}[n-1]$ and $\mathtt{W}[0]$. Due to the enumeration of workers in a non-modular arithmetic, the Ring has three distinct roles: *Starter*, represented by $\mathtt{W}[0]$, *Middle*, represented by $\mathtt{W}[j]$, and *Last*, represented by $\mathtt{W}[\mathrm{n}]$. To ensure the presence of all three roles in a session, we set the number of participants to $n \geq 2$ but we do not set any upper-bound, which would be the case in the previous work [21]. Below, we provide the main program and roles of the Ring:

$$\begin{array}{l}\texttt{def}\ \ \mathtt{W} = \mathbf{R}\ \mathtt{W}[n]\ \lambda i.\lambda X.\mathtt{W}[i+2], X\ (n-2) \\ \textit{Starter} \triangleq \bar{a}[\mathtt{W}[0], \mathtt{W}[1], \mathtt{W}](y).y!\langle\mathtt{W}[1], v\rangle; y?(\mathtt{W}[n], z); R \\ \textit{Middle(i)} \triangleq a[\mathtt{W}[i+1]](y).y?(\mathtt{W}[i], z); y!\langle\mathtt{W}[i+2], z\rangle; R' \\ \textit{Last} \triangleq a[\mathtt{W}[n]](y).y?(\mathtt{W}[n-1], z); y!\langle\mathtt{W}[0], z\rangle; S \\ \mathrm{Ring} \triangleq \lambda n.((\mathbf{R}\ \textit{Starter} \mid \textit{Last}\ \lambda i.\lambda X.\textit{Middle(i)} \mid X)\ (n-1))\end{array}$$

where *W* denotes the parameterised list of principals $\mathtt{W}[2], ..., \mathtt{W}[n]$ mathematically represented through the $\mathbf{R}$ operator, and *Starter* and *Last* are parameterised by n and *Middle* by i. *Middle* is composed in parallel with the process variable X that is used as a placeholder of processes generated in each iteration; in the last iteration for $n{=}0$, X will be replaced with processes of *Starter* and *Last*.

Below, we give the implementation of the *Starter* role in Java. The role is naturally implemented in a particular class and so would the other roles, avoiding the MPI style of programming Single Program Multiple Data.

```
public class Starter{
   public Starter(int port_l, String host_r, int port_r){
      // Set up the sockets for the pattern.
      ServerSocket serverSocket = null;
      Socket clientSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;
      try{
          serverSocket = new ServerSocket(port_l);
          clientSocket = new Socket(host_r, port_r);
          out = ...; // Init. the output stream on clientSocket.
          in = ...; // Init. the input stream on serverSocket.
          // Exchange messages with neighbors.
          out.println("1");
          String m = in.readline();
          ... // close streams and sockets, capture exceptions.
   }
}
```

The class `Starter`, similarly to the role, provides a communication abstraction of the session object that will be generated at run-time given the values for the parameters `port_l`, `host_r`, `port_r`. The parameters abstract the neighbors' address and ports, similarly as n in the role definition. The feature that does not match the Java model with that of our calculus is the communication abstraction. Java uses client-server sockets as communication abstraction, while in our calculus, we use session channels s to define communication between a group of processes, including the n-party handshake. Thus, a Java library that supports communication over a group of processes, would provide the proper features to model roles and session channels, therefore the proper framework to implement parameterised session types.

3 Global Types

Global types [16] describe the interaction structure of a fixed number of processes from a global point of view. With parameterised session types, a global type can describe the communication pattern of an arbitrary number of participants.

3.1 Global Types for Parameterised Communication Patterns

Figure 3 gives the syntax of global types. A message of type U is exchanged between two principals, $\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G$, where $\mathtt{p}$ and $\mathtt{p}'$ are respectively the sender and the

receiver. Branching is defined over labels which identify the paths of a conversation, $\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i \in I}$; i.e. participant $\mathtt{p}$ internally chooses one of the labels l_i enumerated by I and then sends it to participant $\mathtt{p}'$ and the conversation follows G_i. Infinite behavior is represented by recursively defined global types $\mu \mathbf{t}.G$. end signifies the end of a conversation.

The $\mathbf{R}$ operator is added to the syntax of global types to describe communication patterns of an arbitrary number of principals, as in the previous work. The parameters that abstract the number of participants are bound by the binders in the lambda expressions of roles, since both global type and roles are part of the program definition. This contrasts the design of the previous work where a special binder Π is introduced in the global type syntax, prohibiting syntactically the relation between the number of participants in global types and programs. Throughout the paper we will refer to primitive recursive global types as *product global types*, as they abstract all instances of the parameterised global type. The infinite set of instances generated from the $\mathbf{R}$ operator can be understood through the two reduction rules:

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

For each natural, we obtain a global type by applying the two rules. In each iteration, the index variable in G' is substituted by a predecessor of $n{+}1$ and $\mathbf{x}$ is replaced by instances of the parameterised causalities present in G', except 0 when $\mathbf{x}$ is replaced by instances of G.

Principals $\mathtt{p}, \mathtt{p}', \mathtt{q}, ..$ include primitive participants `Alice`, `Bob`, ... and indexed principals defined over one or multiple index expressions $\mathtt{W}[i]$, $\mathtt{W}[i{+}1][j{+}1]$, ... Index expressions $\mathtt{i}$ are represented by parametric linear functions, where n ranges over naturals, i ranges over index variables and $\mathtt{t}$ ranges over parametric expressions. Parametric expressions range over variables n, naturals n, arithmetical operations ($\mathtt{t}{+}\mathrm{n}$, $\mathtt{t}{-}\mathrm{n}$, $\mathtt{t}{*}\mathrm{n}$) and exponentiation of base natural. The index expressions of the previous work [21] are more general than in this system, including a more sophisticated mathematical definition and more than one index variable per index expression. This expressivity comes at the cost of having values of parameters to range over finite sets of naturals in the conservative type system. Our design of index expressions as parametric linear functions comes from the observation that the information flow follows a straight line, neither a curve nor other forms of line, in the patterns/virtual-topologies we have studied so far, namely Ring, Star, Tree and Mesh. For simplicity and without reducing the practical expressiveness of our system, we have designed index expression to have at most one parameter n. A type U ranges over primitive (bool, nat, ...) and global types ($\langle G \rangle$), and role types (T) (see Figure 4).

3.2 Ring and Tree Communication Patterns

We illustrate how the formal model of this work can represent various communication patterns such as Ring and Tree.

Ring pattern The global type of the Ring, described in the introduction and Section 2, is defined below. The causality $\mathtt{W}[n{-}j{-}1] \to \mathtt{W}[n-j] : \langle U \rangle$ abstracts the repetitive behavior of the pattern from `0` to `n`, while $\mathtt{W}[n] \to \mathtt{W}[0] : \langle U \rangle$ completes the Ring pattern.

$G ::=$ Global types

$\mathtt{p} \to \mathtt{p}' : \langle U \rangle . G$	Message	$\mid \mathbf{R}\ G\ \lambda i . \lambda \mathbf{x} . G'$	Primitive recursion
$\mathtt{p} \to \mathtt{p}' : \{ l_i : G_i \}_{i \in I}$	Branching	$\mid \mathbf{x}$	Primitive rec. type var.
$\mu \mathbf{t} . G$	Recursion	$\mid G\ \mathtt{t}$	Application
$\mid \mathbf{t}$	Rec. type var.	$\mid$ end	End

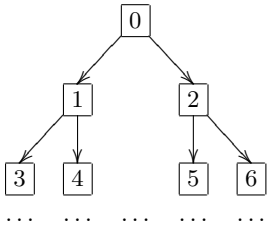
$\mathtt{p} ::= \mathtt{p}[\mathtt{i}] \mid \mathcal{N}$	Principals	$\mathcal{N} ::= \mathtt{Alice} \mid \mathtt{Worker} \mid \ldots$	Participants
$\mathtt{i} ::= \mathtt{t} \mid i \mid \mathrm{n} * \mathtt{i} \mid \mathtt{t} \pm \mathtt{i}$	Index expr.	$U ::= V \mid T$	Message type
$\mathtt{t} ::= \mathrm{n} \mid n \mid \mathtt{t}\ \mathrm{op}\ \mathrm{n} \mid \mathrm{n}^{\mathtt{t}}$	Par. expr.	$V ::= \mathsf{bool} \mid \mathsf{nat} \mid .. \mid \langle G \rangle$	Value type

Fig. 3. Global types

The type specifies that the first message is sent by $\mathtt{W}[0]$ to $\mathtt{W}[1]$ for $j{=}n{-}1$, and the last one is sent by $\mathtt{W}[\mathrm{n}]$ back to $\mathtt{W}[0]$ for $n{=}0$.

$$\begin{array}{l} \mathbf{R}\ \mathtt{W}[n] \to \mathtt{W}[0] : \langle U \rangle . \mathsf{end} \\ \quad \lambda j . \lambda \mathbf{y} . \mathtt{W}[n{-}j{-}1] \to \mathtt{W}[n-j] : \langle U \rangle . \mathbf{y}\ n \end{array}$$

Tree pattern The Tree pattern consists of $2^{n+1}{-}1$ workers organized in a binary tree. The global type below specifies a message exchange between a parent and its children.



$$\begin{array}{l} \mathbf{R}\ \mathsf{end}\ \lambda j . \lambda y . \mathtt{W}[j] \to \mathtt{W}[2{*}j{+}1] : \langle U \rangle . \\ \qquad\qquad \mathtt{W}[j] \to \mathtt{W}[2{*}j{+}2] : \langle U \rangle . y\ (2^n{-}1) \end{array}$$

A tree has three kinds of nodes: root, internal and leaf. The principal running on the root sends a message to its children; the ones on internal nodes send a message to their children and receive a message from their parents; the ones on leaf nodes receive a message from their parents. The three kind of nodes define three distinct roles of the Tree. An internal or leaf node is enumerated by an even or odd number, and thus the mathematical expressions that identify the parent and children of each of these nodes are different. For this reason, even and odd nodes define two distinct roles in the same kind of node, internal/leaf. Thus, we have distinguished five roles in the Tree: *Root* represented by $\mathtt{W}[0]$, *OddIn* and *EvenIn* by $\mathtt{W}[2{*}i{+}1]$ and $\mathtt{W}[2{*}i{+}2]$ ($0{\leq}i{\leq}2^{n-1}{-}2$), and, *OddLeaf* and *EvenLeaf* by $\mathtt{W}[2{*}i{+}1]$ and $\mathtt{W}[2{*}i{+}2]$ ($2^{n-1}{-}1{\leq}i{\leq}2^n{-}2$). To ensure the presence of all five roles in a session, we set $n \geq 2$. We only provide three of the Tree's roles, relegating the other ones and the main program in [5]:

$$\begin{array}{l} \mathit{Root} \triangleq \bar{a}[\mathtt{W}[0], \mathtt{W}[1], \vec{w}](y) . y!\langle \mathtt{W}[1], f(1) \rangle ; y!\langle \mathtt{W}[2], f(2) \rangle ; R' \\ \mathit{OddIn(i)} \triangleq a[\mathtt{W}[2{*}i{+}1]](y) . y!\langle \mathtt{W}[4{*}i{+}3], f(4{*}i{+}3) \rangle ; y!\langle \mathtt{W}[4{*}i{+}4], f(4{*}i{+}4) \rangle ; y?(\mathtt{W}[i], z) ; R \\ \mathit{OddLeaf(i)} \triangleq a[\mathtt{W}[2^n{-}1{+}2{*}i]](y) . y?(\mathtt{W}[2^{n-1}{-}1{+}i], z) ; S \end{array}$$

where f is a function from naturals to U. It is interesting to note that index calculation in the principals of the global type is less complex than in the ones of the roles. This is a direct advantage of the global representation of interactions. The type system of this

$T ::= !\langle \mathtt{p}, U\rangle; T$	Output	$\mid \mathbf{R}\ T\ \lambda i.\lambda \mathbf{x}.T'$	Primitive Recursion
$\mid\ ?\langle \mathtt{p}, U\rangle; T$	Input	$\mid\ T\ \mathtt{t}$	Application
$\mid\ \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle$	Selection	$\mid\ \mathbf{x}$	Primitive Recursion Variable
$\mid\ \&\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle$	Branching	$\mid\ \mathbf{t}$	Recursion Variable
$\mid\ \mu \mathbf{t}.T$	Recursion	$\mid\ \mathsf{end}$	End

Fig. 4. Role types

work statically ensures that the principals in the role's actions are the same to the ones specified in the causalities of global types; e.g. for role *OddIn*, the type system ensures that the first message is sent to $\mathtt{W}[4{*}i{+}3]$. The problem of index calculation in the roles of parallel algorithms has been recognized also by the MPI community [14] as a source of program errors.

4 Type System

This section describes our extension of multiparty's static type system to support parameterised sessions. An essential aspect is the preservation of multiparty's lightweight type annotations and efficient typing strategy of simply global types.

4.1 Projection, Ordering and R-elimination

Projection A global type's projection onto the principals of roles produces types (See Figure 4) that capture the behavior of roles. The given global type is defined on well-formed indexed principals and parametric expressions are applied only to product global types, ensured by the kinding judgment $\Theta; C \vdash G \blacktriangleright \kappa$ (see [5]). A well-formed principal, ensured by the judgment $C \vdash \mathtt{p}$, is either a participant or an indexed principal where the set of values of each index expression is defined over naturals

Definition 4.1 *Given global type G, principal $\mathtt{q}$, and the context C of parameter variables present in G and $\mathtt{q}$, and index variables present in $\mathtt{q}$, if $\emptyset; C \vdash G \blacktriangleright \kappa$ and $C \vdash \mathtt{q}$ then the projection of G onto $\mathtt{q}$, denoted $G\upharpoonright \mathtt{q}$, is defined inductively on G:*

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

$$\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i\in I}\upharpoonright \mathtt{q} = \begin{cases} \oplus\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, \{l_i : \&\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, \{l_i : G_i\upharpoonright \mathtt{q}\}_{i\in I}\rangle(\mathtt{p}')\}_{i\in I}\rangle(\mathtt{p}) & \text{if } C{\vdash}\mathtt{p}{=}\mathtt{q} \text{ and } C{\vdash}\mathtt{p}'{=}\mathtt{q}, \\ \oplus\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, \{l_i : G_i\upharpoonright \mathtt{q}\}_{i\in I}\rangle & \text{if } C{\vdash}\mathtt{p}{=}\mathtt{q}, \\ \&\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, \{l_i : G_i\upharpoonright \mathtt{q}\}_{i\in I}\rangle & \text{if } C{\vdash}\mathtt{p}'{=}\mathtt{q}, \\ \sqcup_{i\in I} G_i\upharpoonright \mathtt{q} & \text{if } C{\nvdash}\mathtt{p}{=}\mathtt{q}, C{\nvdash}\mathtt{p}'{=}\mathtt{q} \\ & \forall i,j{\in}I.G_i\upharpoonright \mathtt{q} \bowtie G_j\upharpoonright \mathtt{q} \end{cases}$$

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

$$\mathbf{R}\ G\ \lambda i.\lambda \mathbf{x}.G'\upharpoonright \mathtt{q} = \mathbf{R}\ (G\upharpoonright \mathtt{q})\ \lambda i.\lambda \mathbf{x}.(G'\upharpoonright \mathtt{q}) \quad \mathbf{x}\upharpoonright \mathtt{q} = \mathbf{x} \quad G\ \mathtt{t}\upharpoonright \mathtt{q} = (G\upharpoonright \mathtt{q})\ \mathtt{t}$$

Projection is intuitive and holds some of the technical challenges of this system, which we discuss in the following paragraphs. In the role types returned, the principal in brackets attached to an action denotes the principal that performs that action, and is used to sort actions and eliminate the $\mathbf{R}$ operator from role types as we shall see later. The equality between a global type principal $\mathtt{p}$ and role principal $\mathtt{q}$ is defined as a relation $\vdash \mathtt{p}{=}\mathtt{q}$ over the context C, which ensures that the set of values of $\mathtt{p}$ is a subset of the set of values of $\mathtt{q}$. For space's sake, we relegate the formal definition of the relation to [5]. The intuition underlying this design originates from the knowledge that an action performed by every process of the same role is captured by the same causality in the global type.

In product global types, an indexed principal can appear in both sides of a parameterised causality for different values of the index variable. This occurrence is covered by the first case of projection for message exchange and branching.

The index variables of principals in global types are different from the ones in roles, as they are bound by different binders. For this reason, we need to translate the role types being expressed from global type indexes to role ones. The $\mathtt{p}'\{\mathtt{p} = \mathtt{q}\}$ operation substitutes the index variables in $\mathtt{p}'$ with expressions in terms of indexes of $\mathtt{q}$, obtained by the relation $\mathtt{p} = \mathtt{q}$ where $\mathtt{p}$ and $\mathtt{p}'$ have the same index variables.

In branching, in the case when $\mathtt{q}$ is not equal neither to $\mathtt{p}$ nor to $\mathtt{p}'$, all inductive projections of $\mathtt{q}$ should return an identical role type up to mergeability $\bowtie$. The notion of mergeability is introduced in [12] as an equivalence relation over role types. Intuitively, two different $\&$ role types are mergeable if denoted by different labels; e.g. the projection of global type $\mathtt{W}[1] \to \mathtt{W}[2] : \begin{cases} \mathit{true} : \mathtt{W}[2] \to \mathtt{W}[3] : \{\mathit{true} : G, \\ \mathit{false} : \mathtt{W}[2] \to \mathtt{W}[3] : \{\mathit{false} : G' \end{cases}$ onto $\mathtt{W}[3]$ returns $\&\langle \mathtt{W}[2], \{\mathit{true}{:}G \upharpoonright \mathtt{W}[3], \mathit{false}{:}G' \upharpoonright \mathtt{W}[3]\}\rangle$, where $G \upharpoonright \mathtt{W}[3] \not\equiv G' \upharpoonright \mathtt{W}[3]$.

Proposition 4.2 *The relation $C \vdash \mathtt{p} = \mathtt{q}$ is decidable.*

Theorem 4.3 *The projection of a global type onto principals is decidable.*

Proof. Straightforward from Proposition 4.2.

Ordering and R-elimination Actions in the role types, returned by projection, are sorted to preserve the order of appearance in all instances of a parameterised global type. We can note from the first case of projection in the message global type, that the order of actions is not preserved; i.e., the sending action is always placed before the receiving one. However, the appearance order of actions is not broken only in the projection of a causality, but also in the sequential composition of other actions returned by projection. The reason behind this is that the order of actions depends on the order of principals performing those actions.

Definition 4.4 *The appearance order relation between two actions (`order`) is defined as the appearance order of the principals performing those actions:*

$$\mathtt{order}(!/?^{2}\langle \mathtt{p}_1, U\rangle(\mathtt{p}_1'), !/?\langle \mathtt{p}_2, U'\rangle(\mathtt{p}_2')) \;\; \textit{iff} \;\; \mathtt{order}(\mathtt{p}_1', \mathtt{p}_2')$$

[2] $!/?$ denotes either ! or ?.

$\mathtt{order}(\oplus/\&\langle \mathtt{p}_1, \{l_i{:}T_i\}_{i\in I}\rangle(\mathtt{p}'_1), \oplus/\&\langle \mathtt{p}_2, \{l_i{:}T'_i\}_{i\in I'}\rangle(\mathtt{p}'_2))$ *iff* $\mathtt{order}(\mathtt{p}'_1, \mathtt{p}'_2)$

Definition 4.5 *The appearance order between principals is defined as a lexicographical order over the index expressions that define them:*

$\mathtt{order}(\mathcal{N}[\mathtt{i}_1]...[\mathtt{i}_i]...[\mathtt{i}_n], \mathcal{N}[\mathtt{i}'_1]...[\mathtt{i}'_i]...[\mathtt{i}'_n])$ *iff* $\mathtt{order}(\mathtt{i}_i, \mathtt{i}'_i)$ *for* $1 \leq i \leq n$ *and* $\forall j.1 \leq j \leq i-1.\ C \vdash \mathtt{i}_j = \mathtt{i}'_j$ *and* $C \nvdash \mathtt{i}_i = \mathtt{i}'_i$,

where the appearance order between index expressions in their canonical form is defined as:

$\mathtt{order}(\mathtt{t}-\mathrm{n}*i, \mathtt{t}'-\mathrm{n}'*i)$ *iff* $C \vdash \mathtt{t}-\mathrm{n}*i \geq \mathtt{t}'-\mathrm{n}'*i$ *and*

$\mathtt{order}(\mathtt{t}+\mathrm{n}*i, \mathtt{t}'+\mathrm{n}'*i)$ *iff* $C \vdash \mathtt{t}+\mathrm{n}*i \leq \mathtt{t}'+\mathrm{n}'*i$.

The order of index expression is defined on the basis that the value of i decreases in each iteration of the $\mathbf{R}$ global type, resulting in the increase of values for expressions $\mathtt{t}-\mathrm{n}*i$ and the decrease for $\mathtt{t}+\mathrm{n}*i$. Thus, in two expressions of the form $\mathtt{t}-\mathrm{n}*i$, a value will appear first in the bigger expression for bigger value of i and then in the smaller one for smaller value of i. And, in two expressions of the form $\mathtt{t}+\mathrm{n}*i$, a value will appear first in the smaller expression for bigger value of i and then in the bigger one for smaller value of i. No ordering can be defined for expressions of opposite monotonicity, e.g. $\mathtt{t}-\mathrm{n}*i$ and $\mathtt{t}'+\mathrm{n}'*i$, as some values will appear first in the former and second in latter, whilst some others vice versa.

The $\mathbf{R}$ operator in global types iterates over parameterised causalities and defines repetitive behavior for non index-parameterised principals. For these principals, we keep the $\mathbf{R}$ operator and the argument applied in the role types, otherwise we eliminate it by composing the two sub-types, and then later the argument. The R-elimination function is denoted by ξ in the typing rules, formally defined in [5].

4.2 Typing Rules

Figure 5 describes the program typing rules. Γ maps shared names, process names and type variables to types, while τ represents channel and product types, defined as:

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

The rules of appealing interest are those for program and session initiation. Rule $\lfloor$TFUN$\rfloor$ augments the context C with mapping for parameter variables and ensures that the subterm is typed. Rule $\lfloor$TAPPF$\rfloor$ checks if the argument applied to the lambda abstraction falls in the set of values $\mathtt{T}$, where $\min(\mathtt{T})$ represents the minimum value n. For primitive recursion, we ensure that the sub-terms are well-typed in the augmented contexts Γ and C. If primitive recursion specifies a repetitive behavior of a role, then $\Delta\ 0$ and $\Delta\ i{+}1$ return the sub-role type for type-checking of the respective sub-terms. Otherwise, $\Delta\ 0$ and $\Delta\ i{+}1$ return Δ. The definition of this rule is similar to that of the previous work, but it does not contain index substitution, simplifying proofs of the properties. The rule of applying a parametric expression to primitive recursion is similar to $\lfloor$TAPPF$\rfloor$, but it also ensures that the argument applied is a successor of the biggest index value. Roles are type-checked by the role types, returned by projection, sorting and $\mathbf{R}$-elimination. The previous work's typechecking algorithm [21] uses the processes types written by programmmers to type-check the processes. The coherence of processes types with respect to global types is proved by an equivalence algorithm

$$\frac{\Gamma; C, n : \mathtt{T} \vdash E \rhd \tau}{\Gamma; C \vdash \lambda n.E \rhd \Pi n{:}\mathtt{T}.\tau} \lfloor \textsc{TFun} \rfloor \qquad \frac{\Gamma; C \vdash E \rhd \Pi n{:}\mathtt{T}.\tau \quad C \vdash \mathtt{t} \geq \mathsf{min}(\mathtt{T})}{\Gamma; C \vdash E\ \mathtt{t} \rhd \tau} \lfloor \textsc{TAppF} \rfloor$$

$$\frac{\Gamma; C \vdash S \rhd \Delta\ 0 \qquad \Gamma, X : \Delta\ i; C, i : \mathtt{I} \vdash R \rhd \Delta\ i+1}{\Gamma; C \vdash \mathbf{R}\ S\ \lambda i.\lambda X.R \rhd \Pi i{:}\mathtt{I}.\Delta} \lfloor \textsc{TPRec} \rfloor \qquad \frac{C \vdash \mathtt{t} \quad \Gamma; C \vdash R \rhd \quad \Pi i{:}\{i \mid 0 \leq i \leq \mathtt{t}-1\}.\Delta}{\Gamma; C \vdash R\ \mathtt{t} \rhd \Delta\ \mathtt{t}} \lfloor \textsc{TAppR} \rfloor$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \emptyset; C \vdash G \blacktriangleright \mathsf{Type} \quad C \vdash \mathtt{p}_0, \mathtt{p}_1, p \quad C \vdash \mathsf{pid}(G) = \{\mathtt{p}_0, \mathtt{p}_1, p\} \quad \Gamma; C \vdash R \rhd \Delta, y : \xi(G \upharpoonright \mathtt{p}_0)}{\Gamma; C \vdash \bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R \rhd \Delta} \lfloor \textsc{TAcc} \rfloor$$

$$\frac{\emptyset; C \vdash G \blacktriangleright \mathsf{Type} \quad \Gamma \vdash u : \langle G \rangle \quad C \vdash \mathtt{p} \quad \Gamma; C \vdash R \rhd \Delta, y : \xi(G \upharpoonright \mathtt{p})}{\Gamma; C \vdash u[\mathtt{p}](y).R \rhd \Delta} \lfloor \textsc{TReq} \rfloor$$

$$\frac{\Gamma; C \vdash e \rhd S \quad \Gamma \vdash R \rhd \Delta, c : T}{\Gamma; C \vdash c!\langle \mathtt{p}, e \rangle; R \rhd \Delta, c :!\langle \mathtt{p}, S \rangle; T} \lfloor \textsc{TOut} \rfloor \qquad \frac{\Gamma, x : S; C \vdash R \rhd \Delta, c : T}{\Gamma; C \vdash c?\langle \mathtt{p}, x \rangle; R \rhd \Delta, c :?\langle \mathtt{p}, S \rangle; T} \lfloor \textsc{TIn} \rfloor$$

Fig. 5. Program and role typing

that uses the sets of parameters' values to produce instances of product global types and processes types. Then, the instances of processes type are checked if they are the same to the one returned from projection of the global type instance, using multiparty's projection algorithm [16]. All the conditions to invoke projection are ensured by $\lfloor$TAcc$\rfloor$ and $\lfloor$TReq$\rfloor$. Rule $\lfloor$TAcc$\rfloor$ checks also that the set of principals, present in the session, is the same to the one of global type. The equality relation between the two sets of principals is defined over the mathematical definition of parameterised list of principals discussed in Section 2, missing from the previous work.

Rules $\lfloor$TOut$\rfloor$ and $\lfloor$TIn$\rfloor$ ensure that the sub-terms are typed and check if the principal in the primitives is the same as the one in the role-types. Other standard rules lookup for type variables in Γ, and type branching, hiding, inaction and parallel composition.

Properties. In this paragraph, we state type preservation for the formal system presented in this paper. The full proof is given in a companion technical report [5].

Theorem 4.6 (Type Preservation) *If $\Gamma; C \vdash E \rhd \tau$, and $E \rightarrow E'$, then there exists τ' where $\tau \Rightarrow \tau'$, such that $\Gamma; C \vdash E' \rhd \tau'$.*

Proof. By induction over the derivation of $E \rightarrow E'$. The proof relies on standard substitution lemmas. Reduction of types $\tau \Rightarrow \tau'$ reflects reduction of processes in presence of application, and sending and receiving of values/labels.

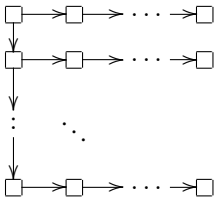
Although, we do not have a formal proof, we believe the system satisfies the standard progress property. Indeed, our system benefits the proof of progress for well-typed processes willing to start a session from Bettini *et al.* [7]. To complete the proof, we need to ascertain that a well-typed program reduces to the above processes and that a well-typed iterative behavior reduces further. We leave the proof of progress for future work. The previous system provides progress for the formal model.

5 Real-World Examples

Jacobi Solution of the Discrete Poisson Equation [14] Poisson's equation is widely used in many areas of the natural sciences. Jacobi's method converges on a solution by repeatedly replacing each element of the input grid by an adjusted average of its four neighbouring values. The grid can be divided up and the algorithm is performed on each subgrid in separate processes. Neighbouring processes must exchange their subgrid boundary values (ghost-points) as they are updated. We illustrate a two-dimensional (mesh) decomposition of the grid into $n{*}m$ processes, where $n, m{\geq}2$. The process on the (n, m) subgrid, top right corner, controls the termination condition for all processes and sends the first message in the mesh. The global type for the said interactions is:

$$\mathit{Jacobi} \triangleq \mu\mathbf{t}.\mathtt{W}[n][m] \to \mathtt{W}[n][m-1], \mathtt{W}[n-1][m] : \{\mathtt{true} : \mathit{iterate}, \mathtt{false} : \mathit{return}\}.$$

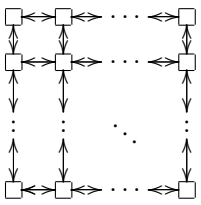
The stopping condition is propagated in the processes following the pattern of the diagram below. Next to it, the global type (*iterate*) for propagating the `true` label.



1. $\mathbf{R}$
2. $\mathbf{R}\ \ (\mathbf{R}\ \mathit{ghst–pnt}\ \lambda k{:}\mathtt{I}'.\lambda\mathbf{z}.\mathtt{W}[0][k{+}1] \to \mathtt{W}[0][k]{:}\{\mathtt{true}{:}\mathbf{z}\}\ m)$
3. $\quad\lambda i : \mathtt{I}.\lambda\mathbf{x}.$
4. $\quad\quad(\mathbf{R}\ \mathtt{W}[i{+}1][m] \to \mathtt{W}[i][m]{:}\{\mathtt{true}{:}\mathbf{x}\}$
5. $\quad\quad\quad\lambda j{:}\mathtt{I}'.\lambda\mathbf{y}.\mathtt{W}[i{+}1][j{+}1] \to \mathtt{W}[i{+}1][j]{:}\{\mathtt{true}{:}\mathbf{y}\}$
6. $\quad\quad\quad m)$
7. $\quad n{-}1$
8. $\lambda l{:}\mathtt{I}.\lambda\mathbf{w}.\mathtt{W}[n][l{+}1] \to \mathtt{W}[n][l]{:}\{\mathtt{true}{:}\mathbf{w}\}\ (m{-}1)$

Propagation of the label in the top row, is described in the causality of line 8, in all the rows, except top and bottom, line 5, in the leftmost column in line 4 and in the bottom row, line 2.

Each process maintains a copy of the boundary values of its neighbours and exchanges them on each iteration of the algorithm. The diagram below portrays how these values are exchanged between the processes, followed by the global type (*ghst–pnt*).



1. $\mathbf{R}\ \ (\mathbf{R}\ \mathit{conv–data}\ \lambda k{:}\mathtt{I}'.\lambda\mathbf{z}.\mathtt{W}[0][k+1] \leftrightarrow \mathtt{W}[0][k]{:}\langle U\rangle\ m)$
2. $\quad\lambda i{:}\mathtt{I}.\lambda\mathbf{x}.$
3. $\quad\quad(\mathbf{R}\ \mathtt{W}[i+1][0] \leftrightarrow \mathtt{W}[i][0]{:}\langle U\rangle.\mathbf{x}$
4. $\quad\quad\quad\lambda j{:}\mathtt{I}'.\lambda\mathbf{y}.\mathtt{W}[i+1][j+1] \leftrightarrow \mathtt{W}[i][j+1]{:}\langle U\rangle.$
5. $\quad\quad\quad\quad\quad\quad\mathtt{W}[i+1][j+1] \leftrightarrow \mathtt{W}[i+1][j]{:}\langle U\rangle.\mathbf{y}$
6. $\quad\quad\quad m)$
7. $\quad n$

$\mathtt{p} \leftrightarrow \mathtt{p}'{:}\langle U\rangle$ is a shortcut for $\mathtt{p} \to \mathtt{p}'{:}\langle U\rangle.\mathtt{p}' \to \mathtt{p}{:}\langle U\rangle$. The exchange of ghost-points in all the rows and columns, except the leftmost column line 3 and bottom row line 1, is described in the causalities of line 4 and 5. The full definition of the global type and roles is given in [5].

Group Diffie-Hellman with Complete Key Authentication Protocol [4] The Diffie-Hellman protocol is used in password-authentication key agreement and public key infrastructure. Every group M_i $(0{<}i{<}n)$ generates and encrypts a random exponent, that together with the data received from M_{i-1} is then sent to M_{i+1}. Lastly, M_n receives

the data from M_{n-1}, computes the group key and broadcasts it to all other parties. The global type of the protocol is defined as:

$\mathbf{R}\ (\mathbf{R}\ \mathsf{end}\ \lambda k : \mathtt{I}.\lambda \mathbf{z}.\mathtt{M}[n] \to \mathtt{M}[k] : \langle key \rangle.\mathbf{z}\ n)$ //Broadcasting the group key
$\lambda i : \mathtt{I}.\lambda \mathbf{x}.\mathtt{M}[n-i-1] \to \mathtt{M}[n-i] : \langle data \rangle.\mathbf{x}\ n$ //Exchanging data on a line pattern

This protocol is modelled in a system of contracts [13]. In that model, an extra private channel is used by the last group M_n to send the key to every other group. The private channel is forwarded between the groups through delegation. A condition is added to the protocol description to check whether the key is sent to every group or not.
In our model, we do not need an extra private channel to send the key, as communications between parties of a session are always defined over private channels. Also, we do not need to add a condition that checks whether the key is sent to every group as this is granted by the semantics of the $\mathbf{R}$ operator.

6 Related Work

The idea of parameterised session types originated from our previous research on investigating the expressivity of session types for parallel algorithms [6]. This idea has been modelled recently in our work [21], which differences with this system were discussed in the introduction and throughout the paper.

Our formal system is modelled after Bettini *et al.* [7], a simpler version of Honda *et al.* [16], which difference was discussed at the beginning of Section 2; none of these systems are expressive enough to model parameterised communication patterns. The $\mathbf{R}$ operator used in the initial and present work was introduced by Gödel in System *T* [1]. The idea of using the $\mathbf{R}$ operator comes from Nelson's work on adding primitive recursion to the lambda calculus [17]. As a result, his system can type functions previously untyped in ML. Our use of the $\mathbf{R}$ operator models parameterised sessions.

Session types were first introduced by Honda *et al.* [15, 19] to capture the interaction structure of two processes. Their type system checks whether for each "send" on one process corresponds a "receive" on the other and vice versa. Honda *et al.* [16] extended their system from two-parties to n-parties. Using an intuitive syntax, they introduced a notion of global type to describe the interaction structure of n processes from a global viewpoint. Multiparty session types have been studied also by Bonelli and Compagnoni [8]. Their type system is defined over binary session types, obtained by projecting processes local types onto principals. Session types have been used to type service-oriented multiparty communications [9]. The calculus proposed permits communications inside and outside a session to model merging of two running sessions. Type safety and progress properties are not provided for the formal model.

Contracts [13] are another typing model of mobile processes, defined over processes as behavioral types and not over channels as session types [7]. Consequently, they can well-type more correct programs than session types. However, the expressiveness of the type system comes at a practical cost. Contracts have no intuitive syntax as global types and no iterative construct as in our system. Thus, they do not provide a practical model to design a programming language that supports communication and elegantly expresses parameterised communication patterns; e.g. the key exchange protocol (Sec. 5) has been augmented with additional interactions to check the end of a send-iteration.

The conversation calculus [11] is based on boxed ambients [10] and not in the π-calculus as session types are. Typing is similar to the one of contracts and thus the system carries the same disadvantages when compared to session types. The calculus models dynamic joining and leaving of participants within a session.

Behaviors [2, 18] describe the communication behaviour that captures the causal constrains of a concurrent program through terms of a process algebra, similarly to contracts. An implementation [3] of their system for deriving behaviors is provided for CML programs, where their notion of communication pattern, expressed using behaviors, is similar to our notion of role type. Behaviors have a similar typing discipline as contracts, thus they lack the same features of session types as contracts do.

7 Conclusions and Future Work

This paper presented a practical design of parameterised session types. The idiom of roles has the same design as classes in class-based languages, offering a practical concept on how to incorporate parameterised session types into mainstream languages such as Java and C#. This contrasts with the amorphous design of the previous work of parameterised session types. The static type system here follows the efficient typing strategy and programming methodology of multiparty session types: programmers first define the global type of the intended pattern and then define each role of it; roles are then validated through projection of the global type onto the principals by type-checking. This contrasts with the heavyweight type annotated programs in the previous work, where programmers write the processes types, in addition to the global type, for type-checking. Also, the coherence of the processes types with respect to the global type is provided through a type equivalence relation. The system here allows values of parameters to range over infinite sets of naturals to provide full computation power of programs that implement parameterised communication patterns. This contrasts with the conservative system of finite sets used by the previous work. We presented a series of examples illustrating the practical utility and effectiveness of this system, including the control of index calculation in roles, one of the main source of errors in MPI.

The next step in developing this work is the implementation of the model as a library of a mainstream language, where session channels can be implemented as communicators in MPI (e.g. MPI_COMM_WORLD)—communication abstraction for a group of processes. We believe that a communication-safe library of parameterised sessions would increase productivity in parallel algorithms, web-services and other distributed applications. From a theoretical perspective, there are several ways to extend the current system. The index expressions in principals can be extended to richer mathematical operations such as congruence and a more expressive form of exponentiation, preserving the decidability of the type system. More dynamic concepts such as joining and leaving of participants within a session are of interest in web-services and cloud management systems design.

Acknowledgements I thank Nobuko Yoshida for helpful technical discussions on this material, and Dave Clarke, Iain Phillips, Raymond Hu, Andrew Farell and the anonymous reviewers of Coordination and ICFEM for comments on an earlier version of this paper.

References

1. S. Alves, M. Fernandez, M. Florido, and I. Mackie. Gödel System T revisited. In *Theoretical Computer Science*, volume 411(11-13), pages 1484–1500. Elsevier, 2010.
2. T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *J. Funct. Program.*, 7(3):321–347, 1997.
3. T. Amtoft, H. R. Nielson, and F. Nielson. Behaviour analysis for validating communication patterns. *Software Tools for Technology Transfer*, 2(1):13–28, 1998.
4. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *CCS*, pages 17–26. ACM, 1998.
5. A. Bejleri. Practical Parameterised Session Types. Tech. Report DTR10-7, Imperial College, Available at `http://www.doc.ic.ac.uk/~ab406/papers/param.pdf`, 2010.
6. A. Bejleri, R. Hu, and N. Yoshida. Session-based programming for parallel algorithms: Expressiveness and performance. In *PLACES'09*, volume 17 of *EPTCS*, pages 17–29, 2010. `http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html`.
7. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
8. E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, 2008.
9. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.
10. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Trans. Program. Lang. Syst.*, 26(1):57–124, 2004.
11. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
12. M. Carbone, N. Yoshida, and K. Honda. Asynchronous session types: Exceptions and multiparty interactions. In *SFM'09*, volume 5569 of *LNCS*, pages 187–212. Springer, 2009.
13. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.
14. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.
15. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
16. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
17. N. Nelson. Primitive recursive functionals with dependent types. In *MFPS*, volume 598 of *LNCS*, pages 125–143, 1991.
18. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 84–97, New York, NY, USA, 1994. ACM.
19. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
20. Web Services Choreography Working Group. Web Services Choreography Description Lang. http://www.w3.org/TR/ws-cdl-10-primer/.
21. N. Yoshida, P.-M. Denielou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FOSSACS*, volume 6014 of *LNCS*, pages 128–145, 2010.