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

Lorenzo Gheri
Imperial College London, UK

Francisco Ferreira
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:

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

Figure 1. MPST in a nutshell

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].

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 $\text{proc}_i$ 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].
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-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:

\(^1\)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.

**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](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.
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 $\mathcal{R}$, of a type into a tree ($\S$ 3.1), and projections $\uparrow$, from global to local objects ($\S$ 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, $\text{proc}$ 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 $\text{proc}$, 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 $\text{nat}$.

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

\[ G = \text{Alice} \rightarrow \text{Bob} :!\text{nat}. \quad \text{Bob} \rightarrow \text{Carol} :!\text{nat}. \quad \text{Carol} \rightarrow \text{Alice} :!\text{nat}. \quad \text{end}. \]

The global type $G$ prescribes the full protocol, where Alice sends a message containing a nat number to Bob (with a generic label $\ell$), 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:

\[ L = ![\text{Bob} :!\text{nat}.]!\text{Carol} :!\text{nat}. \quad \text{end}, \]

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):

\[
\text{proc} = \text{send Bob } (\ell, x : \text{nat})! \quad \text{recv Carol } (\ell, y : \text{nat})? \text{ finish}
\]

Thanks to Zooid’s smart constructors, we obtain that $\text{proc}$ is well-typed with respect to the local type $L$. Additionally, the underlying metatheory certifies, by Coq proofs, that the behaviour of $\text{proc}$ 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:

\[
\begin{align*}
S ::= & \text{ end } | \text{ int } | \text{ bool } | S \times S | S + S \\
G ::= & \text{ end } | X | \mu \times G | \eta (S) | \eta (I) | \eta (L) | \eta (L)^* \\
L ::= & \text{ end } | X | \mu x L | \eta q (S) | \eta (L) | \eta (L)^* \\
\end{align*}
\]

with \( p \neq q \), \( I \neq 0 \), and \( \not{\ell}_i \neq \not{\ell}_j \) 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., \( \mu X.p \rightarrow q : \xi (\text{nat}.), G \) is a valid global type, whereas \( \mu X.X \) is not—and (2) types are closed, i.e., all variables are bound by \( \mu X. \) [12].

In the literature, it is common to adopt the equi-recursive viewpoint [37], i.e., to identify \( \mu X.G \) and \( G(\mu X.G/X) \), given that their intended behaviour is the same. Such unravelling of recursion can be performed infinitely many times, thus obtaining possibly infinite trees², 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 \( G^c \), and semantic local trees (rl_ty in Local/Tree.v), ranged over by \( L^c \), are generated coinductively by:

\[
\begin{align*}
G^c ::= & \text{ end } | p \rightarrow q : \eta (S_i), G^c | p \rightarrow q : \eta (S_i), G^c \\
L^c ::= & \text{ end } | \eta q (S) | \eta (L), L^c | \eta q (S) | \eta (L), L^c \\
\end{align*}
\]

Global and local objects share the type for a terminated protocol \text{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: \( p \rightarrow q : \eta (S_i), G^c \) describes a protocol where participant \( p \) sends to \( q \) one message with label \( \ell_i \) and a value of sort \( S_i \) as payload, for some \( i \in I \); then, depending on which \( \ell_i \) 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: \( p \rightarrow q : \eta (S_i), G^c \) represents the status where a message from \( p \) to \( q \) has yet to be sent; \( p \rightarrow q : \eta (S_i), G^c \) represents the next status: the label \( \ell_i \) has been selected, \( p \) has sent the message, with payload \( S_i \), but \( q \) has not received it yet. Local messages: send type \( !q : \eta (L), L^c \) the participant sends a message to \( q \); if the participant chooses the label \( \ell_i \), then the sent payload value must be of sort \( S_i \), and it continues as prescribed by \( L_i \). Receive type \( ?p : \eta (L), L^c \): the participant waits to receive from \( p \) a value of sort \( S_i \), for some \( i \in I \), via a message with label \( \ell_i \); then the protocol continues as prescribed by \( L_i \). The same intuition holds, mutatis mutandis, for trees.

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

We define the function \( \text{prts} \) 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 \( \text{part.of} \). 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 \( G(\mu X.G/X) \) and unravelling of local types \( L(\mu X.L/X) \) are the relations between global/local types and semantic global/local trees coinductively defined by:

\[
\begin{align*}
\text{end} \in [\text{end}^c] & & \mu X.G \rightarrow G^c \\
\text{end} \in [\text{end}^c] & & \text{end} \in [\text{end}^c] \Rightarrow [\text{end}^c] \Rightarrow [\text{end}^c] \\
\end{align*}
\]

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/1Project.v) is a partial function \( _r : g. ty \rightarrow l. ty \) 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/CPrompt.v) is a relation \( _r : g. ty \rightarrow l. ty \) 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 \( \text{part.of} \), Definition 3.4 ensures (1) that the projection of a global tree on a participant outside the protocol is \( \text{end}^c \) (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]...
Theorem 3.6 (Unravelling preserves projections) of the diagram in Figure 2. For a sketch of its proof see [16].

Example 3.5 (Projection). About rule [proj-cont], we observe that the type \( G' = Alice \to Bob : (f_1(nat).Bob \to Carol : f(nat).end, f_2(nat).Alice \to Carol : f(nat).end) \) 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 \( G = Alice \to Bob : (f_1(nat).Bob \to Carol : f(nat).end, f_2(boool).Bob \to Carol : f(nat).end) \), the projection \( G \mid Carol \) is well defined as the local type \( L = ?(Bob : f(nat).end) \). 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 following 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 \( G \mid r = L \), (b) there exists a global tree \( G^c \) such that \( G \ \sqsubseteq G^c \) and, (c) there exists a local tree \( L^c \) such that \( L \ \sqsubseteq L^c \), then \( G^c \ \sqsubseteq L^c \).

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 \( \ell \) and payload of sort \( S \) and continues on \( L^c \), and dually \( q \) receives from \( p \) the message, with same label and payload, and then continues on \( L^j \). 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 \( Q(p, q) \) (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:

\[
\text{enq} \ Q(p, q)(f, S) = Q[(p, q) \mapsto Q(p, q)@(f, S)]
\]

\[
\text{deq} \ Q(p, q) = \begin{cases} 
  Q(p, q) \mid (f, S) \neq \#s & \text{then } (f, S), Q[(p, q) \mapsto s] \\
  \text{None} & \text{else}
\end{cases}
\]

We use \# as the “cons” constructor for lists and \( @ \) as the “append” operation; \( f[x \leftrightarrow y] \) denotes the updating of a function \( f \) in \( x \) with \( y \), namely \( f[x \leftrightarrow y] x' = f x' \) for all \( x \neq x' \) and \( f[x \leftrightarrow y] x = y \). We use option types for partial functions, with None as the standard returned value where the function is undefined. In case the sequence \( Q(p, q) \) is empty \( \text{deq} \) 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 \( \epsilon \), namely \( (p, q) = \text{None} \) 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 \( \_ \{q\}_\_ : \text{rel} \ G \tyC \tenv \text{coinductively specified by:} \)

\[
\frac{\forall \ell \in I. G_{\ell} \tyC \tenv_{Q \ell} \quad Q(p,q) = \text{None}}{p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q}} \quad \text{(q-proj-send)}
\]
\[
\frac{G_{\ell_0} \tyC \tenv_{Q'} \quad \text{deq} Q'(p,q) = ((t_0(S)_0),Q)}{p \xrightarrow{\ell_0} q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q'}} \quad \text{(q-proj-recv)}
\]

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 \( G^c \) and that map each participant \( p \) to the projection of \( G^c \) onto such \( p \).

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

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 \( G^c \), notation \( G^c \tenv (E, Q) \) if it holds that: \( G^c \tenv E \) and \( G^c \tenv Q \).

Example 3.12. Let us consider the global tree: \( G^c = p \xrightarrow{\ell_0} q : \ell(S).q \rightarrow p : \ell(S),q \rightarrow p : \ell(S) \ldots \). 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: \( E \ p = \ ?c[p] ; \ell(S) ; \ ?c[q]; \ell(S) \ldots \) and \( E \ q = \ ?c[p] ; \ell(S) ; \ ?c[p]; \ell(S) \ldots \). We then define \( Q \) such that: \( Q(p,q) = \{(\ell, S)\} \) and \( Q(q,p) = \text{None} \). It is easy to verify that \( G^c \tenv (E, Q) \); observe that the only “message” enqueued in \( Q \) is \( (\ell, S) \), 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: \( !p(\ell, S) \): send \( ! \) action, from participant \( p \) to participant \( q \), of label \( \ell \) and payload type \( S \), or \( ?q(\ell, S) \): receive \( ? \) action, from participant \( p \) at participant \( q \), of label \( \ell \) and payload type \( S \). We define the subject of an action \( a \) (definition subject in Common/Actions.v), sub \( a \), as \( p \) if \( a != !p(\ell, S) \) and as \( q \) if \( a !=?q(\ell, S) \).

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 \( \xrightarrow{a} \_ : \text{rel} \ G \tyC \ G \tyC \) inductively specified by the following clauses:

\[
\frac{a ! p(\ell, S)}{p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q}} \quad \text{(g-step-send)}
\]
\[
\frac{a !=?q(\ell, S)}{p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q}} \quad \text{(g-step-recv)}
\]
\[
\frac{a \xrightarrow{g-step-str1} p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q}}{p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q}} \quad \text{(g-step-str2)}
\]

The step relation describes a labelled transition system for global trees with the following intuition: \( \text{[g-step-send]} \): sending base case: with the sending action \( !p(\ell, S) \), a message with label \( \ell \) and payload type \( S \) is sent by \( p \), but not yet received by \( q \); \( \text{[g-step-revc]} \): receiving base case: with the receiving action \( ?q(\ell, S) \), a message with label \( \ell \) and payload type \( S \), previously sent by \( p \), is now received by \( q \); in \( \text{[g-step-str1]} \), a step is allowed to be performed under a sending constructor \( p \rightarrow q \); each time that the subject of that action is different from \( p \) and from \( q \) and each continuation steps; \( \text{[g-step-str2]} \) 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 \( q \) (\( q \) has already sent the message and the label \( \ell \) has already been selected), the continuation corresponding to \( \ell \) steps and others stay as the same.

This semantics allows for some degree of non-determinism. For instance, \( p \rightarrow q : \{t_0(S)_0.G_{\ell_0} \}_{i \in I} \tenv_{Q} \) could perform a step according to both rules \( \text{[g-step-send]} \) and \( \text{[g-step-str1]} \) (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). (1-step in Local/Semantics.v) The labelled transition relation for environments (local reduction or local step for short) is, for each

\footnote{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.}
Theorem 3.16 (Step Soundness). (Theorem Project_step in TraceEquiv.v) If $G' \xrightarrow{a} G''$ and $G' \parallel (E, Q)$, there exist $E'$ and $Q'$ such that $G'' \parallel (E', Q')$ and $(E, Q) \xrightarrow{a} (E', Q')$.

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 tree is admissible for a global tree if the coinductive relation $tr^g \parallel -$ (definition g_lts in Global/Semantics.v) holds:

$$tr^g [\parallel ] \xrightarrow{\text{end}^g}$$

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 $tr^l \parallel -$ (definition 1_lts in Local/Semantics.v) holds:

$$\forall p, E, Q \parallel \text{None} \xrightarrow{\text{tr}^l[\parallel]} \text{tr}^l[\parallel] (E, Q) \xrightarrow{t (E', Q')}$$

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

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 $G' \parallel (E, Q)$, then $tr^g \parallel (E, Q)$ if and only if $tr^l \parallel (E', Q')$.

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 diagram in Figure 2. In the next section we specify a language for communicating systems inside Coq and 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

Footnote 1: For traces, we use the same notation as for lists, however we bear in mind that this definition is coinductive: it generates possibly infinite streams.

Footnote 2: About non-determinism in our semantics, see [12].
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.

\[
\begin{align*}
\text{e ::= } & \quad\text{proc } | \quad e + e \quad| \quad \text{if } e \quad\text{then } e \quad\text{else } e \\
& \quad\text{fun } x \quad\text{=} \quad e \quad| \quad \ldots \quad n \in \mathbb{N} \quad\quad\ell \in \mathbb{N} \\
\text{proc } \in \text{Proc} \quad\text{=} \quad & \quad\text{finish} \quad| \quad\text{jump } X \quad| \quad\text{loop } X \quad\{ e \} \\
& \quad\text{recv } p \quad\{ t_i.e_i \}_{i \in I} \quad| \quad\text{send } p \quad\{ t_i.e_i \}_{i \in I} \\
& \quad\text{act}_r(x.e) \quad| \quad\text{act}_r(x.e) \quad| \quad\text{act}_r(x.e) \quad| \quad\text{act}_r(x.e) \\
& \quad\text{read }\text{act}_r(x.e) \quad| \quad\text{write }\text{act}_r(x.e) \quad| \quad\text{interact }\text{act}_r(x.e) \quad| \quad\text{interact }\text{act}_r(x.e)
\end{align*}
\]

**Figure 5. Process Typing System**

The constructs of Proc mirror those of local types: finish is the ended process; jump X is a jump to recursive variable X; loop X { e} is a recursive process, built by expression e, that introduces a new recursion variable X; recv p { { t_i.e_i} } is the process receiving from p a message with label t_i, a value x, and continues as (e_1 x); and send p { t_i.e_i}. e_2 is the sending process with label t 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:

\[
\text{proc}_q = \quad\text{loop } X \{ \text{recv } p \{ t_i.\text{fun } x \text{ = } \text{send } p \{ t_i.x + m \} \\
\text{jump } X \{ t_i.\text{fun } x \text{ = } \text{finish} \}
\]

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:

\[
\text{proc}_r = \quad\text{fun } x \quad\text{=} \quad\text{if } x > n \text{ then send } q \{ t_0.t_t \}.\text{finish} \\
\text{else }\text{send } q \{ t_0.x \}.\text{jump } X
\]

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: act_r, act_w, and act_c for reading, writing or interacting with the environment. act_r is a function that takes a unit and returns a value of payload type (i.e.: a coqTy T for some type T). 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 act_c 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 \( \Gamma \vdash e : L \) in Figure 5, as an inductive predicate in Coq (definition of let 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 \( [S_j] \) (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 t has a send type, if t 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 \( x : [S_1] \). 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 \( \Gamma \vdash \ell_1 : L \) and \( \Gamma \vdash \ell_2 : L \) then \( \Gamma \vdash \ell_3 \) if \( e \) then \( \ell_1 \) else \( \ell_2 : L \) 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 (ranked over by \( Z \)) are dependent pairs of a \texttt{proc}, and a proof that it is well-typed with respect to a given local type \( L \).

**Definition** \( \text{wt_proc} L := \{ P : \text{Proc} | \text{of_\_L} P \text{ } \} \).

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 \texttt{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 \texttt{proc} is well-typed with respect to some local type \( L \), second component of the dependent pair.

The simplest example is the \texttt{finish} term for inactive processes of type \( \text{1_end} \). Coq infers most parameters.

**Notation** \( \text{finish} := \text{wt_end} \).

\[ \text{Definition} \text{wt_end} : \text{wt_proc} \text{1_end} := \text{exist } t \_ \text{.t.Finish} \]

On the other hand, the notation \( \text{\texttt{\_send}} \) is defined in the same way, but the definition of the dependent pair requires a simple proof (i.e.: \( \text{wt_send} \)). The \texttt{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** \( \text{wt_send} p l T (pl : \text{coq_ty} T ) L (P : \text{wt_proc} L) := \text{wt_proc} (1_{\text{msg}} l_{\text{send}} p P \text{ } :: (l, (T, L))) := \text{exist } t \_ \text{.t.Send p pl (of_{wt_proc} P)} \)

\[ \text{find_cont_sing L T}) \). \]

**Notation** \( \text{\texttt{\_send}} := \text{wt_send} \).

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) \texttt{if then else} is a Zooid construct since it needs to carry the proof that the underlying proc is well-typed; (b) \texttt{branch} and \texttt{select} must take a list of alternatives (\( Z^b \) and \( Z^s \) respectively), and \texttt{send}/receive are defined as \texttt{branch/select} with a singleton alternative. The alternatives for \texttt{branch}, \( Z^b \), are pairs of labels and continuations. The alternatives for \texttt{select}, \( Z^s \) are:

1. \texttt{case} \( e_1 \Rightarrow \ell, e_2 : S! Z \), specifies to send \( \ell \) and \( e_2 : [S] \)

**Definition 4.3** (Zooid syntax). \( Z^b \Leftarrow e \Rightarrow \ell, e_2 : S! Z \Rightarrow \ell \Rightarrow e \Rightarrow e_2 : [S] \)

and then continue as \( Z \), when \( e_1 \) evaluates to true:

2. \text{otherwise} \( \Rightarrow \ell, e_2 : S! Z \Rightarrow \ell \Rightarrow e \Rightarrow e_2 : [S] \)

3. \text{skip} \( \Rightarrow \ell, e : S! Z \Rightarrow \ell \Rightarrow e \Rightarrow e \Rightarrow e \Rightarrow e_2 : [S] \)

\[ \text{Definition} \text{wt_end} : \text{wt_proc} \text{1_end} := \text{exist } t \_ \text{.t.Finish} \]

As follows:

**Definition** \( \text{wt_end} : \text{wt_proc} \text{1_end} := \text{exist } t \_ \text{.t.Finish} \)

\[ \text{Definition} \text{wt_send} p l T (pl : \text{coq_ty} T ) L (P : \text{wt_proc} L) := \text{wt_proc} (1_{\text{msg}} l_{\text{send}} p P \text{ } :: (l, (T, L))) := \text{exist } t \_ \text{.t.Send p pl (of_{wt_proc} P)} \)

\[ \text{find_cont_sing L T}) \). \]

**Notation** \( \text{\texttt{\_send}} := \text{wt_send} \).

Figure 6. Zooid Syntax

and then continue as \( Z \), when \( e_1 \) evaluates to true:

2. \text{otherwise} \( \Rightarrow \ell, e_2 : S! Z \Rightarrow \ell \Rightarrow e \Rightarrow e_2 : [S] \)

3. \text{skip} \( \Rightarrow \ell, e : S! Z \Rightarrow \ell \Rightarrow e \Rightarrow e \Rightarrow e \Rightarrow e_2 : [S] \)

\[ \text{Definition} \text{wt_end} : \text{wt_proc} \text{1_end} := \text{exist } t \_ \text{.t.Finish} \]

\[ \text{Definition} \text{wt_send} p l T (pl : \text{coq_ty} T ) L (P : \text{wt_proc} L) := \text{wt_proc} (1_{\text{msg}} l_{\text{send}} p P \text{ } :: (l, (T, L))) := \text{exist } t \_ \text{.t.Send p pl (of_{wt_proc} P)} \]

\[ \text{find_cont_sing L T}) \). \]

**Notation** \( \text{\texttt{\_send}} := \text{wt_send} \).

The steps of the LTS are: \( [\text{\[p-step-send\]}] \) states that a send process transitions to the continuation \( e_2 \) with the action that sends a label \( \ell \) and value \( e_1 \); \( [\text{\[p-step-recv\]}] \) states that a receive process transitions to \( (e_2 \bar{e}) \) with the receive action from participant \( p \); and \( [\text{\[p-step-loop\]}] \) unfolds recursion once to perform a step on a recursive process.

We prove the type preservation for \( \tau_{\text{LTS}} \). 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 \( a = \text{\texttt{\_pq}((\ell), e_1) \} \Rightarrow \ell \Rightarrow e_2 = \text{\texttt{\_pq}((\ell), e_1) \} \Rightarrow \ell \Rightarrow e_2 \Rightarrow e_2 : [S] \), then \( |a| = \text{\texttt{\_pq}((\ell), S) \}

\[ \text{Definition 4.4} \text{ (LTS for processes). The LTS for processes is, for each action } a, \text{ defined as:} \]

\[ [\text{\[p-step-send\]}] \]

\[ [\text{\[p-step-recv\]}] \]

\[ [\text{\[p-step-loop\]}] \]

The steps of the LTS are: \( [\text{\[p-step-send\]}] \) states that a send process transitions to the continuation \( e_2 \) with the action that sends a label \( \ell \) and value \( e_1 \); \( [\text{\[p-step-recv\]}] \) states that a receive process transitions to \( (e_2 \bar{e}) \) with the receive action from participant \( p \); and \( [\text{\[p-step-loop\]}] \) unfolds recursion once to perform a step on a recursive process.

We prove the type preservation for \( \tau_{\text{LTS}} \). 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 \( a = \text{\texttt{\_pq}((\ell), e_1) \} \Rightarrow \ell \Rightarrow e_2 \Rightarrow e_2 : [S] \), then \( |a| = \text{\texttt{\_pq}((\ell), S) \}

\[ \text{\[p-step-send\]} \]

\[ \text{\[p-step-recv\]} \]

\[ \text{\[p-step-loop\]} \]
(Process and global type traces)

Theorem 4.5 (Type preservation). (Theorem preservation in the file Proc.v) If $\Gamma \vdash \tau \triangleright e : L$ and $e \rightarrow e'$, then there exists $L'$ such that $L \rightarrow L'$, and $\Gamma \vdash e' \triangleright e$.

We write $\text{tr}^p_t e$ to express that a trace $t$ is admissible by process $e$. The formal definition goes analogously to Definition 3.20 for $\text{tr}^1_{- -}$; 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 $\text{subtrace}$ 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 $t_1 \leq_p t_2$ as the greatest relation satisfying:

\[
\begin{align*}
\text{subj } a \neq p & \quad t_1 \leq_p t_2 \\
\text{subj } a = p & \quad t_1 \leq_p t_2 \\
(a \# t_1) \leq_p (a \# t_2) \\
\| \leq p \| 
\end{align*}
\]

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.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 $G^c \vdash (E, e)$ and $\Gamma \vdash e \triangleright L$ such that $L \not\rightarrow (E, p)$. Then, for all traces $t_p$ such that $\text{tr}^p_t e$ there exists a trace $t$ such that $\text{tr}^p_\tau G^c$, and $t_p \leq t$.

Figure 7 presents the meaning of the above theorem graphically. Any trace $t_p = a_2 \# a_3 \# \ldots$ of a process $\text{proc}$ is contained within a larger system trace $t = a_1 \# a_2 \# a_3 \# \ldots$ of $G^c$, given that $\text{proc}$ behaves as some participant $p$ in $G^c$. 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 $G^c$, then the behaviour of $e$ is that of $p$ in $G^c$.

4.4 Extraction

Terms of type $\text{Proc}$, 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 $\text{proc}$ terms by a suitable OCaml handler. Figure 8 shows the declaration of a module for that purpose.

Module ProcessMonad.

Parameter $t : \text{Type} \rightarrow \text{Type}$.

Parameter $\text{bind} : \forall T1 T2, t T1 \rightarrow (T1 \rightarrow t T2) \rightarrow t T2$.

Parameter $\text{pure} : \forall T1, T1 \rightarrow t T1$.

Parameter $\text{send} : \forall T1, T1 \rightarrow \text{lbl} \rightarrow T \rightarrow t \text{unit}$.

Parameter $\text{recv} : \forall T1, T \rightarrow \text{lbl} \rightarrow t \text{unit}$.

Parameter $\text{recv_one} : \forall T1, T \rightarrow t \text{unit}$.

Parameter $\text{run} : \forall A, t A \rightarrow A$.

End ProcessMonad.

Figure 8. The Process Monad.

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 $\text{extract_proc}$ from Proc.v. [12] shows the function in its entirety.

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 which is deferred to the implementation of the monad. The runtime implements the monad relying on the module provided by OCaml’s Lwt library\footnote{https://ocsigen.org/lwt/5.2.0/manual/manual}, 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\footnote{https://ocaml.org/releases/4.11/htmlman/libref/Marshal.html}. 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...
module type PROCESS_FUNCTOR =
  functor (MP : ProcessMonad) -> sig
  module PM : sig
    type 'x MP.t = 'x MP.t
    val run : 'a1 t -> 'a1 t
    val send : role -> lb1 -> 'a1 -> unit t
    val recv :
      role -> (lb1 -> 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
  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 := μX. Alice → Bob : f(nat).
  Bob → Carol : f(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 pipeline₁ := \project pipeline.

If pipeline is not well-formed, then \project will not typecheck. Otherwise, pipeline₁ 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 bob₁ := \get Bob pipeline₁.

The notation \get expands into a lookup in pipeline₁ that requires a proof that Bob is in pipeline₁. If we write \get p pipeline₁ with some p \notin pipeline₁, then the command will fail to typecheck. There are now two possibilities for using bob₁ to implement Bob: (1) providing bob₁ as a type index; or (2) omitting bob₁, inferring the local type, and then proving that the inferred local type is equal to bob₁ up to unrolling. 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 bob₁ := loop X (recv Alice (f, x : nat)?
  interact compute x (fun res ⇒
    send Carol (f, 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.

### 5.2 A Certified Two Buyer Protocol

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

``` Ocaml
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 ℓ₂), and finally quitting (label ℓ₃). Bob, for each ping received, will reply a pong message (label ℓ₃). 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:

``` Ocaml
Definition alice : typed_proc := [proc
select Bob [skip ⇒ ℓ₁, unit! end
| otherwise ⇒ ℓ₂, 0 : nat!
  loop X (recv Bob (ℓ₂, 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 ℓ₁. We observe that here the local type for Alice is not syntactically equal to alice ℓ₁:

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

This is not a problem since a simple proof by coinduction can show that both types unroll 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.

### 5.3 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 commu-
nity, where they explore different representations of bind-
ers (names, de Bruijn indices/levels, nominals respectively),
see [50, Discussion]. In this work we sidetrack the ques-
tion 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 com-
municating programs and implements DSLs, embedded in
Idris, relying on the Idris type checker. Thiemann [47] de-
velops 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 du-
ality 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 trans-
lating between effect systems and session types in a type
preserving manner. Castro et al. [8] present a type preserva-
tion 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, fi-
nally, 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 mechan-
isation). 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 lan-
guage for concurrent processes supporting MPST. We start
by mechanising the meta-theory of asynchronous MPST,
and prove the soundness and completeness 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 implemen-
tations of the processes specified in Zooid.

This work on mechanising MPST and Zooid is a found-
ing 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 imme-
diate 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 ses-
tion types [10], timed specifications [2, 3], or session/chann-
el 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 in-
carnation 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 suppor-
ted by EPSRC EP/T065544/1, 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 ded-
icate this work to the eternal memory of Paolo Gheri.
References


