Castro-Perez, David, Ferreira, Francisco, Gheri, Lorenzo and Yoshida, Nobuko (2021) *Zooid: A DSL for Certified Multiparty Computation.* In: PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. . ACM ISBN 978-1-4503-8391-2.

Zooid: A DSL for Certified Multiparty Computation

From Mechanised Metatheory to Certified Multiparty Processes

David Castro-Perez
Imperial College London and University of Kent, UK

Francisco Ferreira
Imperial College London, UK

Lorenzo Gheri
Imperial College London, UK

Nobuko Yoshida
Imperial College London, UK

Abstract

We design and implement Zooid, a domain specific language for certified multiparty communication, embedded in Coq and implemented atop our mechanisation framework of asynchronous multiparty session types (the first of its kind). Zooid provides a fully mechanised metatheory for the semantics of global and local types, and a fully verified end-point process language that faithfully reflects the type-level behaviours and thus inherits the global types properties such as deadlock freedom, protocol compliance, and liveness guarantees.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Theory of computation** → *Type theory*; *Program semantics*; **Process calculi**.

Keywords: multiparty session types, mechanisation, Coq, concurrent processes, protocol compliance, deadlock freedom, liveness

ACM Reference Format:
David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454041

1 Introduction

Concurrent behavioural type systems [31] accurately simulate and abstract the behaviour of interactive *processes*, as opposed to sequential types for programs that simply describe values. The *session types system* [24, 26, 44] is one of such behavioural type systems, which can determine protocol compliance for processes. Session types consist of actions for sending and receiving, sequencing, choices, and recursion. In session types, when a typed process communicates, its type also evolves, thus reflecting the progression of the state of the protocol (type) after performing an action. This rich behavioural aspect of session types has opened new areas of study, such as a connection with communicating automata [5] and concurrent game semantics [38] by linking actions of session types to transitions of state machines [15] and events of games [7].

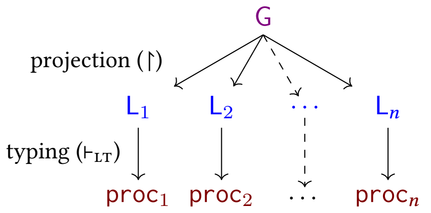


Figure 1. MPST in a nutshell

Originally, binary session types (BST) provide deadlock-freedom for a pair of processes, but not when more than two *participants* (often also called *roles*) are involved. For more than two processes, ensuring deadlock-freedom in BST requires either complicated additional causality-based typing systems on top of plain BST, e.g. [1, 17] or limitation to deterministic, strongly-normalising session types [48, 49].

Multiparty session types (MPST, [27, 28]) solve this limitation, by defining *global types* as an overall specification of all the communications by every participant involved. The essence of the MPST theory (depicted in Figure 1) is *end-point projection* where a global type G is projected into one *local type* L_i for each participant, so that the participant $\mathtt{proc}_i$ can be implemented following an abstract behaviour represented by the local type. To ensure correctness, the collection of behaviours of the local types projected from a global type need to mirror the behaviour of that global type.

The behaviour of global and local types is defined by (asynchronous) labelled transition systems (LTS) whose sound and complete correspondence is key to provide: progress of processes [28], synthesis of global protocols [16, 33], and to establish bisimulation for processes [32]. Practically, type-level transition systems are particularly useful for, e.g., dynamic monitoring of components in distributed systems [14]

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454041

and generating deadlock-free APIs of various programming languages, e.g., [10, 29, 34, 39, 53].

Unfortunately, the more complicated the behaviour is, the more *error-prone* the theory becomes. The literature reveals broken proofs of subject reduction for several MPST systems [40], and a flaw of the decidability of subtyping [6] for asynchronous MPST. All of which are caused by an incorrect understanding of the (asynchronous) behaviour of types.

Motivated by this experience, we design and implement Zooid[1], a certified Domain Specific Language (DSL) to write *well-typed by construction* communicating processes. Zooid's implementation is embedded in the Coq proof assistant [46], so that it relies on solid and precise foundations: in Coq we have formalised the metatheory for MPST, which serves as the type system for Zooid. On one side, mechanising the metatheory is immediately useful for documenting, clarifying, and ensuring the validity of proofs, on the other it results in certified specifications and implementations of the concepts in the theory. Zooid exemplifies this for MPST, a complex and relevant theory with many real-world applications. In this system, not only the theory is validated in Coq, the actual implementation of projection, type checking and validation of processes, is extracted from certified proofs.

We provide the first *fully mechanised* proof of sound and complete correspondence between the labelled transition systems of global and local types, in terms of equivalence of execution traces, recapturing the original LTS provided in [16]. In this work, instead of trying to formalise existing proofs in the literature, we approach the problem with a fresh look and use tools that would allow for a successful and reusable mechanisation. On the theory side, we use coinductive trees inspired by [20, 51]; on the tool side, we depend on the Coq proof assistant [46], taking advantage of small scale reflection (SSReflect) [21] to structure our proofs, and PaCo [30] to provide a powerful parameterised coinduction library, which we use extensively.

To certify an MPST end-point process implementation, we define a concurrent process language and an LTS semantics for it. This guarantees that process traces respect the ones from its local and global types. Naturally, processes do not need to implement every aspect of the protocol. Therefore, we define the notion of complete subtraces to represent the fact that an implementation may choose not to implement some aspects, but it still needs to match the global trace (we make precise this concept in § 4.3). Our final result is the design and implementation of Zooid, a Coq-embedded DSL to write end-point processes that are well-typed (hence deadlock-free and live) *by construction*. This development takes full advantage of the metatheory to provide a certified validation, projection, and type checking for Zooid processes.

The contributions of this work are fourfold:

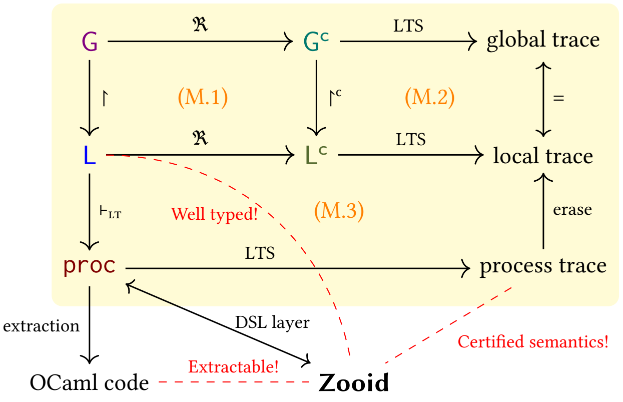


Figure 2. Our contribution at a glance.

Fully mechanised transition systems for global and local types, using *asynchronous communications* and proofs of their sound and complete trace equivalence.

Semantic representation of behavioural types based on *coinductive trees*, proposing a novel approach to the proof of trace equivalences.

A concurrent process language with an associated typing discipline and the notion of *complete subtraces* to relate process traces to global traces, as processes may not fully implement a protocol and still be compliant.

Zooid a DSL embedded in Coq and framework that *specifies* global protocols, performs *projections*, and implements intrinsically well-typed processes, using code *certified* by Coq proofs. The code of Zooid processes is *extracted into OCaml* code for execution. Zooid uses the mechanisation to provide a framework for processes that enjoy deadlock freedom and liveness (with a type checker certified in Coq).

Outline. In § 2, we provide an overview of the theory and the paper. In § 3, we present the theory of MPST together with the soundness and completeness results. We describe the process language, its metatheory and the Zooid DSL in § 4. In § 5, we present Zooid's workflow and showcase its use with some examples. In § 6 we discuss related work and offer some future work and conclusions.

The git repository of our development is publicly available: https://github.com/emtst/zooid-cmpst; it contains all the complete Coq definitions and proofs from the paper, together with the examples and case studies implemented using Zooid. Additionally, the artifact associated to this paper is available at [11]. We present the proofs of our theorems, and additional technical details of the toolchain, in the appendix of the full version of the paper (https://arxiv.org/pdf/2103.10269).

2 Overview

In this section, we present our formalised results and the relationship that puts them together to build Zooid; and we show, with an example, how our development allows to *certify* the implementation of a multiparty protocol.

[1] A zooid is a single animal that is part of a colonial animal, akin to how an endpoint process is part of a distributed system.

2.1 Results and Development

Figure 2 summarises our contribution. The yellow rectangle on the background encases the metatheory that we have formalised for types and processes. On such solid basis, we build Zooid, our language for specifying end-point processes.

Types as Trees, Projection and Unravelling. We formalise in Coq the inductive syntaxes of global types and local types. Of these, we give an alternative representation in terms of coinductive trees, moving one step forward towards semantics. By defining the unravelling relation $\mathfrak{R}$, of a type into a tree (§ 3.1), and projections $\upharpoonright$, from global to local objects (§ 3.2), we prove Theorem 3.6: projection is preserved by unravelling (square (M.1) in Figure 2).

Trace Semantics. Moving further to the right, we define labelled transition systems for trees (§ 3.3 and 3.4). Exploiting their tree representation, we give an asynchronous semantics in terms of execution traces to global and local types (§ 3.5). *Soundness and completeness* come together in the *trace equivalence theorem* for global and local types, Theorem 3.21, thus closing square (M.2) in Figure 2.

Process Language and Typing. We formalise the syntax for specifying (core) processes, proc in Figure 2 (§ 4.1). We define a typing relation between local types and processes, then we give semantics to processes (§ 4.3), again in terms of an LTS and execution traces, and finally we prove type preservation, Theorem 4.5. We conclude the metatheory part with Theorem 4.7, (thus closing square (M.3) of Figure 2): we show that *process traces are global traces.*

2.2 Process Language: Zooid

On the foundations of a formalised metatheory, we build a domain specific language embedded in Coq, Zooid, as presented in § 4 and 5. Processes specified in Zooid are well-typed by construction. Zooid terms are dependent pairs of a core process proc, and a proof that it is well-typed with respect to a given local type L, obtained via projection of the global type G given for the protocol. Zooid terms are built using a collection of *smart constructors*: we make sure that the local type of any smart constructor is fully determined by its inputs, so that we can use Coq to infer the local type for every Zooid process.

To summarise, our end product Zooid is a DSL embedded in Coq. The user specifies as inputs:

1. the general discipline of the protocol as a global type;
2. the communicating process they are interested in, as a Zooid term.

From this the user will obtain:

(a) a collection of local types inferred by projection from the given global type;
(b) that their process is well-typed by construction;
(c) a certified semantics for their process, namely the guarantee that the behaviour of their process adheres to the semantics of the global protocol.

Moreover the user's process is easily translated to an OCaml program, thanks to Coq code-extraction.

2.3 Zooid at Work

We briefly illustrate how Zooid works with a simple example, a ring protocol. We want to write a certified process for Alice that sends a message to Bob and then receives a message from Carol, but only after Bob and Carol have exchanged a message themselves. In what follows, all the considered messages are natural numbers of type nat.

First, we provide Zooid with the intended disciplining protocol, a global type G:

$$\mathsf{G} = \mathsf{Alice} \to \mathsf{Bob} : \ell(\mathtt{nat}).\ \mathsf{Bob} \to \mathsf{Carol} : \ell(\mathtt{nat}).\ \mathsf{Carol} \to \mathsf{Alice} : \ell(\mathtt{nat}).\mathtt{end}$$

The global type G prescribes the full protocol, where Alice sends a message containing a nat number to Bob (with a generic label ℓ), Bob receives it and sends another number to Carol, who receives and can send the last message to Alice. Alice receives and the protocol terminates (end).

Taking the point of view of Alice, we automatically obtain a local type L, *projection* of G onto the role Alice:

$$\mathsf{L} = \mathtt{!}[\mathsf{Bob}];\ell(\mathtt{nat}).\mathtt{?}[\mathsf{Carol}];\ell(\mathtt{nat}).\mathtt{end},$$

which prescribes for Alice that she will send a number to Bob, receive a number from Carol and terminate.

A Zooid implementation for Alice's process, respecting L, is (Alice sends x to Bob and gets y from Carol):

$$\mathsf{proc} = \mathsf{send}\ \mathsf{Bob}\ (\ell,\ x : \mathtt{nat})!\ \mathsf{recv}\ \mathsf{Carol}\ (\ell,\ y : \mathtt{nat})?\ \mathtt{finish}$$

Thanks to Zooid's smart constructors, we obtain that proc is *well-typed* with respect to the local type L. Additionally, the underlying metatheory certifies, by Coq proofs, that the behaviour of proc conforms to the semantics of protocol G.

3 Sound and Complete Asynchronous Multiparty Session Types

In this section, we describe the first layer of Zooid's certified development: a mechanisation of the metatheory of multiparty session types. We focus on the design, main concepts and results, while for a more in-detail presentation with pointers to the Coq mechanisation, we refer to [12].

3.1 Global and Local Types

A *global type* describes the communication protocol in its entirety, recording all the interactions between the different participants. Each participant has a *local type* specifying its intended behaviour within the protocol. The literature offers a wide variety of presentations of global and local types [13, 27, 28, 41]: here, building on [16], we formalise full *asynchronous multiparty session types* (MPST), which captures asynchronous communication, with choice and recursion.

Definition 3.1 (Sorts, global and local types)**.** *Sorts* (`mty` in `Common/AtomSets.v`), *global types* (`g_ty` in `Global/Syntax.v`), and *Local types* (`l_ty` in `Local/Syntax.v`), ranged over by S, G, and L respectively, are generated by:

$$\mathsf{S} ::= \mathtt{nat} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathsf{S}{+}\mathsf{S} \mid \mathsf{S}{*}\mathsf{S}$$

$$\mathsf{G} ::= \mathtt{end} \mid X \mid \mu X.\mathsf{G} \mid \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}$$

$$\mathsf{L} ::= \mathtt{end} \mid X \mid \mu X.\mathsf{L} \mid ![\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I} \mid ?[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I}$$

with $\mathsf{p} \neq \mathsf{q}$, $I \neq \emptyset$, and $\ell_i \neq \ell_j$ when $i \neq j$, for all $i, j \in I$.

Above, *sorts* refer to the types of supported message payloads. We are interested in types such that (1) bound variables are *guarded*—e.g., $\mu X.\mathsf{p} \to \mathsf{q} : \ell(\mathtt{nat}).\mathsf{G}$ is a valid global type, whereas $\mu X.X$ is not—and (2) types are *closed*, i.e., all variables are bound by μX ([12]).

In the literature, it is common to adopt the *equi-recursive viewpoint* [37], i.e., to identify $\mu X.\mathsf{G}$ and $\mathsf{G}\{\mu X.\mathsf{G}/X\}$, given that their intended behaviour is the same. Such unravelling of recursion can be performed infinitely many times, thus obtaining possibly infinite trees[2], whose structure derives from the syntax of global and local types [20].

Definition 3.2 (Semantic global and local trees)**.** *Semantic global trees* (`rg_ty` and `ig_ty` in `Global/Tree.v`, see also [12]), ranged over by $\mathsf{G^c}$, and *semantic local trees* (`rl_ty` in `Local/Tree.v`), ranged over by L, are generated *coinductively* by:

$$\mathsf{G^c} ::= \mathtt{end^c} \mid \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \mid \mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}$$

$$\mathsf{L^c} ::= \mathtt{end^c} \mid !^{\mathsf{c}}[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I} \mid ?^{\mathsf{c}}[\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I}$$

with $\mathsf{p} \neq \mathsf{q}$, $I \neq \emptyset$, and $\ell_i \neq \ell_j$ when $i \neq j$, for all $i, j \in I$.

Global and local objects share the type for a terminated protocol $\mathtt{end}$, the injection of a variable X, and the recursion construct $\mu X.\ldots$; semantic global/local trees do not include the last two constructs, since recursion is captured by infinite depth (see [12]). **Global messages:** $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}$ describes a protocol where participant p sends to q one message with label ℓ_i and a value of sort S_i as payload, for some $i \in I$; then, depending on which ℓ_i was sent by p, the protocol continues as G_i. With trees, we make explicit the two asynchronous stages of the communication of a message: $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}$ represents the status where a message from p to q has yet to be sent; $\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}$ represents the next status: the label ℓ_j has been selected, p has sent the message, with payload S_j, but q has not received it yet. **Local messages:** *send type* $![\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I}$: the participant sends a message to q; if the participant chooses the label ℓ_i, then the sent payload value must be of sort S_i, and it continues as prescribed by L_i. *Receive type* $?[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I}$: the participant waits to receive from p a value of sort S_i, for some $i \in I$, via a message with label ℓ_i; then the protocol continues as prescribed by L_i. The same intuition holds, mutatis mutandis, for trees.

We define the function `prts` to return the set of participants (or *roles*) of a global type; e.g. p and q above. For global trees, we define the predicate `part_of`. The formal definitions can be found in [12].

We formalise equi-recursion by relating types with their representation as trees, as follows:

Definition 3.3 (Unravelling)**.** *Unravelling of global types types* (`GUnroll` in `Global/Unravel.v`) and *unravelling of local types* (`LUnroll` in `Local/Unravel.v`) are the relations between global/local types and semantic global/local trees coinductively defined by:

$$\text{[G-UNR-END]}\ \frac{}{\mathtt{end}\ \mathfrak{R}\ \mathtt{end^c}} \qquad \text{[G-UNR-REC]}\ \frac{\mathsf{G}\{\mu X.\mathsf{G}/X\}\ \mathfrak{R}\ \mathsf{G^c}}{\mu X.\mathsf{G}\ \mathfrak{R}\ \mathsf{G^c}}$$

$$\text{[G-UNR-MSG]}\ \frac{\forall i \in I.\mathsf{G}_i\ \mathfrak{R}\ \mathsf{G^c}_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}\ \mathfrak{R}\ \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}}$$

$$\text{[L-UNR-END]}\ \frac{}{\mathtt{end}\ \mathfrak{R}\ \mathtt{end^c}} \qquad \text{[L-UNR-SEND]}\ \frac{\forall i \in I.\mathsf{L}_i\ \mathfrak{R}\ \mathsf{L^c}_i}{![\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I}\ \mathfrak{R}\ !^{\mathsf{c}}[\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I}}$$

$$\text{[L-UNR-REC]}\ \frac{\mathsf{L}\{\mu X.\mathsf{L}/X\}\ \mathfrak{R}\ \mathsf{L^c}}{\mu X.\mathsf{L}\ \mathfrak{R}\ \mathsf{L^c}} \qquad \text{[L-UNR-RECV]}\ \frac{\forall i \in I.\mathsf{L}_i\ \mathfrak{R}\ \mathsf{L^c}_i}{?[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i\in I}\ \mathfrak{R}\ ?^{\mathsf{c}}[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I}}$$

Representing types in terms of trees allows for a smoother mechanisation of the semantics. The unravelling operation formally relates the two representations.

3.2 Projections, or How to Discipline Communication

Projection is the key operation of multiparty session types: it extracts a local perspective of the protocol, from the point of view of a single participant, from the global bird's-eye perspective offered by global types. We define both inductive and coinductive projections.

Definition 3.4. The inductive *projection of a global type onto a participant* r (`project` in `Projection/IProject.v`) is a partial function $\_\upharpoonright \mathsf{r} : \mathtt{g\_ty} \rightharpoonup \mathtt{l\_ty}$ defined by recursion on G whenever one of the clauses in Figure 3a applies and the recursive call is defined; the coinductive *projection of a global tree onto a participant* r (`Project` and `IProj` in `Projection/CProject.v`) is a relation $\_ \upharpoonright^{\mathsf{c}} \mathsf{r}\ \_ : \mathtt{rel}\ \mathtt{g\_ty^c}\ \mathtt{l\_ty^c}$ coinductively defined in Figure 3b.

In rules [CO-PROJ-END] and [CO-PROJ-CONT] we have added explicit conditions on participants. By factoring in the predicate `part_of`, Definition 3.4 ensures (1) that the projection of a global tree on a participant outside the protocol is $\mathtt{end^c}$ (rule [CO-PROJ-END]) and (2) that this discipline is preserved in the continuations (rule [CO-PROJ-CONT]). We see that the clauses for projecting of types and trees follow the same intuition: projecting a global object onto a sending (resp. receiving) role gives a sending (resp. receiving) local object, provided that the local continuations are also projections of the corresponding global continuations. As expected, the tree projection takes care explicitly of asynchronicity (rules [CO-PROJ-SEND-2]

[2] Formally, in Coq, a coinductively defined datatype (codatatype) of finitely branching trees with possible infinite depth.

[PROJ-END]
$\mathsf{end} \upharpoonright \mathsf{r} = \mathsf{end}$

[PROJ-SEND]
$\mathsf{r} = \mathsf{p}$ implies $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i \in I} \upharpoonright \mathsf{r} = ![\mathsf{q}];\ \{\ell_i(\mathsf{S}_i).\mathsf{G}_i \upharpoonright \mathsf{r}\}_{i \in I}$

[PROJ-RECV]
$\mathsf{r} = \mathsf{q}$ implies $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i \in I} \upharpoonright \mathsf{r} = ?[\mathsf{p}];\ \{\ell_i(\mathsf{S}_i).\mathsf{G}_i \upharpoonright \mathsf{r}\}_{i \in I}$

[PROJ-VAR]
$X \upharpoonright \mathsf{r} = X$

[PROJ-REC]
$(\mu X.\mathsf{G}) \upharpoonright \mathsf{r} = \mu X.(\mathsf{G} \upharpoonright \mathsf{r})$ if $\mathsf{guarded}(\mathsf{G} \upharpoonright \mathsf{r})$

[PROJ-CONT]
$\mathsf{r} \neq \mathsf{p}, \mathsf{r} \neq \mathsf{q}$ and $\forall i, j \in I, \mathsf{G}_i \upharpoonright \mathsf{r} = \mathsf{G}_j \upharpoonright \mathsf{r}$ implies $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i \in I} \upharpoonright \mathsf{r} = \mathsf{G}_{\bar{\imath}} \upharpoonright \mathsf{r}$ (with $\bar{\imath} \in I$)

(a) Rules for recursive projection, Definition 3.4

[CO-PROJ-SEND-1]
$$\frac{\mathsf{r} = \mathsf{p} \qquad \forall i \in I.\mathsf{G^c}_i \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i \in I} \upharpoonright^{\mathsf{c}} \mathsf{r}\ !^{\mathsf{c}}[\mathsf{q}];\ \{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i \in I}}$$

[CO-PROJ-SEND-2]
$$\frac{\mathsf{r} \neq \mathsf{q} \qquad \forall i \in I.\mathsf{G^c}_i \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_i}{\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i \in I} \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_j}$$

[CO-PROJ-RECV-1]
$$\frac{\mathsf{r} = \mathsf{q} \qquad \forall i \in I.\mathsf{G^c}_i \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i \in I} \upharpoonright^{\mathsf{c}} \mathsf{r}\ ?^{\mathsf{c}}[\mathsf{p}];\ \{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i \in I}}$$

[CO-PROJ-RECV-2]
$$\frac{\mathsf{r} = \mathsf{q} \qquad \forall i \in I.\mathsf{G^c}_i \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_i}{\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i \in I} \upharpoonright^{\mathsf{c}} \mathsf{r}\ ?^{\mathsf{c}}[\mathsf{p}];\ \{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i \in I}}$$

[CO-PROJ-CONT]
$$\frac{\mathsf{r} \neq \mathsf{p} \quad \mathsf{r} \neq \mathsf{q} \quad \forall i \in I.\mathsf{G^c}_i \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_i \quad \forall i, j \in I.\mathsf{L^c}_i = \mathsf{L^c}_j \quad \forall i \in I.\mathsf{part\_of}\ \mathsf{r}\ \mathsf{G^c}_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i \in I} \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}_{\bar{\imath}} \ \text{(with } \bar{\imath} \in I)}$$

[CO-PROJ-END]
$$\frac{\neg\, \mathsf{part\_of}\ \mathsf{r}\ \mathsf{G^c}}{\mathsf{G^c} \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{end^c}}$$

(b) Rules for coinductive projection, Definition 3.4

Figure 3. Projection rules

and [CO-PROJ-RECV-2]). This is an adaptation to our coinductive setting of the definition in [16, Appendix A.1]. Below we give an example to clarify the meaning of [PROJ-CONT].

Example 3.5 (Projection)**.** About rule [PROJ-CONT], we observe that the type $\mathsf{G}' = \mathsf{Alice} \to \mathsf{Bob} : \{\ell_1(\mathsf{nat}).\mathsf{Bob} \to \mathsf{Carol} : \ell(\mathsf{nat}).\mathsf{end}, \ell_2(\mathsf{nat}).\mathsf{Alice} \to \mathsf{Carol} : \ell(\mathsf{nat}).\mathsf{end}\}$ is not projectable onto Carol, since, after skipping the first interaction between Alice and Bob, it would not be clear whether Carol should expect a message from Alice or from Bob. If we take instead $\mathsf{G} = \mathsf{Alice} \to \mathsf{Bob} : \{\ell_1(\mathsf{nat}).\mathsf{Bob} \to \mathsf{Carol} : \ell(\mathsf{nat}).\mathsf{end}, \ell_2(\mathsf{bool}).\mathsf{Bob} \to \mathsf{Carol} : \ell(\mathsf{nat}).\mathsf{end}\}$, the projection $\mathsf{G} \upharpoonright \mathsf{Carol}$ is well defined as the local type $\mathsf{L} = ?[\mathsf{Bob}]; \ell(\mathsf{nat}).\mathsf{end}$. Following common practice, we use an option type to encode projection as a partial function in Coq.

Coinductive projection is more permissive than its inductive counterpart, since it removes the technical issues related to formally dealing with (equi)recursion, allowing for a smoother development in Coq ([12] and [20, Definition 3.6 and Remark 3.14]).

If, when reasoning about semantics, coinductive trees are more convenient objects to work with, we still want to rely on session types for imposing a typing discipline on the communication. The followng theorem allows us to do so.

Theorem 3.6 (Unravelling preserves projections)**.** *(*`ic_proj` *in* `Projection/Correctness.v`*) Given a global type* G*, such that* guarded G *and* closed G*, if (a) there exists a local type* L *such that* $\mathsf{G} \upharpoonright \mathsf{r} = \mathsf{L}$ *, (b) there exists a global tree* $\mathsf{G^c}$ *such that* $\mathsf{G}\ \mathfrak{R}\ \mathsf{G^c}$ *and, (c) there exists a local tree* $\mathsf{L^c}$ *such that* $\mathsf{L}\ \mathfrak{R}\ \mathsf{L^c}$*, then* $\mathsf{G^c} \upharpoonright^{\mathsf{c}} \mathsf{r}\ \mathsf{L^c}$*.*

This first central result closes the first metatheory square (M.1) of the diagram in Figure 2. For a sketch of its proof see [12].

3.3 Projection Environments for Asynchronous Communication

In this subsection, we introduce key concepts for building an asynchronous operational semantics for MPST. In [16] a precise correspondence is drawn between communicating finite-state automata and MPST. We do not formalise an explicit syntax for automata, but develop labelled transition systems for global and local trees with automata in mind.

Consider the following scenario: p sends a message to q with label ℓ and payload of sort S and continues on $\mathsf{L^c}$, and dually q receives from p the message, with same label and payload, and then continues on $\mathsf{L^c}'$. For q to receive the message, it is necessary that p has first sent it. To model this asynchronous behaviour, we use FIFO queues: in the designated queue $Q(\mathsf{p}, \mathsf{q})$ (empty at first) we enqueue the message sent from p, until the message is received by q and removed from the queue. We use one queue for each ordered pair of participants (p, q) to store in-transit messages sent from p to q, and we collect such queues in *queue environments.*

Definition 3.7 (Queue environments)**.** We call *queue environment* (notation `qenv` in `Local/Semantics.v`) any finitely supported function that maps a pair of participants into a finite sequence (*queue*) of pairs of labels and sorts.

We define the operations of enqueuing and dequeuing on queue environments:

$$\begin{array}{lcl}
\mathsf{enq}\ Q\ (\mathsf{p}, \mathsf{q})\ (\ell, \mathsf{S}) & = & Q[(\mathsf{p}, \mathsf{q}) \hookleftarrow Q(\mathsf{p}, \mathsf{q})@(\ell, \mathsf{S})] \\
\mathsf{deq}\ Q\ (\mathsf{p}, \mathsf{q}) & = & \mathtt{if}\ Q(\mathsf{p}, \mathsf{q}) = (\ell, \mathsf{S})\#s \\
 & & \mathtt{then}\ ((\ell, \mathsf{S}), Q[(\mathsf{p}, \mathsf{q}) \hookleftarrow s]) \quad \mathtt{else\ None}
\end{array}$$

We use $\#$ as the "cons" constructor for lists and @ as the "append" operation; $f[x \hookleftarrow y]$ denotes the updating of a function f in x with y, namely $f[x \hookleftarrow y]\ x' = f\ x'$ for all $x \neq x'$ and $f[x \hookleftarrow y]\ x = y$. We use option types for partial functions, with None as the standard returned value where the function is undefined. In case the sequence $Q(\mathsf{p}, \mathsf{q})$ is empty deq will not perform any operation on it, but return None; in case the sequence is not empty it will return both its head and its tail (as a pair). We denote the empty queue environment by ϵ, namely $\epsilon\ (\mathsf{p}, \mathsf{q}) = \mathsf{None}$ for all (p, q).

Global trees can represent stages of the execution, where a participant has already sent a message, but it has not yet been received. We adapt the "queue projection" from [16, Appendix A.1] to our coinductive setting, to associate global trees to the queue contents of a system.

Definition 3.8 (Queue projection)**.** (Definition `qProject` in `Projection/QProject.v`) *Projection on queue environments of a global tree* (*queue projection* for short) is the relation `_`$\upharpoonright^{\mathsf{q}}$`_ : rel g_ty`$^{\mathsf{c}}$ qenv coinductively specified by:

$$[\text{Q-PROJ-SEND}]\ \frac{\forall i \in I. \mathsf{G^c}_i \upharpoonright^{\mathsf{q}} Q \qquad Q(\mathsf{p},\mathsf{q}) = \mathsf{None}}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \upharpoonright^{\mathsf{q}} Q} \qquad \frac{[\text{Q-PROJ-END}]}{\mathsf{end^c} \upharpoonright^{\mathsf{q}} \epsilon}$$

$$[\text{Q-PROJ-RECV}]\ \frac{\mathsf{G^c}_j \upharpoonright^{\mathsf{q}} Q \qquad \mathsf{deq}\ Q'(\mathsf{p},\mathsf{q}) = ((\ell_j, \mathsf{S}_j), Q)}{\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \upharpoonright^{\mathsf{q}} Q'}$$

See [12] for more details.

Analogously to queue environments, we consider all the local types of the protocol at once.

Definition 3.9 (Local environments)**.** We call *local environment*, or simply *environment*, any finitely supported function E that maps participants into local types.

We are interested in those environments that are defined on the participants of a global protocol $\mathsf{G^c}$ and that map each participant p to the projection of $\mathsf{G^c}$ onto such p.

Definition 3.10 (Environment projection)**.** (Definition `eProject` in `Projection/CProject.v`.) We say that E is an environment projection for $\mathsf{G^c}$, notation $\mathsf{G^c} \upharpoonright E$, if it holds that $\forall \mathsf{p}.\ \mathsf{G^c} \upharpoonright^{\mathsf{c}} \mathsf{p}\ (E\ \mathsf{p})$.

We define the semantics on a set of local types together with queue environments. We therefore consider the projection of a global tree both on local environments and on queue environments, together in one shot.

Definition 3.11 (One-shot projection)**.** (Definition `Projection` in `Projection.v`) We say that the pair of a local environment and of a queue environment (E, Q) is a (one-shot) projection for the global tree $\mathsf{G^c}$, notation $\mathsf{G^c} \upharpoonright\upharpoonright (E, Q)$ if it holds that: $\mathsf{G^c} \upharpoonright E$ and $\mathsf{G^c} \upharpoonright^{\mathsf{q}} Q$.

Example 3.12. Let us consider the global tree: $\mathsf{G^c} = \mathsf{p} \overset{\ell}{\rightsquigarrow} \mathsf{q} : \ell(\mathsf{S}).\mathsf{q} \to \mathsf{p} : \ell(\mathsf{S}).\mathsf{q} \to \mathsf{p} : \ell(\mathsf{S}).\dots$. Participant p has sent a message to q, q will receive it next (but has not yet) and then the protocol continues indefinitely with q sending a message to p after the other. We define E such that: $E\ \mathsf{p} = {?^{\mathsf{c}}}[\mathsf{q}];\ell(\mathsf{S}).{?^{\mathsf{c}}}[\mathsf{q}];\ell(\mathsf{S}).\dots$ and $E\ \mathsf{q} = {?^{\mathsf{c}}}[\mathsf{p}];\ell(\mathsf{S}).{!^{\mathsf{c}}}[\mathsf{p}];\ell(\mathsf{S}).{!^{\mathsf{c}}}[\mathsf{p}];\ell(\mathsf{S}).\dots$. We then define Q such that: $Q(\mathsf{p},\mathsf{q}) = [(\ell, \mathsf{S})]$ and $Q(\mathsf{q},\mathsf{p}) = \mathsf{None}$. It is easy to verify that $\mathsf{G^c} \upharpoonright\upharpoonright (E, Q)$; observe that the only "message" enqueued in Q is (ℓ, S), since this is the only one sent, but not yet received (at this stage of the execution).

3.4 Labelled Transition Relations for Tree Types

At the core of the *trace semantics* for session types lies a labelled transition system (LTS) defined on trees, with regard to *actions*. The basic *actions* (datatype `act` in `Common/Actions.v`) of our asynchronous communication are objects, ranged over by a, of the shape either: $!\mathsf{pq}(\ell, \mathsf{S})$: send ! action, from participant p to participant q, of label ℓ and payload type S, or $?\mathsf{qp}(\ell, \mathsf{S})$: receive ? action, from participant p at participant q, of label ℓ and payload type S. We define the *subject* of an action a (definition `subject` in `Common/Actions.v`), `subj` a, as p if $a = !\mathsf{pq}(\ell, \mathsf{S})$ and as q if $a = ?\mathsf{qp}(\ell, \mathsf{S})$.[3] Given an action, our types (represented as trees) can perform a reduction step.

Definition 3.13 (LTS for global trees)**.** (`step` in `Global/Semantics.v`) The *labelled transition relation for global trees* (*global reduction* or *global step* for short) is, for each action a, the relation `_` $\xrightarrow{a}$ `_ : rel g_ty`$^{\mathsf{c}}$ `g_ty`$^{\mathsf{c}}$ inductively specified by the following clauses:

$$[\text{G-STEP-SEND}]\ \frac{a = !\mathsf{pq}(\ell_j, \mathsf{S}_j)}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \xrightarrow{a} \mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}}$$

$$[\text{G-STEP-RECV}]\ \frac{a = ?\mathsf{qp}(\ell_j, \mathsf{S}_j)}{\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \xrightarrow{a} \mathsf{G^c}_j}$$

$$[\text{G-STEP-STR1}]\ \frac{\mathtt{subj}\ a \neq \mathsf{p} \quad \mathtt{subj}\ a \neq \mathsf{q} \quad \forall i \in I. \mathsf{G^c}_i \xrightarrow{a} \mathsf{G^c}'_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \xrightarrow{a} \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}'_i\}_{i\in I}}$$

$$[\text{G-STEP-STR2}]\ \frac{\mathtt{subj}\ a \neq \mathsf{q} \quad \mathsf{G^c}_j \xrightarrow{a} \mathsf{G^c}'_j \quad \forall i \in I\backslash\{j\}. \mathsf{G^c}_i = \mathsf{G^c}'_i}{\mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I} \xrightarrow{a} \mathsf{p} \overset{\ell_j}{\rightsquigarrow} \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}'_i\}_{i\in I}}$$

The step relation describes a labelled transition system for global trees with the following intuition: [G-STEP-SEND] *sending base case*: with the sending action $!\mathsf{pq}(\ell_j, \mathsf{S}_j)$, a message with label ℓ_j and payload type S_j is sent by p, but not yet received by q; [G-STEP-RECV] *receiving base case*: with the receiving action $?\mathsf{qp}(\ell_j, \mathsf{S}_j)$, a message with label ℓ_j and payload type S_j, previously sent by p, is now received by q; in [G-STEP-STR1], a step is allowed to be performed under a sending constructor $\mathsf{p} \to \mathsf{q}$: each time that the subject of that action is different from p and from q and each continuation steps; [G-STEP-STR2] with an action a a step is allowed to be performed under a receiving constructor: each time that the subject of that action is different from q (p has already sent the message and the label ℓ_j has already been selected), the continuation corresponding to ℓ_j steps and others stay as the same.

This semantics allows for some degree of non-determinism. For instance, $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G^c}_i\}_{i\in I}$ could perform a step according to both rules [G-STEP-SEND] and [G-STEP-STR1] (depending on the subject of the action).

Below we define a transition system for environments of local trees, together with environments of queues.

Definition 3.14 (LTS for environments)**.** (`l_step` in `Local/Semantics.v`) The *labelled transition relation for environments* (*local reduction* or *local step* for short) is, for each

[3] The representation of actions is directly taken from [16], however we have swapped the order of p and q in the receive action, so that the subject of an action always occurs in first position.

$$\begin{tikzcd}
\mathsf{p} \to \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "!\mathsf{pq}(\ell{,}\mathsf{S})", "(g.1)"'] \arrow[d, "(p.1)"', "\upharpoonright_{\mathsf{p}}"] & \mathsf{p} \overset{\ell}{\rightsquigarrow} \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "?\mathsf{qp}(\ell{,}\mathsf{S})", "(g.2)"'] \arrow[dr, "\upharpoonright_{\mathsf{p}}", "(p.2)"'] & \mathsf{G}^{\mathsf{c}} \arrow[d, "(p.3)"', "\upharpoonright_{\mathsf{p}}"] \\
!^{\mathsf{c}}[\mathsf{q}];(\ell, \mathsf{S}).\mathsf{L}^{\mathsf{c}} \arrow[rr, "!\mathsf{pq}(\ell{,}\mathsf{S})", "(l.1)"'] & & \mathsf{L}^{\mathsf{c}}
\end{tikzcd}$$

(a) p sends

$$\begin{tikzcd}
\mathsf{p} \to \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "!\mathsf{pq}(\ell{,}\mathsf{S})", "(g.1)"'] \arrow[d, "(q.1)"', "\upharpoonright_{\mathsf{q}}"] & \mathsf{p} \overset{\ell}{\rightsquigarrow} \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "?\mathsf{qp}(\ell{,}\mathsf{S})", "(g.2)"'] \arrow[dl, "(q.2)"', "\upharpoonright_{\mathsf{q}}"] & \mathsf{G}^{\mathsf{c}} \arrow[d, "(q.3)"', "\upharpoonright_{\mathsf{q}}"] \\
?^{\mathsf{c}}[\mathsf{p}];(\ell, \mathsf{S}).\mathsf{L}^{\mathsf{c}\prime} \arrow[rr, "?\mathsf{qp}(\ell{,}\mathsf{S})", "(l.2)"'] & & \mathsf{L}^{\mathsf{c}\prime}
\end{tikzcd}$$

(b) q receives

Figure 4. Basic send/receive steps for global and local trees

a, the relation $\_ \xrightarrow{a} \_$: `rel (renv * qenv) (renv * qenv)` inductively specified by the following clauses:

$$[\textsc{l-step-send}]\quad \frac{a = !\mathsf{pq}(\ell_j, \mathsf{S}_j) \quad E\ \mathsf{p} = !^{\mathsf{c}}[\mathsf{q}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I}}{(E, Q) \xrightarrow{a} (E[\mathsf{p} \hookleftarrow \mathsf{L^c}_j], \mathsf{enq}\ Q\ (\mathsf{p},\mathsf{q})\ (\ell_j, \mathsf{S}_j))}$$

$$[\textsc{l-step-recv}]\quad \frac{\begin{array}{l} a = ?\mathsf{qp}(\ell_j, \mathsf{S}_j) \\ E\ \mathsf{q} = ?^{\mathsf{c}}[\mathsf{p}];\{\ell_i(\mathsf{S}_i).\mathsf{L^c}_i\}_{i\in I} \quad Q(\mathsf{p},\mathsf{q}) = (\ell_j, \mathsf{S}_j)\#s \end{array}}{(E, Q) \xrightarrow{a} (E[\mathsf{q} \hookleftarrow \mathsf{L^c}_j], Q[(\mathsf{p},\mathsf{q}) \hookleftarrow s])}$$

Example 3.15 (Basic steps for global and local trees)**.** Figure 4a shows the transitions for a global tree, regulating the sending of a message from p to q, and the local transition for its projection on p. The asynchronicity of our system is witnessed by the two different steps: $(g.1)$, for the sending action $!\mathsf{pq}(\ell, \mathsf{S})$, and $(g.2)$, for the receiving one $?\mathsf{qp}(\ell, \mathsf{S})$. Projecting $\mathsf{p} \to \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}}$ on p (arrow $(g.1)$) gives us a local tree that performs a sending step $(l.1)$ corresponding to $(g.1)$, and projection is preserved (arrow $(p.2)$). However this does not happen for the receiving step $(g.2)$: here the projections on p of $\mathsf{p} \overset{\ell}{\rightsquigarrow} \mathsf{q} :(\ell, \mathsf{S}).\mathsf{G}^{\mathsf{c}}$ along $(p.2)$ and of G^{c} along $(p.3)$ are the same. Dually if we consider the projection on the receiving participant q, Figure 4b. Here the projections along $(q.1)$ and $(q.2)$, corresponding to the global tree performing a sending action, result in the same local tree. We have instead a local step $(l.2)$ preserving the local projections on q along $(q.2)$ and $(q.3)$ for the receiving action along $(g.2)$.

Figure 4 confirms our intuition: when the global tree performs one step, *there is one local tree* (namely, one projection of the global tree) such that it performs a corresponding step. We have indeed defined semantics for collections of local trees, as opposed to single local trees. The formal relation of the small-step reductions with respect to projection is established with soundness and completeness results (see [12] for proof outlines).

Theorem 3.16 (Step Soundness)**.** *(Theorem* `Project_step` *in* `TraceEquiv.v`*) If* $\mathsf{G}^{\mathsf{c}} \xrightarrow{a} \mathsf{G}^{\mathsf{c}\prime}$ *and* $\mathsf{G}^{\mathsf{c}} \upharpoonleft\upharpoonright (E, Q)$*, there exist* E' *and* Q' *such that* $\mathsf{G}^{\mathsf{c}\prime} \upharpoonleft\upharpoonright (E', Q')$ *and* $(E, Q) \xrightarrow{a} (E', Q')$.

Theorem 3.17 (Step Completeness)**.** *(Theorem* `Project_lstep` *in* `TraceEquiv.v`*) If* $(E, Q) \xrightarrow{a} (E', Q')$ *and* $\mathsf{G}^{\mathsf{c}} \upharpoonleft\upharpoonright (E, Q)$*, there exist* $\mathsf{G}^{\mathsf{c}\prime}$ *such that* $\mathsf{G}^{\mathsf{c}\prime} \upharpoonleft\upharpoonright (E', Q')$ *and* $\mathsf{G}^{\mathsf{c}} \xrightarrow{a} \mathsf{G}^{\mathsf{c}\prime}$.

3.5 Trace Semantics and Trace Equivalence

We finally show trace equivalence for global and local types with our Coq development of semantics for coinductive trees.

Definition 3.18 (Traces)**.** (Codatatype `trace` in `Action.v`), ranged over by t, are terms generated *coinductively* by $t ::= [] \mid a\#t$ where a is any action, as defined in § 3.4[4].

We associate traces to the execution of global trees and local environments.

Definition 3.19 (Admissible traces for a global tree)**.** We say that a trace is admissible for a global tree if the coinductive relation $\mathtt{tr}^{\mathtt{g}}$ _ _ (definition `g_lts` in `Global/Semantics.v`) holds:

$$\frac{}{\mathtt{tr}^{\mathtt{g}}\ []\ \mathsf{end}^{\mathsf{c}}} \qquad \frac{\mathsf{G}^{\mathsf{c}} \xrightarrow{a} \mathsf{G}^{\mathsf{c}\prime} \quad \mathtt{tr}^{\mathtt{g}}\ t\ \mathsf{G}^{\mathsf{c}\prime}}{\mathtt{tr}^{\mathtt{g}}\ a\#t\ \mathsf{G}^{\mathsf{c}}}$$

Definition 3.20 (Admissible traces for environments)**.** We say that a trace is admissible for a pair of a local environment and a queue environment if the coinductive relation $\mathtt{tr}^{\mathtt{l}}$ _ _ (definition `l_lts` in `Local/Semantics.v`) holds:

$$\frac{\forall \mathsf{p}.E\ \mathsf{p} = \mathsf{None}}{\mathtt{tr}^{\mathtt{l}}\ []\ (E, \epsilon)} \qquad \frac{(E, Q) \xrightarrow{a} (E', Q') \quad \mathtt{tr}^{\mathtt{l}}\ t\ (E', Q')}{\mathtt{tr}^{\mathtt{l}}\ a\#t\ (E, Q)}$$

Observe that generally more than one execution trace are admissible for a global tree or for an environment[5].

We can now state the *trace equivalence* theorem, our final result for multiparty session types. We sketch an outline of the proof in [12].

Theorem 3.21 (Trace equivalence)**.** *(Theorem* `TraceEquivalence` *in* `TraceEquiv.v`*)*
If $\mathsf{G}^{\mathsf{c}} \upharpoonleft\upharpoonright (E, Q)$*, then* $\mathtt{tr}^{\mathtt{g}}\ t\ \mathsf{G}^{\mathsf{c}}$ *if and only if* $\mathtt{tr}^{\mathtt{l}}\ t\ (E, Q)$.

Trace equivalence for global and local types (trees) concludes our formalisation of the metatheory of multiparty session types: squares (M.1) and (M.2) of the diagramin Figure 2. In the next section we specify a language for communicating systems inside Coq and extend extend the trace equivalence result to well-typed processes.

4 A Certified Process Language

This section defines Zooid, an embedded domain specific language in Coq for specifying certified multiparty processes. Zooid combines shallow and deep embedding: on one hand process actions are deeply embedded, represented as an inductive type; on the other, the exchanged values, and computations applied to them are a shallow embedding expressed as

[4] For traces, we use the same notation as for lists, however we bear in mind that this definiton is coinductive: it generates possibly infinite streams.

[5] About non-determinism in our semantics, see [12].

$$\text{[P-TY-END]}\ \frac{}{\Gamma \vdash_{\mathrm{LT}} \mathtt{finish} : \mathtt{end}} \qquad \text{[P-TY-JUMP]}\ \frac{}{\Gamma \vdash_{\mathrm{LT}} \mathtt{jump}\ X : X} \qquad \text{[P-TY-LOOP]}\ \frac{\Gamma \vdash e : \mathtt{Proc} \quad \Gamma \vdash_{\mathrm{LT}} e : \mathsf{L}}{\Gamma \vdash_{\mathrm{LT}} \mathtt{loop}\ X\ \{e\} : \mu X.\mathsf{L}}$$

$$\text{[P-TY-READ]}\ \frac{\Gamma \vdash \mathtt{act}_r : \mathtt{unit} \longrightarrow \llbracket \mathsf{S} \rrbracket \quad \Gamma, x : \llbracket \mathsf{S} \rrbracket \vdash_{\mathrm{LT}} e : \mathsf{L}}{\Gamma \vdash_{\mathrm{LT}} \mathtt{read}\ \mathtt{act}_r\ (x.e) : \mathsf{L}}$$

$$\text{[P-TY-WRITE]}\ \frac{\Gamma \vdash \mathtt{act}_w : \llbracket \mathsf{S} \rrbracket \longrightarrow \mathtt{unit} \quad \Gamma \vdash e_v : \llbracket \mathsf{S} \rrbracket \quad \Gamma \vdash_{\mathrm{LT}} e : \mathsf{L}}{\Gamma \vdash_{\mathrm{LT}} \mathtt{write}\ \mathtt{act}_w\ e_v\ e : \mathsf{L}}$$

$$\text{[P-TY-SEND]}\ \frac{\Gamma \vdash e_1 : \llbracket \mathsf{S}_j \rrbracket \quad \Gamma \vdash e_2 : \mathtt{Proc} \quad \Gamma \vdash_{\mathrm{LT}} e_2 : \mathsf{L}_j \quad j \in I}{\Gamma \vdash_{\mathrm{LT}} \mathtt{send}\ \mathtt{p}\ (\ell_j, e_1).\ e_2 : ![\mathtt{p}]; \{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i \in I}}$$

$$\text{[P-TY-RECV]}\ \frac{\forall i \in I \quad \Gamma \vdash e_i : \llbracket \mathsf{S}_i \rrbracket \longrightarrow \mathtt{Proc} \quad \Gamma, x : \llbracket \mathsf{S}_i \rrbracket \vdash_{\mathrm{LT}} e_i\ x : \mathsf{L}_i}{\Gamma \vdash_{\mathrm{LT}} \mathtt{recv}\ \mathtt{p}\ \{\ell_i.e_i\}_{i \in I} : ?[\mathtt{p}]; \{\ell_i(\mathsf{S}_i).\mathsf{L}_i\}_{i \in I}}$$

$$\text{[P-TY-INTERACT]}\ \frac{\Gamma \vdash \mathtt{act}_i : \llbracket \mathsf{S} \rrbracket \longrightarrow \llbracket \mathsf{S}' \rrbracket \quad \Gamma \vdash e_v : \llbracket \mathsf{S} \rrbracket \quad \Gamma, x : \llbracket \mathsf{S}' \rrbracket \vdash_{\mathrm{LT}} e : \mathsf{L}}{\Gamma \vdash_{\mathrm{LT}} \mathtt{interact}\ \mathtt{act}_i\ e_v\ (x.e) : \mathsf{L}}$$

Figure 5. Process Typing System

Gallina terms. The core process calculus of Zooid is session-typed, where the typing derivation is described as a Coq inductive predicate. The constructs of Zooid are *smart constructors* that build both a process, and a proof that this is well-typed with respect to a given local type. Each process is single threaded and the concurrent semantics occurs due to the asynchronous nature of the channels.

4.1 Core Processes

The core process calculus of Zooid differs to those generally used in the session-types literature in several aspects. First, the combination of shallow and deep embedding implies that a process may be defined in terms of a larger expression of the ambient calculus. Secondly, the process calculus does not include parallel composition. Just as *"zooid"*, in biology, is used to refer to the single individual in a colonial organism, a process proc implements the behaviour of a single participant in the distributed system: we are interested in certifying processes in isolation to the larger system. This approach plays well with the usual MPST methodology and it admits heterogenous development, as in one can use Zooid for the critical roles and other roles can be implemented in different languages, using different frameworks.

Definition 4.1 (Syntax of untyped processes)**.** Processes, proc (definition Proc in Proc.v), are embedded in an ambient calculus e. In our implementation, proc is the inductive type of processes, of type Proc, and the ambient calculus is Gallina, the specification language of Coq.

$$\begin{array}{rcl} e & ::= & \mathtt{proc} \mid e + e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\ & \mid & \mathtt{fun}\ x \Rightarrow e \mid \ldots \qquad n \in \mathbb{N} \qquad \ell_i \in \mathbb{N} \\ \mathtt{proc} \in \mathtt{Proc} & ::= & \mathtt{finish} \mid \mathtt{jump}\ X \mid \mathtt{loop}\ X\ \{e\} \\ & \mid & \mathtt{recv}\ \mathtt{p}\ \{\ell_i.e_i\}_{i \in I} \mid \mathtt{send}\ \mathtt{p}\ (\ell, e).\ e \\ & \mid & \mathtt{read}\ \mathtt{act}_r\ (x.e) \mid \mathtt{write}\ \mathtt{act}_w\ e_v\ e \\ & \mid & \mathtt{interact}\ \mathtt{act}_i\ e_v\ (x.e) \end{array}$$

The constructs of Proc mirror those of local types: finish is the ***ended*** process; jump X is a ***jump*** to recursion variable X; loop X $\{e\}$ is a ***recursive process***, built by expression e, that introduces a new recursion variable X; recv p $\{\ell_i.e_i\}_{i \in I}$ is the process ***receiving*** from p a message with label ℓ_i, a value x, and continues as $(e_i\ x)$; and send p (ℓ, e_1). e_2 is the ***sending*** process with label ℓ and expression e_1 to participant p, and then continues as e_2. Our calculus does not include parallel composition: we assume that the system is implemented as the parallel composition of all the participants. For example, the following is a process that receives requests from a participant p and replies increasing the received number by m, until p chooses to finish:

$$\begin{array}{rl} \mathtt{proc}_{\mathtt{q}} = & \mathtt{loop}\ X\{\mathtt{recv}\ \mathtt{p}\{\ell_1.\mathtt{fun}\ x \Rightarrow \mathtt{send}\ \mathtt{p}\ (\ell_1, x + m). \\ & \qquad\qquad \mathtt{jump}\ X; \ell_2.\mathtt{fun}\ x \Rightarrow \mathtt{finish}\}\} \end{array}$$

A process can be defined mixing Gallina terms and proc. For example, in the process above, the term $x + m$ is a term in Gallina. These Gallina terms can be used to specify branching in the control flow of the process. The process below is one possible implementation for p that loops until the value received is greater than some threshold n:

$$\begin{array}{rcl} e_{\mathtt{p}} & = & \mathtt{fun}\ x \Rightarrow \mathtt{if}\ x > n\ \mathtt{then}\ \mathtt{send}\ \mathtt{q}\ (\ell_2, \mathtt{tt}).\ \mathtt{finish} \\ & & \mathtt{else}\ \mathtt{send}\ \mathtt{q}\ (\ell_1, x).\ \mathtt{jump}\ X \\ \mathtt{proc}_{\mathtt{p}} & = & \mathtt{send}\ \mathtt{q}\ (\ell_1, 0).\ \mathtt{loop}\ X\ \{\mathtt{recv}\ \mathtt{q}\ \{\ell_1.e_{\mathtt{p}}\}\} \end{array}$$

Zooid processes interact with their environment by calling functions written in the language of the runtime (OCaml in this case). These functions exchange information between Zooid and the environment in a safe way by not exposing channels or the transport API. The interaction happens by calling an external function: $\mathtt{act}_r$, $\mathtt{act}_w$, and $\mathtt{act}_i$ for reading, writing or interacting with the environment. $\mathtt{act}_r$ is a function that takes a unit and returns a value of payload type (i.e.: a coq_ty T for some type T). $\mathtt{act}_w$ is a function that takes a parameter of payload type and returns unit, allowing the process to call OCaml to print on the screen or write to file or similar things. Finally $\mathtt{act}_i$ is the action function that passes data to the OCaml runtime and receives some response, thus combining the two other environment interaction functions. These functions *do not affect the communication structure* of the process: they are internal actions and do not appear in the trace of the process.

Definition 4.2 (Process typing system)**.** We define typing for processes $\Gamma \vdash_{\mathrm{LT}} e : \mathsf{L}$ in Figure 5, as an inductive predicate in Coq (definition of_lt in Proc.v). Since proc is embedded in Coq, we assume the standard typing judgement for Gallina terms, of the form $\Gamma \vdash e : T$. We assume a set of sorts S_j, and an encoding as a Coq type $\llbracket \mathsf{S}_j \rrbracket$ (see Definition 3.1).

Rules [P-TY-END], [P-TY-JUMP], and [P-TY-LOOP] state that the local type of the ended process, a jump to X, and recursion are end, X, and a recursive type respectively. Rule [P-TY-SEND] specifies that a send process with label ℓ has a send type, if ℓ is in the set of accepted labels. Rule [P-TY-RECV] specifies that a receive process has a receive type, if all the alternatives have the correct local type for all possible payloads $x : [\![\mathsf{S}_i]\!]$. Any expression e that does not match any of these rules must be proven to be of the correct type for all of its possible reductions. For example, it is straightforward to prove that if $\Gamma \vdash_{\text{LT}} e_t : \mathsf{L}$ and $\Gamma \vdash_{\text{LT}} e_f : \mathsf{L}$ then $\Gamma \vdash_{\text{LT}} \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : \mathsf{L}$ by case analysis on e. Finally, rules [P-TY-READ], [P-TY-WRITE], and [P-TY-INTERACT], have no impact on the local type, so they simply check that the actions are well typed, and that the continuation process has the expected type.

4.2 Zooid

In the Coq library `Zooid.v`, Zooid terms (ranged over by Z) are dependent pairs of a `proc`, and a proof that it is well-typed with respect to a given local type L.

```
Definition wt_proc L := { P : Proc | of_lt P L }.
```

They are built using smart constructors, helper functions and notations to define processes that are well-typed by construction (i.e.: a process and a witness of its type derivation). Moreover, we take care that the local type of each smart constructor is fully determined by their inputs, so we can use Coq to *infer* the local type of each of these processes. Given a Zooid expression Z, we can project the *first component* to extract the underlying `proc` term. Since the behaviour of alternatives in Z terms is fully specified, we can infer its local type. By construction, if a term Z can be defined, then its underlying `proc` is well-typed with respect to some local type L, *second component* of the dependent pair.

The simplest example is the `finish` term for inactive processes of type `l_end`. Coq infers most parameters.

```
Definition wt_end : wt_proc l_end := exist _ _ t_Finish.
Notation finish := wt_end.
```

On the other hand, the notation `\send` is defined in the same way, but the definition of the dependent pair requires a simple proof (i.e.: `wt_send`). The send command is implemented using a singleton choice, and this proof simply says that this label is the one in the singleton choice. The definition is as follows:

```
Definition wt_send p l T (pl : coq_ty T) L (P : wt_proc L)
  : wt_proc (l_msg l_send p [::(l, (T, L))])
  := exist _ _ (t_Send p pl (of_wt_proc P)
                           (find_cont_sing l T L)).
Notation "\send" := wt_send.
```

Despite not being directly encoded as a Coq datatype, Figure 6 presents the syntax for Zooid terms in BNF notation. The syntactic constructs are the expected, with only a few differences: (a) `if then else` is a Zooid construct since it needs to carry the proof that the underlying proc is well–typed; (b) branch and select must take a list of alternatives (Z^b and Z^s respectively), and send/receive are defined as branch/select with a singleton alternative. The alternatives for branch, Z^b, are pairs of labels and continuations. The alternatives for `select`, Z^s are:
(1) $\mathtt{case}\ e_1 \Rightarrow \ell,\ e_2 : \mathsf{S}!\ \mathsf{Z}$, specifies to send ℓ and $e_2 : [\![\mathsf{S}]\!]$ and then continue as Z, when e_1 evaluates to `true`;
(2) $\mathtt{otherwise} \Rightarrow \ell,\ e_2 : \mathsf{S}!\ \mathsf{Z}$, specifies that the default alternative is to send ℓ and e_2, and then continue as Z; and
(3) $\mathtt{skip} \Rightarrow \ell,\ \mathsf{S}!\ \mathsf{L}$, specifies the unimplemented alternative of sending ℓ and a value of sort S, and then continuing as L. We require `skip` to enforce a unique local type: since Definition 4.2 does not include subtyping, Zooid requires that all the possible behaviours in the local type must be either implemented or declared. We impose a syntactic condition on `select`: there must be exactly one default case, which must occur after the last `case`. The three constructs to interact with external code (`read`, `write`, and `interact`) are similar to their untyped counterparts from § 4.1. These actions do not impact the traces nor the local types, so they simply sport the local type of their continuations.

Definition 4.3 (Zooid syntax)**.**

$$
\begin{array}{rcl}
\mathsf{Z}^b & ::= & \ell,\ x : \mathsf{S}?\ \mathsf{Z} \\
\mathsf{Z}^s & ::= & \mathtt{case}\ e \Rightarrow \ell,\ e : \mathsf{S}!\ \mathsf{Z} \mid \mathtt{skip} \Rightarrow \ell,\ \mathsf{S}!\ \mathsf{L} \\
 & & \mid \mathtt{otherwise} \Rightarrow \ell,\ e : \mathsf{S}!\ \mathsf{Z} \\
\mathsf{Z} & ::= & \mathtt{jump}\ X \mid \mathtt{loop}\ X\ (\mathsf{Z}) \mid \mathtt{if}\ e\ \mathtt{then}\ \mathsf{Z}\ \mathtt{else}\ \mathsf{Z} \\
 & & \mid \mathtt{send}\ \mathsf{p}\ (\ell,\ e : \mathsf{S})!\ \ \mathsf{Z} \mid \mathtt{recv}\ \mathsf{p}\ (\ell,\ x : \mathsf{S})?\ \ \mathsf{Z} \\
 & & \mid \mathtt{finish} \mid \mathtt{branch}\ \mathsf{p}\ \ [\mathsf{Z}^b{}_1 \mid \ldots \mid \mathsf{Z}^b{}_n] \\
 & & \mid \mathtt{select}\ \mathsf{p}\ \ [\mathsf{Z}^s{}_1 \mid \ldots \mid \mathsf{Z}^s{}_n] \\
 & & \mid \mathtt{read}\ \mathtt{act}_r\ (x.\mathsf{Z}) \mid \mathtt{write}\ \mathtt{act}_w\ e\ \mathsf{Z} \\
 & & \mid \mathtt{interact}\ \mathtt{act}_i\ e\ (x.\mathsf{Z})
\end{array}
$$

Figure 6. Zooid Syntax

4.3 Semantics of Zooid

The semantics of Zooid is defined as a labelled transition system of the underlying `proc` terms, analogously to that of local type trees in Definition 3.14[6], but with values instead of sorts in the trace, and explicitly unfolding recursion.

Definition 4.4 (LTS for processes)**.** The *LTS for processes* is, for each action a, defined as:

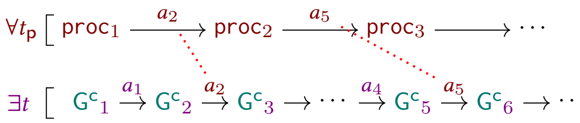
$$
[\text{P-STEP-SEND}]\ \frac{a = !\mathsf{pq}(\ell, e_1)}{\mathtt{send}\ \mathsf{q}\ (\ell, e_1).\ e_2 \xrightarrow{a} e_2}
\qquad
[\text{P-STEP-RECV}]\ \frac{a = ?\mathsf{qp}(\ell_{\bar{\imath}}, e)}{\mathtt{recv}\ \mathsf{p}\ \{\ell_i.e_i\}_{i \in I} \xrightarrow{a} (e_{\bar{\imath}}\ e)}
$$

$$
[\text{P-STEP-LOOP}]\ \frac{[(\mathtt{loop}\ X\ \{e\})/(\mathtt{jump}\ X)]e \xrightarrow{a} e'}{(\mathtt{loop}\ X\ \{e\}) \xrightarrow{a} e'}
$$

The steps of the LTS are: [P-STEP-SEND] states that a send process transitions to the continuation e_2 with the action that sends a label ℓ and value e_1; [P-STEP-RECV] states that a receive process transitions to $(e_{\bar{\imath}}\ e)$ with the receive action from participant p; and [P-STEP-LOOP] unfolds recursion once to perform a step on a recursive process.

We prove the type preservation for $\vdash_{\text{LT}}$. To show this, we need to relate process actions with local/global type actions. This is done by a simple erasure that removes the values, but preserves the types in an action, denoted by $|a|$. For example, if $a = !\mathsf{pq}(\ell, e)$ and $e : [\![\mathsf{S}]\!]$, then $|a| = !\mathsf{pq}(\ell, \mathsf{S})$.

[6] For the sake of uniformity, here we present the LTS for processes as a relation, however in Coq we define it, equivalently, as a recursive function: `do_step_proc` in `Proc.v`.



$\forall t_p$ [$\mathsf{proc}_1 \xrightarrow{a_2} \mathsf{proc}_2 \xrightarrow{a_5} \mathsf{proc}_3 \longrightarrow \cdots$
$\exists t$ [$\mathsf{G^c}_1 \xrightarrow{a_1} \mathsf{G^c}_2 \xrightarrow{a_2} \mathsf{G^c}_3 \rightarrow \cdots \xrightarrow{a_4} \mathsf{G^c}_5 \xrightarrow{a_5} \mathsf{G^c}_6 \rightarrow \cdots$

Figure 7. Theorem 4.7, visually.

Theorem 4.5 (Type preservation)**.** *(Theorem* `preservation` *in the file* `Proc.v`*.) If* $\Gamma \vdash_{LT} e : \mathsf{L}$ *and* $e \xrightarrow{a} e'$*, then there exists* L' *such that* $\mathsf{L} \xrightarrow{|a|} \mathsf{L}'$*, and* $\Gamma \vdash_{LT} e' : \mathsf{L}'$*.*

We write $\mathsf{tr^p}\ t\ e$ to express that a trace t is admissible by process e. The formal definition goes analogously to Definition 3.20 for tr^1 _ _; note, however, that the admission of a trace by process is checked *in isolation* to other processes. To relate process traces to global/local type traces we need to define the notion of a *complete subtrace.*

Definition 4.6 (Complete subtrace)**.** We say that t_1 is a complete subtrace of t_2 for participant p (definition `subtrace` in `Local.v`), if all actions in t_2 that have p as a subject occur in t_1 in the same relative position (i.e. the n-th action of p in t_2 must be the n-th action of t_1). We write $t_1 \preceq_{\mathsf{p}} t_2$ as the greatest relation satisfying:

$$\frac{\mathsf{subj}\ a \neq \mathsf{p} \quad t_1 \preceq_{\mathsf{p}} t_2}{t_1 \preceq_{\mathsf{p}} (a\#t_2)} \qquad \frac{\mathsf{subj}\ a = \mathsf{p} \quad t_1 \preceq_{\mathsf{p}} t_2}{(a\#t_1) \preceq_{\mathsf{p}} (a\#t_2)} \qquad \frac{}{[] \preceq_{\mathsf{p}} []}$$

The main result for Zooid states that for all admissible traces for a well-typed process, there exists at least a trace in the larger system that is a complete supertrace of that of the process. We state this formally as Theorem 4.7 (`process_traces_are_global_types` in `Proc.v`). Thus, well-typed processes inherit the global type properties of protocol compliance, deadlock freedom and liveness.

Theorem 4.7 (Process and global type traces)**.** *Let* $\mathsf{G^c} \upharpoonleft\upharpoonright (E, \epsilon)$ *and* $\Gamma \vdash_{LT} e : \mathsf{L}$ *such that* $\mathsf{L}\ \mathfrak{R}\ (E\ \mathsf{p})$*. Then, for all traces* t_{p} *such that* $\mathsf{tr^p}\ t_{\mathsf{p}}\ e$ *there exists a trace* t *such that* $\mathsf{tr^g}\ t\ \mathsf{G^c}$*, and* $|t_{\mathsf{p}}| \preceq_{\mathsf{p}} t$*.*

Figure 7 presents the meaning of the above theorem graphically. Any trace $t_{\mathsf{p}} = a_2\#a_5\#\ldots$ of a process proc is contained within a larger system trace $t = a_1\#a_2\#a_3\#\ldots$ of $\mathsf{G^c}$, given that proc behaves as some participant p in $\mathsf{G^c}$. Namely, if a process e is well typed with a local type L, which is equal up to unravelling to that of participant p in $\mathsf{G^c}$, then the behaviour of e is that of p in $\mathsf{G^c}$.

4.4 Extraction

Terms of type `Proc`, in Coq, can be easily extracted to executable OCaml code, following an approach similar to that of Interaction Trees [51]: we can substitute the occurrences of `proc` terms by a suitable OCaml handler. Figure 8 shows the declaration of a module for that purpose.

Module `ProcessMonad` specifies a monadic type `t`, that supports the standard `bind` and `pure` operations, as well as constructs for adding the required effects, in this case network communication and looping (with potential non-termination). During extraction this module becomes the ambient monad for the extracted code. In order to run the code the user instantiates the monad to provide a low level implementation, which fills in the details about the network transport. Zooid processes are translated into the monad using the function `extract_proc` from `Proc.v`. [12] shows the function in its entirety.

```
Module ProcessMonad.
  Parameter t : Type → Type.
  (* monadic bind and pure values *)
  Parameter bind : forall T1 T2, t T1 →
    (T1 → t T2) → t T2.
  Parameter pure : forall T1, T1 → t T1.
  (* actions to send and receive *)
  Parameter send : forall T, role → lbl → T → t unit.
  Parameter recv : (lbl → t unit) → t unit.
  Parameter recv_one : forall T, role → t T.
  (* actions for setting up a loop and jumping *)
  Parameter loop : forall T1, nat → t T1 → t T1.
  Parameter set_current: nat → t unit.
  (* function to run the monad *)
  Parameter run : forall A, t A → A.
End ProcessMonad.
```

Figure 8. The Process Monad.

4.5 Runtime

The code for an endpoint process is extracted as a value inside of the process monad from § 4.4. Zooid's runtime provides an implementation of `ProcessMonad`. The endpoint process is independent of the transport and network protocols; the exact specification of those is deferred to the implementation of the monad. The runtime implements the monad relying on the monad provided by OCaml's Lwt library[7], as well as its asynchronous communication primitives. The transport uses TCP/IP and the payloads are encoded and decoded using the 'Marshal' module in OCaml's standard library[8]. This design prioritises OCaml based technologies to implement asynchronous I/O and data encoding. Other transports are possible (e.g., web services over HTTP).

4.5.1 Implementation. In Zooid, the user implements their processes in the DSL, then uses Coq to produce OCaml code for the monad's module type and for the process, using extraction. The runtime implements a means to run that code. Concretely it provides the transport and serialization.

A runnable process amounts to an instance of the functor type in Figure 9, in which we provide the process monad instance together with the extracted process.

Communication primitives in processes are unaware of transport or other networking issues, they simply expect to be able to communicate with the other roles involved in the

[7] https://ocsigen.org/lwt/5.2.0/manual/manual
[8] https://ocaml.org/releases/4.11/htmlman/libref/Marshal.html

```
module type PROCESS_FUNCTOR =
  functor (MP : ProcessMonad) -> sig
    module PM : sig
      type 'x t = 'x MP.t
      val run : 'a1 t -> 'a1
      val send : role -> lbl -> 'a1 -> unit t
      val recv :
        role -> (lbl -> unit t) -> unit t
      val recv_one : role -> 'a1 t
      val bind : 'a1 t -> ('a1 -> 'a2 t) -> 'a2 t
      val pure : 'a1 -> 'a1 t
      val loop : var -> (unit -> 'a1 t) -> 'a1 t
      val set_current : var -> unit t
    end
    val proc : unit MP.t
  end
```

Figure 9. The Process Functor.

protocol. The runtime implementation requires the user to provide for each role a list of channels to communicate with the other roles. It is specified as:

```
type connection_spec
  = Server of sockaddr | Client of sockaddr
type conn_desc =
  { role_to : role; spec : connection_spec }
```

where each process needs to specify a `conn_desc list` detailing a channel to each role where it either starts a connection (using the `Client` connector and specifying IP and port in the `sockaddr` datatype) or waits for a connection (in a similar way using the `Server` constructor).

So finally, the runtime is invoked by calling the function:

```
val execute_extracted_process
  : conn_desc list -> (module PROCESS_FUNCTOR) -> unit
```

which connects a participant to all the roles as specified in the connection list and executes extracted process passed as first-class module value to the function. If the extracted process interacts with OCaml code, the library that implements all the external functions has to be compiled into the executable.

With the addition of the runtime Zooid processes become certified code that can be readily executed to implement distributed multiparty services.

5 Evaluation: Certified Processes

This section displays several common use cases in the MPST literature, implemented and certified using Zooid: (1) several implementations of a recursive ping-pong protocol; (2) a recursive pipeline; and (3) the two-buyer protocol from [27]. We conclude the section with a summary evaluating our mechanisation effort.

A Common Workflow. Our workflow consists of the following steps: (1) specify the global type for the protocol; (2) project the global type into the set of local types; (3) implement a process using Zooid; (4) (if necessary) prove that the local type of the process is equal up to unravelling to the projection of some participant; (5) use extraction to OCaml; and (6) implement external OCaml actions (if any).

Steps (1), (3), and (6) are the necessary inputs for implementing a certified process. Steps (2) and (5) are fully automated, and step (4) is often automated too, although it may require a simple manual proof. Finally, while step (5) is fully automated, it is possible to control the result by using common Coq commands (e.g. marking some definitions opaque to avoid inlining them).

5.1 Examples of Certified Processes

Pipeline. We start with a recursive variant of the example in § 2.3. The first step is to specify the global type. We write its inductive representation:

Definition pipeline := $\mu X.$ Alice → Bob :ℓ(nat).
Bob → Carol :ℓ(nat).X.

The next step is to project `pipeline` into all of its participants. There are two reasons to apply the projection at this step: (1) only well-formed protocols are projectable; and (2) we obtain the local types that will guide the implementation: the local types will need to typecheck the implemented processes. If the global type is not projectable, or the processes do not implement the resulting local types (or one of their unrollings), then we cannot guarantee anything about a Zooid implementation of any participant. We define a notation for performing the projection of all participants:

Definition pipeline$_{lt}$:= \project pipeline.

If `pipeline` is not well-formed, then `\project` will not typecheck. Otherwise, `pipeline`$_{lt}$ will be a list of pairs of participants and local types. This list will contain an entry for Alice, Bob and Carol. We get local type for Bob with:

Definition bob$_{lt}$:= \get Bob pipeline$_{lt}$.

The notation `\get` expands into a lookup in `pipeline`$_{lt}$ that requires a proof that Bob is in `pipeline`$_{lt}$. If we write `\get` p `pipeline`$_{lt}$ with some p $\notin$ `pipeline`$_{lt}$, then the command will fail to typecheck. There are now two possibilities for using bob$_{lt}$ to implement Bob: (1) providing bob$_{lt}$ as a type index; or (2) omitting bob$_{lt}$, inferring the local type, and then proving that the inferred local type is equal to bob$_{lt}$ up to unravelling. Here we use (1), but sometimes the process actually implements an unrolling of the local type. We will show examples of (2) in the next section.

Definition bob : wt_proc bob$_{lt}$
:= loop X (recv Alice (ℓ, x : nat)?
interact compute x (fun res $\Rightarrow$
send Carol (ℓ, res : nat)! jump X)).

With Zooid's `interact` command we can call the `compute` function, which is implemented in OCaml, allowing any arbitrary computation safely because the runtime hides the communication channels to prevent errors.

Finally, to do extraction to OCaml, we call `extract_proc` : `Proc` ⟶ `MP.t`. The user has options for code extraction: (1)

since `Proc` is defined inductively, use Coq's `Eval compute` to first replace all occurrences of `Proc` to `MP.t`; (2) extract the inductive representation, as well as `extract_proc`. The former may evaluate and unfold more terms than desired. To control this, we use Coq's command `Opaque` to specify any function or definition that we do not wish to be unfolded.

Ping-Pong. In the anonymous supplement, we present several implementations of the clients of a ping-pong server. The global protocol is:

```
Definition ping_pong := μX.Alice → Bob : {
        ℓ1(unit). end;  ℓ2(nat).Bob → Alice :ℓ3(nat).X}.
```

Here, Alice acts as the client for Bob, which is the ping-pong server. Alice can send zero or more *ping* messages (label ℓ_2), and finally quitting (label ℓ_1). Bob, for each ping received, will reply a *pong* message (label ℓ_3). In particular, we wish to implement a client that sends an undefined number of pings, stopping when the server replies with a natural number greater than some k. We show below the Zooid specification:

```
Definition alice : typed_proc := [proc
select Bob [skip ⇒ ℓ1, unit! end
           | otherwise ⇒ ℓ2, 0 : nat!
             loop X (recv Bob (ℓ3, x : nat)?
           select Bob [case x ≥ k ⇒ ℓ1, tt : unit! finish
                            | otherwise ⇒ ℓ2, x : nat! jump X])]]
```

We project `ping_pong` and get the expected local type for Alice: `alice`$_{lt}$. We observe that here the local type for `alice` is not syntactically equal to `alice`$_{lt}$:

```
alice_lt = μX.![Bob];{ℓ1(unit).end; ℓ2(nat).?[Bob];ℓ3(nat).X}

projT1 alice = ![Bob];{ ℓ1(unit).end;  ℓ2(nat).μX.
        ?[Bob];ℓ3(nat).![Bob];{ℓ1(unit).end;  ℓ2(nat).X.}}
```

This is not a problem since a simple proof by coinduction can show that both types unravel to the same local tree. This gains the flexibility to have processes that implement any unrolling of their local type, and the proofs are mostly simple as they follow the way the types were unrolled. See [12] for more details on how to construct gradually this client, showing how to iteratively program using Zooid.

5.2 A Certified Two Buyer Protocol

We conclude this section presenting an implementation of the *two-buyer protocol* [27], a common benchmark of MPST. This is a protocol for an online purchase service that enables customers to split the cost of an item among two participants, as long as they agree on their shares. First, buyer A queries the seller S for an item. Then, S sends the item cost first to A, then to B. Then, A sends a proposed share for the item. B then either accepts the proposal, and receives the delivery date from S, or rejects the proposal.

Figure 10 shows the protocol as a global type, the local type, B$_{1t}$, that results from the projection on B, and a possible implementation of the role of B in Zooid. Different implementations of the local type will differ in how the choice is made, but the local type will always need to be syntactically equal to the projected B$_{1t}$, due to the absence of recursion. In the implementation chosen in Figure 10, the participant B will reject any proposal where B pays more than one third of the cost of the item. This implementation is *guaranteed* to behave as B in the protocol `two_buyer`, hence deadlock-free. Our workflow preserves the ability to define and implement each participant independently: A and S could be implemented in any language, as long as they are implemented using a compatible transport to that of the OCaml implementation of `MP.t`. The code that checks the types and performs the projections is certified, as it is exactly the same code about which the properties were established.

```
Definition two_buyer :=  A → S :ItemId(nat).
  S → A :Quote(nat).S → B :Quote(nat).
  A → B :Propose(nat).B → S :{ Accept(nat).
  S → B :Date(nat).end; Reject(unit). end}
```

```
B1t := ?[S];Quote(nat).?[A];Propose(nat).![S];{Accept(nat).
?[S];Date(nat).end;Reject(unit).end}
```

```
Definition buyer_B : wt_proc B1t :=
recv S (Quote, x : nat)? recv A (Propose,  y : nat)?
  select S [case y >= divn x 3 ⇒ Accept,  y − x : nat!
                recv S (Date, d : nat)? .finish
              | otherwise ⇒ Reject, tt : unit! finish]
```

Figure 10. The Two Buyer protocol

5.3 Mechanisation Effort

The development is 7.3KLOC of Coq code, and 1.7KLOC of OCaml for the runtime (including examples). The certified code consists of 269 definitions, including functions and (co)inductive definitions, and 396 proved lemmas and theorems.

An important feature of our proof design is the *correspondence of syntactic objects and their infinite tree representation.* Coinductive trees allow us to deal smoothly with semantics and avoid bindings: such a technique applies to languages with equi-recursion, a widespread construct [20, 37, 42]. On the other hand we have kept an inductive type system for processes, so that we have finite, easy-to-inspect, structures, on which we can make computations. Our novel design takes advantage of the infinite-tree representation of syntactic objects, thus providing us with syntactic types for Zooid and coinductive representation for the proofs.

The most challenging part was working out the right definitions: the finite syntax object/infinite unrolling correspondence felt like a convoluted approach at first, but it greatly accelerated our progress afterwards.

6 Related Work and Conclusion

In the concurrency and behavioural types communities, there is growing interest in mechanisation and the use of proof assistants to validate research. As a recent example, Hinrichsen et al. [23] explore the notion of semantic typing using a concurrent separation logic as a semantic domain to build on top a language to describe binary session types. On the same vein, SteelCore [43] allows DSLs to take advantage of solid the semantic foundations provided by a proof assistant. Where their works use separation logic as a foundation, Zooid uses MPST and their coinductive expansion.

The ambition of mechanisation in behavioural types is increasing and collaborative projects that explore the space of available solutions are an important tool for the community, where they explore different representations of binders (names, de Bruijn indices/levels, nominals respectively), see [50, Discussion]. In this work we sidetrack the question by designing Zooid to use a shallow embedding of its binders (thus avoiding to need an explicit representation for variables). In our experience, this is a simple and valuable technique for the situations where it is applicable.

Other works also explore ideas on *binary* session types using proof assistants and mechanised proofs. For example, Brady [4] develops a methodology to describe safely communicating programs and implements DSLs, embedded in Idris, relying on the Idris type checker. Thiemann [47] develops an intrinsically typed semantics in Agda that provides preservation and a notion of progress for binary session types. Gay et al. [18] explore the interaction between duality and recursive types and how they take advantage of mechanisation to formalise some of their results. Tassarotti et al. [45] show the correctness (in the Coq proof assistant) of a compiler that uses an intermediate language based on a simplified version of the GV system [19] to add session types to a functional programming language. And Orchard and Yoshida [36] discuss the relation between session types and effect systems, and implement their code in the Agda proof assistant. Their formalisation concentrates on translating between effect systems and session types in a type preserving manner. Castro et al. [8] present a type preservation of binary session types [26, 52] as a case study of using their tool [9]. Furthermore, Goto et al. [22] present a session types system with session polymorphism and use Coq to prove type soundness of their system. Note that none of the above works on session types treats *multiparty session types* – they are limited to *binary* session types.

Our work on MPST uses mechanisation to both give a fresh look at trace equivalence [16] in MPST and to further explore its relation to a process calculus. At the same time our aim is to provide a bedrock for future projects dealing with the MPST theories. And crucially, this is the first work that tackles *a full syntax of asynchronous multiparty session types* that type the whole interaction, as opposed to binary session types, which only type individual channels.

Furthermore, in this work, we present not only Zooid as a certified process language, but also the methodology to design a certified language like this. Zooid's design starts with the theory, then the mechanised metatheory, and, finally, implementing a deeply embedded process language (deeply embedded in two ways: as a DSL and in the library of definitions and lemmas provided in the proof mechanisation). We propose Zooid as an alternative to writing an implementation that is proved correct post facto. There is no tension between proofs and implementation, since the proofs enable the implementation.

Regarding the choice of tool and inspiration in this work, we point out that the first objective is to mechanise trace equivalence between global and local types. For that, we took inspiration from more semantic representations of session types [20, 51]. The choice of the Coq proof assistant [46] was motivated by its stability, rich support for coinduction, and good support for the extraction of certified code. Stability is important since this is a codebase that we expect to work on and expand for future projects. The proofs take advantage of small scale reflection [21] using Ssreflect to structure our development. And given the pervasive need for greatest fixed points in MPST, we extensively use the PaCo library [30] for the proofs that depend on coinduction.

To conclude, we design and implement a certified language for concurrent processes supporting MPST. We start by mechanising the meta-theory of asynchronous MPST, and prove the soundness and completness theorems of trace semantics of global and local types. We then build Zooid, a process language on top of that. Using code extraction, we interface with OCaml code to produce running implementations of the processes specified in Zooid.

This work on mechanising MPST and Zooid is a founding stone, there are many exciting opportunities for future work. On top of our framework, we plan to explore new ideas and extensions of the theory of session types. The immediate next step is to make the proofs extensible, for example by allowing easy integration of custom merge strategies, adding advanced features such as indexed dependent session types [10], timed specifications [2, 3], or session/channel delegation [27]. Moreover, we intend to apply the work in this paper (and its extensions) to implement a certified toolchain for the Scribble protocol description language (http://www.scribble.org), also known as "the practical incarnation of multiparty session types" [25, 35]. To this aim we plan to translate from Scribble to MPST style global types, following the Featherweight Scribble formalisation [35].

Acknowledgments

We thank the PLDI reviewers for their careful reviews and suggestions. We thank Fangyi Zhou for their comments and for helping testing the artifact. The work is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1, and EP/V000462/1 and by NCSS/EPSRC VeTSS.

Additionally, and very specially, the authors wish to dedicate this work to the eternal memory of Paolo Gheri.

References

[1] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

[2] Laura Bocchi, Julien Lange, and Nobuko Yoshida. 2015. Meeting Deadlines Together. In *26th International Conference on Concurrency Theory (LIPIcs, Vol. 42)*. Schloss Dagstuhl, 283–296. https://doi.org/10.4230/LIPIcs.CONCUR.2015.283

[3] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *25th International Conference on Concurrency Theory (LNCS, Vol. 8704)*. Springer, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29

[4] Edwin Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Computer Science* 18, 3 (2017). https://doi.org/10.7494/csci.2017.18.3.1413

[5] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342. https://doi.org/10.1145/322374.322380

[6] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2017. Undecidability of Asynchronous Session Subtyping. *Inf. Comput.* 256 (2017), 300–320. https://doi.org/10.1016/j.ic.2017.07.010

[7] Simon Castellan and Nobuko Yoshida. 2019. Two Sides of the Same Coin: Session Types and Game Semantics. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2019, Lisbon, Portugal*. https://doi.org/10.1145/3290340

[8] David Castro, Francisco Ferreira, and Nobuko Yoshida. 2019. *Engineering the Meta-Theory of Session Types.* Technical Report 2019/4. Imperial College London. https://www.doc.ic.ac.uk/research/technicalreports/2019/#4

[9] David Castro, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 278–285. https://doi.org/10.1007/978-3-030-45237-7_17

[10] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290342

[11] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: a DSL for Certified Multiparty Computation (Artifact). https://doi.org/10.5281/zenodo.4581294.

[12] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: a DSL for Certified Multiparty Computation (Long Version). arXiv:2103.10269 [cs.PL]

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS, Vol. 9104)*. Springer, 146–178. https://doi.org/10.1007/978-3-319-18941-3_4

[14] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *FMSD* (2015), 1–29. https://doi.org/10.1007/s10703-014-0218-8

[15] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *ESOP 2012*. 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

[16] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Automata, Languages, and Programming*, Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

[17] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2008. On Progress for Structured Communications. In *Trustworthy Global Computing*, Gilles Barthe and Cédric Fournet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–275. https://doi.org/10.1007/978-3-540-78663-4_18

[18] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020 (EPTCS, Vol. 314)*, Stephanie Balzer and Luca Padovani (Eds.). 23–33. https://doi.org/10.4204/EPTCS.314.3

[19] Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

[20] Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. 2019. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming* 104 (2019), 127 – 173. https://doi.org/10.1016/j.jlamp.2018.12.002

[21] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

[22] Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitchar, and James Riely. 2016. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science* 26, 3 (2016), 465–509. https://doi.org/10.1017/S0960129514000231

[23] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

[24] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[25] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology*, Raja Natarajan and Adegboyega Ojo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

[26] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/BFb0053567

[27] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. of 35th Symp. on Princ. of Prog. Lang.* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328897.1328472

[28] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

[29] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (LNCS, Vol. 10202)*. 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

[30] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

[31] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi

Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (2016). https://doi.org/10.1145/2873052

[32] Dimitrios Kouzapas and Nobuko Yoshida. 2015. Globally Governed Session Semantics. *LMCS* 10 (2015). Issue 4. https://doi.org/10.2168/LMCS-10(4:20)2014

[33] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 221–232. https://doi.org/10.1145/2676726.2676964

[34] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In *27th International Conference on Compiler Construction*. ACM, 128–138. https://doi.org/10.1145/3178372.3179495

[35] Rumyana Neykova and Nobuko Yoshida. 2019. *Featherweight Scribble*. LNCS, Vol. 11665. Springer, Cham, 236–259. https://doi.org/10.1007/978-3-030-21485-2_14

[36] Dominic A. Orchard and Nobuko Yoshida. 2015. Using session types as an effect system. In *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015*. 1–13. https://doi.org/10.4204/EPTCS.203.1

[37] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press. https://dl.acm.org/doi/book/10.5555/509043

[38] Silvain Rideau and Glynn Winskel. 2011. Concurrent Strategies. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. 409–418. https://doi.org/10.1109/LICS.2011.13

[39] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[40] Alceste Scalas and Nobuko Yoshida. 2019. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Vol. 3. ACM, 30:1–30:29. https://doi.org/10.1145/3290343

[41] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 502–516. https://doi.org/10.1145/3314221.3322484

[42] Paula Severi and Mariangiola Dezani-ciancaglini. 2019. Observational Equivalence for Multiparty Sessions: Dedicated to Pawel Urzyczyn on the occasion of his 65th birthday. *Fundamenta Informaticae* 170 (10 2019), 267–305. https://doi.org/10.3233/FI-2019-1863

[43] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4, ICFP, Article 121 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409003

[44] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–413. https://doi.org/10.1007/3-540-58184-7_118

[45] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

[46] The Coq Development Team. 2020. *The Coq Proof Assistant Reference Manual v. 8.11.2*. Institut National de Recherche en Informatique et en Automatique. https://coq.inria.fr/refman/

[47] Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019* (Porto, Portugal) *(PPDP '19)*. ACM, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3354166.3354184

[48] Bernardo Toninho and Nobuko Yoshida. 2018. Interconnectability of Session Based Logical Processes. *ACM Transactions on Programming Languages and Systems* 40 (2018), 1–42. Issue 4. https://doi.org/10.1145/3242173

[49] Bernardo Toninho and Nobuko Yoshida. 2018. On Polymorphic Sessions And Functions: A Tale of Two (Fully Abstract) Encodings. In *27th European Symposium on Programming (LNCS, Vol. 10801)*. Springer, 827–855. https://doi.org/10.1007/978-3-319-89884-1_29

[50] VEST. 2020. Verification of Session Types (VEST). http://groups.inf.ed.ac.uk/abcd/VEST/. Accessed: 2020-07-06.

[51] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

[52] Nobuko Yoshida and Vasco T. Vasconcelos. 2007. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science* 171, 4 (2007), 73 – 93. https://doi.org/10.1016/j.entcs.2007.02.056 Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006).

[53] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. In *OOPSLA 2020: Conference on Object-Oriented Programming Systems, Languages and Applications (PACMPL, OOPSLA (Article 148))*. Association for Computing Machinery, New York, NY, USA, 30 pages. https://doi.org/10.1145/3428216