Formally Verified Liveness with Multiparty Session Types in Rocq

Omer Keskin
University of Edinburgh, UK

Nobuko Yoshida
University of Oxford, UK

Rob van Glabbeek
University of Edinburgh, UK
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Abstract

Multiparty session types (MPST) offer a framework for the description of communication-based protocols involving multiple participants. In the *top-down* approach to MPST, the communication pattern of the session is described using a *global type*. Then the global type is *projected* on to a *local type* for each participant, and the individual processes making up the session are type-checked against these projections. Typed sessions possess certain desirable properties such as *safety*, *deadlock-freedom* and *liveness*.

In this work, we present the first mechanised proof of liveness for synchronous multiparty session types in the Rocq Proof Assistant. Building on recent work, we represent global and local types as coinductive trees using the Paco library. We use a coinductively defined *subtyping* relation on local types together with another coinductively defined *plain-merge* projection relation relating local and global types. We then *associate* collections of local types, or *local type environments*, with global types using these projection and subtyping relations, and prove an *operational correspondence* between a local type environment and its associated global type. We utilise this association relation to prove the safety and liveness of associated local type environments and, consequently, the multiparty sessions typed by these environments.

Besides clarifying the often informal proofs found in the MPST literature, our Rocq mechanisation also enables the certification of liveness properties of communication protocols. Our contribution amounts to around 14K lines of Rocq code, available at https://github.com/omerskeskin/mpstlive.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Program verification; Theory of computation → Operational semantics

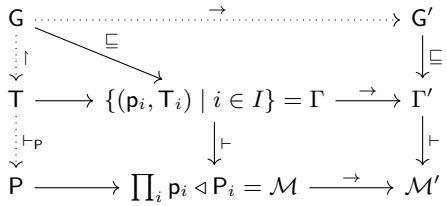
Keywords and phrases Multiparty Session Types, Liveness, Safety, Fairness, Deadlock-Freedom, Endpoint Projection, Subtyping, Rocq, Coinduction, Property Verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.4

Supplementary Material *Software (Source)*: https://github.com/omerskeskin/mpstlive

Funding *Nobuko Yoshida*: EPSRC EP/T006544/2 , EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, EU Horizon 101093006 and UKRI 10066667, Advanced Research and Invention Agency (ARIA), EP/Z533749/1 and a grant from the Simons Foundation.
Rob van Glabbeek: Supported by Royal Society Wolfson Fellowship RSWF\R1\221008

Acknowledgements We deeply thank Burak Ekici for his collaborations, guidance and detailed feedback. We also thank the ITP'26 reviewers for the detailed feedback.



T refers to a local type, G a global type, P a process. ↾ denotes projection and → denotes reduction. The dotted lines correspond to relations inherited from [16] while the solid lines denote relations that are new, or rewritten, in this paper.

Figure 1 Design overview

1 Introduction

Multiparty session types [24] provide a type discipline for the correct-by-construction specification of message-passing protocols. Desirable protocol properties guaranteed by session types include *communication safety* (the labels and types of senders' payloads cohere with the capabilities of the receivers), *deadlock-freedom* (also called *progress* or *non-stuck property* [16]) (it is possible for the session to progress so long as it has at least one active participant), and *liveness* (also called *lock-freedom* [21] or *starvation-freedom* [9]) (if a process is waiting to send or receive then a communication involving it eventually happens).

There exists two common methodologies for multiparty session types. In the *bottom-up* approach, the individual processes making up the session are typed using a collection of *participants* and *local types*, that is, a *local type environment*, and the properties of the session are examined by model-checking this local type environment. Contrastingly, in the *top-down* approach sessions are typed by a *global type* that is related to the processes using endpoint *projections* and *subtyping*. The structure of the global type ensures that the desired properties are satisfied by the session. These two approaches have their advantages and disadvantages: the bottom-up approach is generally able to type more sessions, while type-checking and type-inference in the top-down approach tend to be more efficient than model-checking the bottom-up system [46].

In this work, we present the Rocq [5] formalisation of a synchronous MPST system [20, 48] that ensures the aforementioned properties for typed sessions. Our system types ($\vdash$) *sessions* with *local type environments*. Our local types closely mimic the structure of the processes; however they are not enough to guarantee the properties we desire. To that end, we target typability by a certain set of well-behaved local type environments that are *associated* with a global type.

The *association* ($\sqsubseteq$) [49, 38] relation relates local type environments and global types using (coinductive plain) projection [45] and *subtyping*. It ensures *operational correspondence* between the labelled transition system (LTS) semantics we define for local type environments and global types. By exploiting the structure of the global type and relating the transition semantics for sessions, local type environments and global types, we show that if an associated local type environment Γ types a session $\mathcal{M}$, then $\mathcal{M}$ is guaranteed to possess safety (Theorem 6.5), deadlock-freedom (Theorem 6.6) and liveness (Theorem 6.9).

To our best knowledge, this work presents the first mechanisation of liveness for multiparty session types in a proof assistant. It is also the first work which mechanises a MPST system based on the association relation, and proves its correctness by formalising properties of local type environments Γ (such as safety and deadlock-freedom).

Our Rocq implementation builds upon the recent formalisation of subject reduction for MPST by Ekici et al. [16], which itself is based on [20]. The methodology in [16] takes an equirecursive approach where an inductive syntactic global or local type is identified with

the coinductive tree obtained by fully unfolding the recursion. It then defines a coinductive projection relation between global and local type trees, the LTS semantics for global type trees, and typing rules for the session calculus outlined in [20]. We extensively use these definitions and the lemmas concerning them, but we depart from and extend [16] in numerous ways by introducing local type environments, their correspondence with global types and a new typing relation. Our addition to the code amounts to around 14K lines of Rocq, which we link throughout with the symbol .

As with [16], our implementation heavily uses the parameterised coinduction technique of the Paco [25] library. Namely, our liveness property is defined using possibly infinite *execution traces* which we represent as coinductive streams. The relevant predicates on these traces, such as fairness, are then defined as mixed inductive-coinductive predicates using linear temporal logic (LTL) [39]. In Paco coinductive predicates are defined as the greatest fixpoints of inductive relations parameterised by an auxiliary relation representing the knowledge accumulated during the course of the proof. Statements on these predicates can then be proved by incrementally adding to this knowledge over the course of the proof, sidestepping the slow and non-compositional syntactic guardedness checks needed for the soundness of Rocq's native `cofix` tactic. Our approach using LTL and parameterised coinduction results in compositional and clear proofs.

Outline. In Section 2 we define our session calculus and its LTS semantics. In Section 3 we recapitulate the definitions of local and global type trees, and the subtyping and projection relations on them, from [16]. In Section 4 we give LTS semantics to local type environments and global types, and detail the association relation between them. In Section 5 we define safety and liveness for local type environments, and prove that they hold for contexts associated with a global type tree. In Section 6 we give the typing rules for our session calculus, and prove the desired properties of typable sessions.

2 Synchronous Multiparty Session Calculus

We introduce a simple synchronous multiparty session calculus [48] that our type system will be used on.

2.1 Processes and Multiparty Sessions

▶ **Definition 2.1** (Expressions and Processes). *We define processes as follows:*

$$\mathsf{P} ::= \mathsf{p}!\ell(\mathsf{e}).\mathsf{P} \mid \textstyle\sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{P}_i \mid \mathsf{if\ e\ then\ P\ else\ P} \mid \mu \boldsymbol{X}.\mathsf{P} \mid \boldsymbol{X} \mid \boldsymbol{0}$$

where e *is an expression, which is either a variable, a value such as* `true`, 0 *or* -3*, or a term built from expressions with operators such as* `succ`, `neg`, $\neg$ *and non-deterministic choice* $\oplus$.

Process $\mathsf{p}!\ell(\mathsf{e}).\mathsf{P}$ sends the value of expression e with label ℓ to participant p, and continues with process P. Process $\sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{P}_i$ receives a value from p with any label ℓ_i where $i{\in}I$, with I being a finite non-empty index set, binding the result to x_i and continuing with P_i, depending on which ℓ_i the value was received with. $\mathbf{X}$ is a recursion variable, $\mu\mathbf{X}.\mathsf{P}$ is a recursive process, $\mathsf{if\ e\ then\ P\ else\ P}$ is a conditional and $\mathbf{0}$ is a terminated process. We always assume that recursion is guarded.

Processes can be composed in parallel into sessions.

▶ **Definition 2.2** (Multiparty Sessions). *Multiparty sessions are defined as follows.*

$$\mathcal{M} ::= \mathsf{p} \triangleleft \mathsf{P} \quad \mid \quad (\mathcal{M} \mid \mathcal{M}) \quad \mid \quad \mathcal{O}$$

$\mathsf{p} \triangleleft \mathsf{P}$ denotes that participant p is running the process P, $|$ indicates parallel composition. We write $\prod_{i \in I} \mathsf{p}_i \triangleleft \mathsf{P}_i$ to denote the session formed by p_i running P_i in parallel for all $i \in I$. $\mathcal{O}$ is an empty session with no participants, that is, the unit of parallel composition. In Rocq processes and sessions are defined with the inductive types `process` and `session`. We follow [16] and represent the continuations in a receiving process using a `list` of `option` types. In a continuation `gcs : list (option process)`, index `k` (using zero-indexing) being equal to `Some P_k` means that $\ell_k(x).\mathsf{P}_k$ is available in the continuation. Similarly index `k` being equal to `None` or being out of bounds of the list means that the message label ℓ_k is not present in the continuation. The function `onth` formalises this convention in Rocq. This representation is also used for local (Definition 3.1) and global (Definition 3.3) type trees.

```
Inductive process : Type ≜
  | p_send : part → label → expr → process → process
  | p_recv : part → list(option process) → process
  | p_ite : expr → process → process → process
  | p_rec : process → process
  | p_var : nat → process
  | p_inact : process.
Inductive session: Type ≜
  | s_ind : part   → process → session
  | s_par : session → session → session
  | s_zero : session.
```

2.2 Structural Congruence and Operational Semantics

We define the operational semantics for sessions by the means of a labelled transition system. We omit the semantics for expressions as they are standard and are found in [20].

We start by defining a structural congruence relation $\equiv$ on sessions which expresses the commutativity, associativity and unit of the parallel composition operator. For reductions, we use labelled *reactive* semantics [21, 7] which doesn't contain explicit silent τ actions for internal reductions (that is, evaluation of if-expressions and unfolding of recursion) while still considering β-reductions up to those internal reductions by using an unfolding relation. This stands in contrast to the more standard semantics used in [16, 20, 21]. For the advantages of our approach see Remark 6.4.

In Table 1, $\mathcal{M} \Rrightarrow \mathcal{N}$ means that $\mathcal{M}$ can transition to $\mathcal{N}$ through some internal actions, that is, a reduction that doesn't involve a communication. We say that $\mathcal{M}$ *unfolds* to $\mathcal{N}$. Then [R-COMM] captures communications between processes, and [R-UNFOLD] lets us consider reductions up to unfoldings.

In Rocq, the unfolding is captured by the predicate `unfoldP : session → session → Prop` and `betaP_lbl M lambda M'` denotes $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$. We write $\mathcal{M} \to \mathcal{M}'$ if $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$ for some λ, which is written `betaP M M'` in Rocq. We write $\to^*$ to denote the reflexive transitive closure of $\to$, which is called `betaRtc` in Rocq.

3 The Type System

We briefly recap the core definitions of local and global type trees, subtyping and projection from [20]. We take an equirecursive approach and work directly on the possibly infinite local and global type trees obtained by unfolding the recursion in guarded syntactic types; details of this approach can be found in [16] and hence are omitted here.

$$[\text{R-COMM}]\quad \frac{j \in I \quad e \downarrow v}{\mathsf{p} \triangleleft \sum_{i \in I} \mathsf{q}?\ell_i(x_i).\mathsf{P}_i \mid \mathsf{q} \triangleleft \mathsf{p}!\ell_j(\mathsf{e}).\mathsf{Q} \mid \mathcal{N} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_j} \mathsf{p} \triangleleft \mathsf{P}_j[v/x_j] \mid \mathsf{q} \triangleleft \mathsf{Q} \mid \mathcal{N}}$$

$$[\text{UNF-TRANS}]\quad \frac{\mathcal{M} \Rrightarrow \mathcal{M}' \quad \mathcal{M}' \Rrightarrow \mathcal{N}}{\mathcal{M} \Rrightarrow \mathcal{N}}$$

$$[\text{R-UNFOLD}]\quad \frac{\mathcal{M} \Rrightarrow \mathcal{M}' \quad \mathcal{M}' \xrightarrow{\lambda} \mathcal{N}' \quad \mathcal{N}' \Rrightarrow \mathcal{N}}{\mathcal{M} \xrightarrow{\lambda} \mathcal{N}}$$

$$[\text{UNF-STRUCT}]\quad \frac{\mathcal{M} \equiv \mathcal{N}}{\mathcal{M} \Rrightarrow \mathcal{N}}$$

$$[\text{UNF-CONDT}]\quad \frac{e \downarrow \text{true}}{\mathsf{p} \triangleleft \text{ if } e \text{ then } \mathsf{P} \text{ else } \mathsf{Q} \mid \mathcal{N} \Rrightarrow \mathsf{p} \triangleleft \mathsf{P} \mid \mathcal{N}}$$

$$[\text{UNF-REC}]\quad \mathsf{p} \triangleleft \mu\mathbf{X}.\mathsf{P} \mid \mathcal{N} \Rrightarrow \mathsf{p} \triangleleft \mathsf{P}[\mu\mathbf{X}.\mathsf{P}/\mathbf{X}] \mid \mathcal{N}$$

$$[\text{UNF-CONDF}]\quad \frac{e \downarrow \text{false}}{\mathsf{p} \triangleleft \text{ if } e \text{ then } \mathsf{P} \text{ else } \mathsf{Q} \mid \mathcal{N} \Rrightarrow \mathsf{p} \triangleleft \mathsf{Q} \mid \mathcal{N}}$$

$$[\text{SC-SYM}]\quad \mathcal{M} \mid \mathcal{N} \equiv \mathcal{N} \mid \mathcal{M} \qquad [\text{SC-ASSOC}]\quad (\mathcal{L} \mid \mathcal{M}) \mid \mathcal{N} \equiv \mathcal{L} \mid (\mathcal{M} \mid \mathcal{N}) \qquad [\text{SC-O}]\quad \mathcal{M} \mid \mathcal{O} \equiv \mathcal{M}$$

Table 1 Structural Congruence, Unfolding and Reductions of Sessions

3.1 Local Type Trees

We start by defining the sorts that will be used to type expressions, and local types that will be used to type single processes.

▶ **Definition 3.1** (Sorts and Local Type Trees). *We define three sorts:* `int`, `bool` *and* `nat`. *Local type trees are then defined coinductively with the following syntax* :

$$\mathsf{T} ::= \quad \mathtt{end} \;\mid\; \mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \;\mid\; \mathsf{p}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$$

In the above definition, `end` represents a role that has finished communicating. $\mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ denotes a role that may, from any $i \in I$, with I being a non-empty finite indexing set, receive a value of sort S_i with message label ℓ_i and continue with T_i. Similarly, $\mathsf{p}\oplus\{\ell_i(S_i).\mathsf{T}\}_{i\in I}$ represents a role that may choose to send a value of sort S_i with message label ℓ_i and continue with T_i for any $i \in I$. Local type trees are expressed in Rocq with the following:

```
Inductive sort: Type ≜ | sbool: sort | sint : sort | snat : sort.
CoInductive ltt: Type ≜
  | ltt_end : ltt
  | ltt_recv: part → list (option(sort*ltt)) → ltt
  | ltt_send: part → list (option(sort*ltt)) → ltt.
```

As with processes, we represent the continuations using a `list` of `option` types. Index k of the continuation being a `Some` value means that label ℓ_k exists in the continuation.

3.2 Subtyping

We define the subsorting relation on sorts and the process-oriented [19] subtyping relation on local type trees.

▶ **Definition 3.2** (Subsorting and Subtyping). *Subsorting* $\leq$ *is the least reflexive binary relation that satisfies* `nat` $\leq$ `int`. *Subtyping* $\leqslant$ *is the largest relation between local type trees*

coinductively defined by the following rules:

$$\frac{}{\mathtt{end} \leqslant \mathtt{end}}\ [\text{SUB-END}] \qquad \frac{\forall i \in I: \quad S'_i \leq S_i \qquad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I\cup J} \leqslant \mathsf{p}\&\{\ell_i(S'_i).\mathsf{T}'_i\}_{i\in I}}\ [\text{SUB-IN}]$$

$$\frac{\forall i \in I: \quad S_i \leq S'_i \qquad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\mathsf{p}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \leqslant \mathsf{p}\oplus\{\ell_i(S'_i).\mathsf{T}'_i\}_{i\in I\cup J}}\ [\text{SUB-OUT}]$$

Intuitively, $\mathsf{T}_1 \leqslant \mathsf{T}_2$ means that a role of type T_1 can be supplied anywhere a role of type T_2 is needed. [SUB-IN] captures the fact that we can supply a role that is able to receive more labels than specified, and [SUB-OUT] captures that we can supply a role that has fewer labels available to send. Note the contravariance of the sorts in [SUB-IN]; if the supertype demands the ability to receive an `nat` then the subtype can receive `nat` or `int`.

In Rocq, the subtyping relation `subtypeC : ltt → ltt → Prop` is expressed as a greatest fixpoint using the `Paco` library [25]; for details we refer to [20].

3.3 Global Type Trees

We now define global types which give a bird's eye view of the whole protocol. As before, we work directly on infinite trees and omit the details which can be found in [16].

▶ **Definition 3.3** (Global type trees). *We define global type trees coinductively as follows:*

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

```
CoInductive gtt: Type ≜
  | gtt_end    : gtt
  | gtt_send   : part → part → list
    (option (sort*gtt)) → gtt.
```

`end` denotes a protocol that has ended, $\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ denotes a protocol where for any $i \in I$, with I being a non-empty finite index set, participant p may send a value of sort S_i to another participant q via message label ℓ_i, after which the protocol continues as G_i. We further define a function $\mathtt{pt}(\mathsf{G})$ that denotes the participants of the global type G as the least solution[1] to the following equations:

$$\mathtt{pt}(\mathtt{end}) = \emptyset \qquad \mathtt{pt}(\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}) = \{\mathsf{p},\mathsf{q}\} \cup \bigcup_{i\in I} \mathtt{pt}(\mathsf{G}_i)$$

In Rocq `pt` is captured with the predicate `isgPartsC : part → gtt → Prop`, where `isgPartsC p G` denotes $\mathsf{p} \in \mathtt{pt}(\mathsf{G})$.

3.4 Projection

We now define coinductive projections with plain merging (see [46] for a survey of other notions of merge).

▶ **Definition 3.4** (Projection). *The projection of a global type tree onto a participant* r *is the largest relation* $\upharpoonright_{\mathsf{r}}$ *between global type trees and local type trees such that, whenever* $\mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{T}$*:*

- $\mathsf{r} \notin \mathtt{pt}\{\mathsf{G}\}$ *implies* $\mathsf{T} = \mathtt{end}$*;* [PROJ-END]
- $\mathsf{G} = \mathsf{p} \to \mathsf{r} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ *implies* $\mathsf{T} = \mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ *and* $\forall i \in I, \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}_i$ [PROJ-IN]
- $\mathsf{G} = \mathsf{r} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ *implies* $\mathsf{T} = \mathsf{q}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ *and* $\forall i \in I, \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}_i$ [PROJ-OUT]
- $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ *and* $\mathsf{r} \notin \{\mathsf{p},\mathsf{q}\}$ *implies that* $\forall i \in I, \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}$ [PROJ-CONT]

[1] This is a simplified presentation of the definition of `pt` in Rocq; for technical details see [16].

Informally, the projection of a global type tree G onto a participant r extracts a role for participant r from the protocol whose bird's-eye view is given by G. [Proj-End] expresses that if r is not a participant of G then r does nothing in the protocol. [Proj-In] and [Proj-Out] handle the cases where r is involved in a communication in the root of G. [Proj-Cont] says that, if r is not involved in the root communication of G and all continuations of G project on to the same type, then G also projects on to that type. In Rocq, projection is defined as a Paco greatest fixpoint with the relation `projectionC : gtt → part → ltt → Prop`.

Using a result from [16], we can regard projection as a partial function. We write $\mathsf{G} \upharpoonright \mathsf{r} = \mathsf{T}$ when $\mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{T}$. Furthermore we will frequently be making assertions about subtypes of projections of a global type e.g. $\mathsf{T} \leqslant \mathsf{G} \upharpoonright \mathsf{r}$. In our Rocq implementation we define the predicate `issubProj : ltt → gtt → part → Prop` as a shorthand for this.

3.5 Balancedness, Global Tree Contexts and Grafting

We introduce an important constraint on the types of global type trees we will consider, *balancedness*. We omit the technical details of the definition and the Rocq implementation; they can be found in [20] and [16].

▶ **Definition 3.5** (Balanced Global Type Trees). *A path on a tree is a sequence of nodes starting from the root such that every node is a child of the preceding one. A global type tree* G *is balanced if for any subtree* G' *of* G*, there exists* k *such that for all* $\mathsf{p} \in \mathtt{pt}(\mathsf{G}')$*,* p *occurs on every path from the root of* G' *that has length at least* k *or ends in* `end`*.*

Balancedness is a regularity condition that imposes a notion of *liveness* on the protocol described by the global type tree. Indeed, our liveness results in Section 6 hold only for balanced global types. Another reason for formulating balancedness is that it allows us to use the *grafting* technique, turning proofs by coinduction on infinite trees to proofs by induction on finite *global type tree contexts*, or *g-contexts* for short.

▶ **Definition 3.6** (g-contexts and Grafting). *g-contexts are defined inductively with the following syntax :*

$$\mathcal{G} ::= \quad \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathcal{G}_i\}_{i\in I} \quad | \quad [\,]_j$$

```
Inductive gtth: Type ≜
  | gtth_hol    : fin → gtth
  | gtth_send   : part → part → list
    (option (sort * gtth)) → gtth.
```

Given a g-context $\mathcal{G}$ *whose holes are in the indexing set* J *and a set of global types* $\{\mathsf{G}_j\}_{j\in J}$*, the grafting* $\mathcal{G}[\mathsf{G}_j]_{j\in J}$ *denotes the global type tree obtained by substituting* $[\,]_j$ *with* G_j *in* $\mathcal{G}$*.*

In Rocq the indexed set $\{\mathsf{G}_j\}_{j\in J}$ *is represented using a* `list (option gtt)`*. Grafting is expressed with the inductive relation* `typ_gtth : list (option gtt) → gtth → gtt → Prop`*.* `typ_gtth gs gcx gt` *means that the grafting of the set of global type trees* `gs` *onto the g-context* `gcx` *results in the tree* `gt`*. We additionally define* `pt` *and* `ishParts` *on g-contexts analogously to* `pt` *and* `isgPartsC` *on trees .*

A g-context can be thought of as the finite prefix of a global type tree, where holes $[\,]_j$ indicate the cutoff points. g-contexts are related to global type trees with the *grafting* operation that fills in the holes with type trees. The following lemma relates g-contexts to balanced global type trees.

▶ **Lemma 3.7** (Proper Grafting Lemma, [16]). *If* G *is a balanced global type tree and* $\mathsf{p} \in \mathtt{pt}(\mathsf{G})$*, then there is a g-context* $\mathcal{G}$ *and an indexed set* $\{\mathsf{G}_i\}_{i\in I}$ *such that* $\mathcal{G}[\mathsf{G}_i]_{i\in I} = \mathsf{G}$*,*

$\mathsf{p} \notin \mathtt{pt}(\mathcal{G})$, *and for all* $i \in I$, G_i *is in the shape* $\mathtt{end}$, $\mathsf{p} \to \mathsf{q} : \{\dots\}$ *or* $\mathsf{q} \to \mathsf{p} : \{\dots\}$. *In this case, we refer to* $\mathcal{G}$ *and* $\{\mathsf{G}_i\}_{i \in I}$ *as the* p*-grafting of* G. *When we do not care about the* G_i *we may just say that* G *is* p*-grafted by* $\mathcal{G}$.

Lemma 3.7 allows us to turn proofs by coinduction on infinite trees to proofs by induction on the grafting g-context—one of the main proof techniques used in this work.

▶ Remark 3.8. From now on, all the global type trees we will be referring to are assumed to be balanced. When talking about the Rocq implementation, any `G : gtt` we mention is assumed to satisfy the predicate `wfgC G`, expressing that `G` corresponds to a well-formed [16, after Definition 24], balanced type. Furthermore, we will often require that a global type is projectable onto all its participants. This is captured by the predicate `projectableA G = ∀ p, ∃ T, projectionC G p T`. As with `wfgC`, we will be assuming that all types we mention are projectable.

4 Semantics of Global and Local Types

In this section we introduce local type environments, and define Labelled Transition System semantics on these constructs.

4.1 Local Type Environments and Reductions

We start by defining local type environments, also called *local type contexts* in the related work [49, 38, 50].

▶ **Definition 4.1** (Local Type Environments). *Local type environments are defined as a finite mapping of participants to local type trees with the following :*

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma, \mathsf{p} : \mathsf{T}$$

```
Module M ≜ MMaps.RBT.Make(Nat).
Module MF ≜
  MMaps.Facts.Properties Nat M.
Definition tctx: Type ≜ M.t ltt.
```

Intuitively, $\mathsf{p} : \mathsf{T}$ means that participant p is associated with a process that has the type tree T. We write $\mathrm{dom}(\Gamma)$ to denote the set of participants occurring in Γ, and abbreviate the singleton type environment consisting of the pair p and T as $\mathsf{p} : \mathsf{T}$. We write $\Gamma(\mathsf{p})$ for the type of p in Γ. We define the composition Γ_1, Γ_2 iff $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$.

In the Rocq implementation we implement local type environments `tctx` as finite maps of participants, which are represented as natural numbers, and local type trees. We use the finite map implementation of the MMaps library [32]. We further enforce the non-emptiness of the indexing sets of the local type trees by positing that for any local type tree in any environment we mention, the well-formedness predicate `wflttC T` holds. This is expressed by the predicate `tctx_wf: tctx → Prop`.

We give LTS semantics to local type environments.

▶ **Definition 4.2** (Transition labels). *A transition label α has the following form:*

$$\begin{array}{lll} \alpha ::= \mathsf{p} : \mathsf{q}\&\ell(S) & & (\mathsf{p} \textit{ receives a value of sort } S \textit{ from } \mathsf{q} \textit{ with message label } \ell) \\ \quad \mid \mathsf{p} : \mathsf{q}\oplus\ell(S) & & (\mathsf{p} \textit{ sends a value of sort } S \textit{ to } \mathsf{q} \textit{ with message label } \ell\ ) \\ \quad \mid (\mathsf{p},\mathsf{q})\ell & & (\textit{A synchronised communication from } \mathsf{p} \textit{ to } \mathsf{q} \textit{ occurs via label } \ell) \end{array}$$

We further define the function $\mathrm{subject}(\alpha)$ *as* $\mathrm{subject}(\mathsf{p} : \mathsf{q}\&\ell(S)) = \mathrm{subject}(\mathsf{p} : \mathsf{q}\oplus\ell(S)) = \{\mathsf{p}\}$ *and* $\mathrm{subject}((\mathsf{p},\mathsf{q})\ell) = \{\mathsf{p},\mathsf{q}\}$.

▶ **Definition 4.3** (Typing environment reductions). *The typing environment transition* $\xrightarrow{\alpha}$ *is defined inductively by the following rules:*

$$\frac{k \in I}{\mathsf{p} : \mathsf{q}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \xrightarrow{\mathsf{p}:\mathsf{q}\&\ell_k(S_k)} \mathsf{p} : \mathsf{T}_k}\ [\Gamma\text{-}\&] \qquad \frac{k \in I}{\mathsf{p} : \mathsf{q}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_k(S_k)} \mathsf{p} : \mathsf{T}_k}\ [\Gamma\text{-}\oplus]$$

$$\frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, \mathsf{p} : \mathsf{T} \xrightarrow{\alpha} \Gamma', \mathsf{p} : \mathsf{T}}\ [\Gamma\text{-},] \qquad \frac{\Gamma_1 \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)} \Gamma_1' \qquad \Gamma_2 \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell(S')} \Gamma_2' \qquad S \leq S'}{\Gamma_1, \Gamma_2 \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma_1', \Gamma_2'}\ [\Gamma\text{-}\oplus\&]$$

We write $\Gamma \xrightarrow{\alpha}$ *if there exists* Γ' *such that* $\Gamma \xrightarrow{a} \Gamma'$. *We define a reduction* $\Gamma \to \Gamma'$ *to hold iff* $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma'$ *for some* p, q, ℓ. *We write* $\Gamma \to$ *iff* $\Gamma \to \Gamma'$ *for some* Γ'. *We write* $\to^*$ *for the reflexive transitive closure of* $\to$.

[Γ-⊕] and [Γ-&], express a single participant sending or receiving. [Γ-⊕&] expresses a synchronised communication where one participant sends while another receives, and they both progress with their continuation. [Γ-,] shows how to extend an environment. In Rocq typing environment reductions are defined with the predicate `tctxR`.

```
Inductive tctxR: tctx → label → tctx → Prop :=
  | Rsend: ...
  | Rrecv: ...
  | Rcomm: ...
  | RvarI: ...
  | Rstruct: ∀ g1 g1' g2 g2' l, tctxR g1' l g2' →
      M.Equal g1 g1' → M.Equal g2 g2' → tctxR g1 l g2.
```

The first four constructors in the definition of `tctxR` correspond to the rules in Definition 4.3, and `Rstruct` expresses the indistinguishability of local environments under the `M.Equal` predicate from the MMaps library. `M.Equal` lets us consider two finite maps with the same keys mapping to the same values as equal, and is the main notion of equality we use for typing environments in this paper.

We illustrate typing environment reductions with an example.

▶ **Example 4.4.** Let $\Gamma = \{\mathsf{p} : \mathsf{T}_\mathsf{p},\ \mathsf{q} : \mathsf{T}_\mathsf{q},\ \mathsf{r} : \mathsf{T}_\mathsf{r}\}$ where $\mathsf{T}_\mathsf{p} = \mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathsf{T}_\mathsf{p}, \ell_1(\mathtt{int}).\mathtt{end}\}$, $\mathsf{T}_\mathsf{q} = \mathsf{p}\&\{\ell_0(\mathtt{int}).\mathsf{T}_\mathsf{q}, \ell_1(\mathtt{int}).\mathsf{r}\oplus\{\ell_2(\mathtt{int}).\mathtt{end}\}\}$ and $\mathsf{T}_\mathsf{r} = \mathsf{q}\&\{\ell_2(\mathtt{int}).\mathtt{end}\}$. We have the reductions $\Gamma \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_0(\mathtt{int})} \Gamma$ and $\Gamma \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell_0(\mathtt{int})} \Gamma$, which synchronise to give the reduction and $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma$. Similarly via synchronised communication of p and q via message label ℓ_1 we get $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1} \Gamma'$ where Γ' is defined as $\{\mathsf{p} : \mathtt{end},\ \mathsf{q} : \mathsf{r}\oplus\{\ell_2(\mathtt{int}).\mathtt{end}\}, \mathsf{r} : \mathsf{T}_\mathsf{r}\}$. We further have that $\Gamma' \xrightarrow{(\mathsf{q},\mathsf{r})\ell_2} \Gamma_\mathtt{end}$ where $\Gamma_\mathtt{end}$ is defined as $\{\mathsf{p} : \mathtt{end},\ \mathsf{q} : \mathtt{end},\ \mathsf{r} : \mathtt{end}\}$.

In Rocq, Γ is defined the following way:

```
Definition prt_p≜0.
Definition prt_q≜1.
Definition prt_r≜2.
CoFixpoint T_p ≜ ltt_send prt_q [Some (sint,T_p); Some (sint,ltt_end); None].
CoFixpoint T_q ≜ ltt_recv prt_p [Some (sint,T_q); Some (sint, ltt_send prt_r
    [None;None;Some (sint,ltt_end)]); None].
Definition T_r ≜ ltt_recv prt_q [None;None; Some (sint,ltt_end)].
Definition gamma ≜ M.add prt_p T_p (M.add prt_q T_q (M.add prt_r T_r M.empty)).
```

Now $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma$ can be expressed as `tctxR gamma (lcomm prt_p prt_q 0) gamma`.

4.2 Global Type Reductions

As with local type environments, we can also define reductions for global types.

▶ **Definition 4.5** (Global type reductions). *The global type transition* $\xrightarrow{\alpha}$ *is defined coinductively as follows.*

$$\frac{k \in I}{\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_k} \mathsf{G}_k} \text{ [GR-}\oplus\&\text{]}$$

$$\frac{\forall i \in I \;\; \mathsf{G}_i \xrightarrow{\alpha} \mathsf{G}_i' \qquad \mathrm{subject}(\alpha) \cap \{\mathsf{p},\mathsf{q}\} = \emptyset \qquad \forall i \in I \;\; \{\mathsf{p},\mathsf{q}\} \subseteq \mathtt{pt}(\mathsf{G}_i)}{\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I} \xrightarrow{\alpha} \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i'\}_{i\in I}} \text{ [GR-Ctx]}$$

[GR-⊕&] says that a global type tree with root $\mathsf{p} \to \mathsf{q}$ can transition to any of its children corresponding to the message label chosen by p. [GR-Ctx] says that if the subjects of α are disjoint from the root and all its children can transition via α, then the whole tree can also transition via α, with the root remaining the same and just the subtrees of its children transitioning. In Rocq global type reductions are expressed using the coinductively defined predicate `gttstepC`. For example, $\mathsf{G} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_k} \mathsf{G}'$ translates to `gttstepC G G' p q k`. We refer to [16] for details.

4.3 Association Between Local Type Environments and Global Types

We have defined local type environments, which specify protocols bottom-up by directly describing the roles of every participant, and global types, which give a top-down view of the whole protocol, and the transition relations on them. We relate these local and global definitions by defining *association* between local type environment and global types.

▶ **Definition 4.6** (Association). *A local type environment* Γ *is associated with a global type tree* G*, written* $\Gamma \sqsubseteq \mathsf{G}$*, if the following hold:*

- *For all* $\mathsf{p} \in \mathtt{pt}(\mathsf{G})$, $\mathsf{p} \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(\mathsf{p}) \leqslant \mathsf{G} \upharpoonright \mathsf{p}$.
- *For all* $\mathsf{p} \notin \mathtt{pt}(\mathsf{G})$, *either* $\mathsf{p} \notin \mathrm{dom}(\Gamma)$ *or* $\Gamma(\mathsf{p}) = \mathtt{end}$.

In Rocq this is expressed with the predicate `assoc : tctx → gtt → Prop`.

Informally, $\Gamma \sqsubseteq \mathsf{G}$ says that the local type trees in Γ obey the specification described by the global type tree G.

▶ **Example 4.7.** In Example 4.4, we have that $\Gamma \sqsubseteq \mathsf{G}$ where $\mathsf{G} := \mathsf{p} \to \mathsf{q} : \{\ell_0(\mathtt{int}).\mathsf{G}, \ell_1(\mathtt{int}).\mathsf{q} \to \mathsf{r} : \{\ell_2(\mathtt{int}).\mathtt{end}\}\}$ (note that G is not a balanced global type tree due to the infinite path of ℓ_0 communications that do not involve r). In fact, we have $\Gamma(\mathsf{s}) = \mathsf{G} \upharpoonright \mathsf{s}$ for $\mathsf{s} \in \{\mathsf{p},\mathsf{q},\mathsf{r}\}$. Similarly, we have $\Gamma' \sqsubseteq \mathsf{G}'$ where $\mathsf{G}' := \mathsf{q} \to \mathsf{r} : \{\ell_2(\mathtt{int}).\mathtt{end}\}$.

It is desirable to have the association be preserved under local type environment and global type reductions, that is, when one of the associated constructs "takes a step" so should the other. We formalise this *operational correspondence* property with the following soundness and completeness theorems.

▶ **Theorem 4.8** (Soundness of Association). *If* $\Gamma \sqsubseteq \mathsf{G}$ *and* $\mathsf{G} \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \mathsf{G}'$*, then there is a local type environment* Γ'*, a global type* G'' *and a message label* ℓ' *such that* $\mathsf{G} \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \mathsf{G}''$*,* $\Gamma' \sqsubseteq \mathsf{G}''$ *and* $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma'$*.*

▶ **Theorem 4.9** (Completeness of Association). *If* $\Gamma \sqsubseteq \mathsf{G}$ *and* $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma'$*, then there exists a global type tree* G' *such that* $\Gamma' \sqsubseteq \mathsf{G}'$ *and* $\mathsf{G} \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \mathsf{G}'$*.*

▶ Remark 4.10. Note that in the statement of soundness we allow the message label for the local type environment reduction to be different from the message label for the global type reduction. This is because our use of subtyping in association causes the entries in the local type environment to be less expressive than the types obtained by projecting the global type. For example consider $\Gamma = \mathsf{p} : \mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathtt{end}\}$, $\mathsf{q} : \mathsf{p}\&\{\ell_0(\mathtt{int}).\mathtt{end}, \ell_1(\mathtt{int}).\mathtt{end}\}$ and $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_0(\mathtt{int}).\mathtt{end}, \ell_1(\mathtt{int}).\mathtt{end}\}$. We have $\Gamma \sqsubseteq \mathsf{G}$ and $\mathsf{G} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1}$. However $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1}$ is not a valid transition. But the $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0}$ transition that involves the same participants is valid, so the local type environment can still match the transition of the global type to some extent.

5 Properties of Local Type Environments

We now use the LTS semantics to define some desirable properties of type environments and their reduction sequences. Namely, we formulate safety, fairness and liveness properties based on the definitions in [49].[2]

5.1 Safety

We start by defining the *safety* property that plays an important role in bottom-up session type systems [41]:

▶ **Definition 5.1** (Safe Local Type Environments). *We define* safe *coinductively as the largest set of local type Environments such that whenever we have* $\Gamma \in \mathsf{safe}$*:*

$$\Gamma \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)} \text{ and } \Gamma \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell'(S')} \text{ implies } \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \qquad [\text{S-}\&\oplus]$$

$$\Gamma \to \Gamma' \text{ implies } \Gamma' \in \mathsf{safe} \qquad [\text{S-}\to]$$

We write $\mathsf{safe}(\Gamma)$ *if* $\Gamma \in \mathsf{safe}$*.*

Safety says that if p and q attempt to communicate with each other and p requests to send a value using message label ℓ, then q should be able to receive that message label. Furthermore, this property should be preserved under any type environment reductions.

[2] Whereas in general, "safety" and "liveness" refer to classes of properties, following [41] we here refer to specific safety and liveness properties, ones that are particularly relevant for MPST.

Being a coinductive property, to show that $\mathsf{safe}(\Gamma)$, it suffices to give a set φ such that $\Gamma \in \varphi$ and φ satisfies [S-&⊕] and [S-→] . This amounts to showing that every element of Γ' of the set of reducts of Γ, defined $\varphi := \{\Gamma' \mid \Gamma \to^* \Gamma'\}$, satisfies [S-&⊕] . We illustrate this with some examples:

▶ **Example 5.2.** Let $\Gamma = \mathsf{p} : \mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathtt{end}\}, \mathsf{q} : \mathsf{p}\&\{\ell_0(\mathtt{nat}).\mathtt{end}\}$. Γ is not safe as we have $\Gamma \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_0}$ and $\Gamma \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell_0}$ but we do not have $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0}$ as $\mathtt{int} \not\leqslant \mathtt{nat}$.

Consider Γ from Example 4.4. All the reducts satisfy [S-&⊕] , hence Γ is safe.

In Rocq, we define safe coinductively with Paco:

```
Definition weak_safety (c: tctx ) ≜
  ∀ p q s s'  k k', tctxRE (lsend p q (Some s) k) c →
  tctxRE (lrecv q p (Some s') k') c → tctxRE (lcomm p q k) c.
Inductive safe (R: tctx → Prop): tctx → Prop ≜
  | safety_red :  ∀ c, weak_safety c →
  (∀ p q c' k,  tctxR c (lcomm p q k) c' → ∃ c'', M.Equal c' c'' ∧ R c'')
    →  safe R c.
Definition safeC c ≜ paco1 safe bot1 c.
```

In the above, `weak_safety` corresponds to [S-&⊕] where `tctxRE l c` is shorthand for `∃ c', tctxR c l c'`. In the type `safe`, the constructor `safety_red` corresponds to [S-→] (up to the predicate `M.Equal`). Then `safeC` is defined as the greatest fixed point of `safe` and constructed by the Paco function `paco1 safe bot1 c`.

We have that local type environments with associated global types are always safe.

▶ **Theorem 5.3** (Safety by Association). *If* $\Gamma \sqsubseteq \mathsf{G}$ *then* $\mathsf{safe}(\Gamma)$.

5.2 Fairness and Liveness

We now focus our attention on fairness and liveness. We first restate the definition of fairness and liveness for local type environment paths from [49].

▶ **Definition 5.4** (Fair, Live Paths). *A local type environment reduction path (also called an execution or a run) is a possibly infinite sequence of transitions* $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_1} ..$ *such that* λ_i *is a synchronous transition label, that is, of the form* $(\mathsf{p},\mathsf{q})\ell$*, for all* i*.*

We say that a local type environment reduction path $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_2} ..$ *is* fair *if, for all valid* $n \in \mathbb{N} : \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ *implies* $\exists k, \ell'$ *such that* $k \geq n$ *and* $\lambda_k = (\mathsf{p},\mathsf{q})\ell'$*, and therefore* $\Gamma_k \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_{k+1}$*. We say that a path* $(\Gamma_n)_{n\in N}$ *is* live *iff,* $\forall n \in N$*:*

1. $\Gamma_n \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)}$ *implies* $\exists k, \ell'$ *such that* $N \ni k \geq n$ *and* $\Gamma_k \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_{k+1}$
2. $\Gamma_n \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell(S)}$ *implies* $\exists k, \ell'$ *such that* $N \ni k \geq n$ *and* $\Gamma_k \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_{k+1}$

▶ **Definition 5.5** (Live Local Type Environment). *A local type environment* Γ *is live if whenever* $\Gamma \to^* \Gamma'$*, every fair path starting from* Γ' *is also live.*

Informally, liveness says that every communication request on the path is eventually answered. With our fairness assumption [22], we focus on "sensible" reduction paths where every communication that's enabled by both participants is eventually executed. Live type environments are then defined to be the Γ such that whenever Γ can evolve (in possibly multiple steps) into Γ', all fair paths that start from Γ' are also live.

▶ **Example 5.6.** Consider the environments Γ, Γ' and Γ_{end} from Example 4.4. One possible reduction path is $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \dots$. Denote this path as $(\Gamma_n)_{n\in\mathbb{N}}$, where $\Gamma_n = \Gamma$ for all $n \in \mathbb{N}$. We have $\forall n, \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0}$ and $\Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1}$ as the only possible synchronised reductions from Γ_n. Accordingly, we also have $\forall n, \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma_{n+1}$ in the path so this path is fair. However, this path is not live as we have $\Gamma_1 \xrightarrow{\mathsf{r}:\mathsf{q}\&\ell_2(\mathtt{int})}$ but there is no n, ℓ' with $\Gamma_n \xrightarrow{(\mathsf{q},\mathsf{r})\ell'} \Gamma_{n+1}$ in the path. Consequently, Γ is not a live type environment.

Now consider the reduction path $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1} \Gamma' \xrightarrow{(\mathsf{q},\mathsf{r})\ell_2} \Gamma_{\mathsf{end}}$. This path is fair and live as it contains the (q,r) transition from the counterexample above.

Definition 5.4, while intuitive, is not really convenient for a Rocq formalisation due to its explicit use of indices to quantify over the contexts in a path. Proofs in this setting would require additional bookkeeping to keep track of all the indices used. All this extra complexity could be avoided if these properties could be expressed as a least or greatest fixed point, which could then be formalised via Rocq's inductive or (via Paco) coinductive types. To achieve this, we recast fairness and liveness for local type environment paths in Linear Temporal Logic (LTL) [39]. The LTL operators *eventually* ($\Diamond$) and *always* ($\Box$) are characterised as least and greatest fixed points using their expansion laws [2, Chapter 5.14]. Hence they are implemented in Rocq as the inductive type `eventually` and the coinductive type `alwaysCG`. We can further represent reduction paths as *cosequences*, or *streams*. Then the Rocq definition of Definition 5.4 amounts to the following:

```
CoInductive coseq (A: Type): Type ≜
  | conil : coseq A
  | cocons: A → coseq A → coseq A.
Notation local_path ≜ (coseq (tctx*option label)).
Definition fair_path_local_inner (pt: local_path): Prop ≜
  ∀ p q n, to_path_prop (tctxRE (lcomm p q n)) False pt →
  eventually (headComm p q) pt.
Definition fair_path ≜ alwaysCG fair_path_local_inner.
Definition live_path_inner (pt: local_path) : Prop ≜ ∀ p q s n,
(to_path_prop (tctxRE (lsend p q (Some s) n)) False pt → eventually (headComm p q)
    pt) ∧
(to_path_prop (tctxRE (lrecv p q (Some s) n)) False pt → eventually (headComm q p)
    pt).
Definition live_path ≜ alwaysCG live_path_inner.
```

With these definitions we can now prove that local type environments associated with a global type are live, which is the most involved of the results mechanised in this work.

▶ Remark 5.7. We once again emphasise that all global types mentioned are assumed to be balanced (Definition 3.5). Indeed association with non-balanced global types doesn't guarantee liveness. As an example, consider Γ from Example 4.4, which is associated with G from Example 4.7. Yet we have shown in Example 5.6 that Γ is not a live type environment. This is not surprising as G is not balanced.

▶ **Theorem 5.8** (Liveness by Association). *If $\Gamma \sqsubseteq \mathsf{G}$ then Γ is live.*

Proof. (Outline) Our proof proceeds in two steps. First, we prove that the type environment obtained by direct projections[3] of G, that is, $\Gamma_{\mathtt{proj}} = \{\mathsf{p}_i : \mathsf{G} \upharpoonright \mathsf{p}_i \mid \mathsf{p}_i \in \mathtt{pt}(\mathsf{G})\}$, is live.

[3] The actual Rocq proof defines an equivalent "enabledness" predicate on global types instead of working with direct projections. The outline given here is a slightly simplified presentation.

We then leverage Theorem 4.8 and Theorem 4.9 to show that if $\Gamma_{\texttt{proj}}$ is live, so is Γ .

Suppose $\Gamma_{\texttt{proj}} \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)}$ (the case for the receive is similar and omitted), and ρ is a fair local type environment reduction path beginning with $\Gamma_{\texttt{proj}}$. To show that ρ is live we need to show the existence of a $(\mathsf{p},\mathsf{q})\ell$ transition in ρ. We achieve this by taking the height of the p-grafting of the global type associated with the head of ρ as our induction invariant. We show that this invariant is bounded from below by the height of the q-grafting, and that it keeps decreasing (,) until the p-graftings and q-graftings become equal, at which point a $(\mathsf{p},\mathsf{q})\ell$ transition is enabled on the path . Our fairness assumption then forces that transition to fire on ρ.

In the second step of the proof, we extend association on to paths to obtain, for each local type environment reduction path σ that begins with Γ, another local type environment reduction path σ' beginning with $\Gamma_{\texttt{proj}}$ such that the elements of σ are subtypes (subtyping on environments defined pointwise) of the corresponding elements of σ', and the corresponding transitions of σ and σ' have the same labels. . This is obtained from Theorem 4.9; however, in Rocq, the statement of Theorem 4.9 is implemented as an $\exists$ statement that lives in `Prop`, hence we need to use the `constructive_indefinite_description` axiom to construct a `CoFixpoint` returning the desired cosequence σ' . Then Theorem 4.8 and Theorem 4.9 are used to show that the liveness of $\Gamma_{\texttt{proj}}$ implies the liveness of Γ, completing the proof. ◀

6 Properties of Multiparty Sessions

We define typing rules for the session calculus introduced in Section 2, and prove subject reduction and deadlock freedom for them. Then we define a liveness property for sessions, and show that processes typable by a local type environment that's associated with a global type tree are guaranteed to satisfy this liveness property.

6.1 Typing rules

We give typing rules for our session calculus based on [20] and [16]. We have two kinds of typing judgements and type environments. $\Theta \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}$ says that the single process P can be typed with local type T using expression and type variables from Θ. On the other hand, $\Gamma \vdash \mathcal{M}$ expresses that session $\mathcal{M}$ can be typed by the local type environment (Definition 4.1). Typing rules for expressions are standard and can be found in e.g. [20], and are therefore omitted.

$$\frac{}{\Theta \vdash_{\mathsf{P}} \mathbf{0} : \texttt{end}}\ [\text{T-END}] \qquad \frac{}{\Theta, \mathbf{X} : \mathsf{T} \vdash_{\mathsf{P}} \mathbf{X} : \mathsf{T}}\ [\text{T-VAR}] \qquad \frac{\Theta, \mathbf{X} : \mathsf{T} \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}}{\Theta \vdash_{\mathsf{P}} \mu\mathbf{X}.\mathsf{P} : \mathsf{T}}\ [\text{T-REC}]$$

$$\frac{\Theta \vdash_{\mathsf{P}} e : \texttt{bool} \quad \Theta \vdash_{\mathsf{P}} \mathsf{P}_1 : \mathsf{T} \quad \Theta \vdash_{\mathsf{P}} \mathsf{P}_2 : \mathsf{T}}{\Theta \vdash_{\mathsf{P}} \texttt{if e then } \mathsf{P}_1 \texttt{ else } \mathsf{P}_2 : \mathsf{T}}\ [\text{T-IF}]$$

$$\frac{\Theta \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T} \quad \mathsf{T} \leqslant \mathsf{T}'}{\Theta \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}'}\ [\text{T-SUB}] \qquad \frac{\forall i \in I, \quad \Theta, x_i : \mathsf{S}_i \vdash_{\mathsf{P}} \mathsf{P}_i : \mathsf{T}_i}{\Theta \vdash_{\mathsf{P}} \sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{P}_i : \mathsf{p}\&\{\ell_i(\mathsf{S}_i).\mathsf{T}_i\}_{i\in I}}\ [\text{T-IN}]$$

$$\frac{\Theta \vdash_{\mathsf{P}} e : \mathsf{S} \quad \Theta \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}}{\Theta \vdash_{\mathsf{P}} \mathsf{p}!\ell(\mathsf{e}).\mathsf{P} \ : \ \mathsf{p}\oplus\{\ell(\mathsf{S}).\mathsf{T}\}}\ [\text{T-OUT}]$$

$$\frac{\forall i \in I : \quad \vdash_{\mathsf{P}} \mathsf{P}_i : \Gamma(\mathsf{p}_i) \qquad \Gamma \sqsubseteq \mathsf{G}}{\Gamma \vdash \prod_i \mathsf{p}_i \triangleleft \mathsf{P}_i}\ [\text{T-SESS}]$$

Table 2 The typing rules for processes and multiparty sessions

Table 2 states the standard [16, 20] typing rules for processes, which we do not elaborate

on. The main rule for typing multiparty sessions is [T-SESS] : it states that a session made of the parallel composition of processes $\prod_i \mathsf{p}_i \triangleleft \mathsf{P}_i$ can be typed by an associated local environment Γ if the local type of participant p_i in Γ types the process P_i. This is expressed in Rocq with the predicate `typ_sess : session → tctx → Prop`.

6.2 Properties of Typed Sessions

We can now prove some properties of typed sessions. The following theorems relating session reductions to types underlie our results.

▶ **Lemma 6.1** (Typing after Unfolding). *If* $\Gamma \vdash \mathcal{M}$ *and* $\mathcal{M} \Rrightarrow \mathcal{M}'$ *then* $\Gamma \vdash \mathcal{M}'$.

▶ **Theorem 6.2** (Subject Reduction). *If* $\Gamma \vdash \mathcal{M}$ *and* $\mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \mathcal{M}'$, *then there exists a type environment* Γ' *such that* $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma'$ *and* $\Gamma' \vdash \mathcal{M}'$.

▶ **Theorem 6.3** (Session Fidelity). *If* $\Gamma \vdash \mathcal{M}$ *and* $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma'$, *then there exists a message label* ℓ', *a environment* Γ'' *and a session* $\mathcal{M}'$ *such that* $\mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \mathcal{M}'$, $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma''$ *and* $\Gamma'' \vdash \mathcal{M}'$.

Lemma 6.1 says that typing is preserved after unfolding. Theorem 6.2 shows that the type environment reduces along with the session it types. Theorem 6.3 is an analogue of Theorem 6.2 in the opposite direction. As in Theorem 4.8, we allow the labels ℓ and ℓ' to be different in Theorem 6.3.

▶ Remark 6.4. Note that in Theorem 6.2 one transition between sessions corresponds to exactly one transition between local type environments with the same label. That is, every session transition is observed by the corresponding type. This is the main reason for our choice of reactive semantics (Section 2.2) as τ transitions are not observed by the type in ordinary semantics. In other words, with τ-semantics the typing relation is a *weak simulation* [35], while it turns into a strong simulation with reactive semantics. For our Rocq implementation working with the strong simulation turns out be more convenient as the mechanisation of weak simulation tends to be not so straightforward [11, 10].

Now we can prove two of our main results, communication safety and deadlock freedom:

▶ **Theorem 6.5** (Communication Safety). *Assume* $\Gamma \vdash \mathcal{M}$ *and* $\mathcal{M} \to^* \mathcal{M}'$.
If $\mathcal{M}' \Rrightarrow (\mathsf{p} \triangleleft \mathsf{q}!\ell_i(e).\mathsf{P} \mid \mathsf{q} \triangleleft \mathsf{p}?\{\ell_j(x_j).\mathsf{Q}_j\}_{j\in J} \mid M'')$, *then* $i \in J$.

Theorem 6.5 means that typed sessions evolve to sessions where if participant $\mathtt{p}$ wants to send to $\mathtt{q}$ with label ℓ_i, and $\mathtt{q}$ is listening to receive from $\mathtt{p}$, then $\mathtt{q}$ is able to receive with label ℓ_i.

▶ **Theorem 6.6** (Deadlock Freedom). *If* $\Gamma \vdash \mathcal{M}$, *one of the following hold :*

1. *Whenever* $\mathcal{M} \Rrightarrow \mathcal{M}'$, $\mathcal{M}' \Rrightarrow \mathcal{M}_{inact}$ *where every process making up* $\mathcal{M}_{inact}$ *is inactive, i.e.* $\mathcal{M}_{inact} \equiv \prod_{i=1}^{n} \mathsf{p}_i \triangleleft \mathbf{0}$ *for some* n.
2. *Or there exists* $\mathcal{M}'$ *such that* $\mathcal{M} \to \mathcal{M}'$.

Theorem 6.6 says that the only way a typed session has no reductions available is if it has terminated.

The final, and the most intricate, session property we prove is liveness.

▶ **Definition 6.7** (Session Liveness). *Let* $(\rightarrow^*\Rrightarrow)$ *denote the composition of the multistep reduction and unfolding relations i.e.* $\mathcal{N} \rightarrow^*\Rrightarrow \mathcal{N}'$ *iff* $\mathcal{N} \rightarrow^* \mathcal{N}'' \Rrightarrow \mathcal{N}'$ *for some* $\mathcal{N}''$*. Then session* $\mathcal{M}$ *is live iff*

1. $\mathcal{M} \rightarrow^*\Rrightarrow \mathcal{M}' = \mathsf{q} \triangleleft \mathsf{p}!\ell(\mathsf{e}).\mathsf{Q} \mid \mathcal{N}$ *implies* $\mathcal{M}' \rightarrow^* \mathsf{q} \triangleleft \mathsf{Q} \mid \mathcal{N}'$ *for some* $\mathcal{N}'$
2. $\mathcal{M} \rightarrow^*\Rrightarrow \mathcal{M}' = \mathsf{q} \triangleleft \sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{Q}_i \mid \mathcal{N}$ *implies* $\mathcal{M}' \rightarrow^* \mathsf{q} \triangleleft \mathsf{Q}_i[v/x_i] \mid \mathcal{N}'$ *for some* $\mathcal{N}', i, v$.

In Rocq this is expressed with the predicate `live_sess`.

Session liveness says that when $\mathcal{M}$ is live, if $\mathcal{M}$ reduces to a session $\mathcal{M}'$ containing a participant that's attempting to send or receive, then $\mathcal{M}'$ reduces to a session where that communication has happened. It's also called *lock-freedom* in [21, 36].

▶ Remark 6.8. In the premises in Definition 6.7, we have used the composition of multistep reduction and unfolding relations. In contrast, previous work, e.g. [46, Definition 2.1.3], defines the premise of Item 1 in Definition 6.7 as $\mathcal{M} \rightarrow^* \mathcal{M}' \Rrightarrow \mathsf{q} \triangleleft \mathsf{p}!\ell(\mathsf{e}).\mathsf{Q} \mid \mathcal{N}$. However, the latter definition accepts stuck sessions that coincidentally look like their state after taking a step. For example, let $\mathcal{M} = \mathsf{p} \triangleleft \mu\mathsf{X}.\mathsf{q}!\ell(0).\mathsf{X} \mid \mathsf{q} \triangleleft \mathbf{0}$. This session cannot progress and thus should not be considered live. By the previous definition, we have $\mathcal{M} \rightarrow^* \mathcal{M} \Rrightarrow \mathsf{p} \triangleleft \mathsf{q}!\ell(0).\mu\mathsf{X}.\mathsf{q}!\ell(0).\mathsf{X} \mid \mathsf{q} \triangleleft \mathbf{0}$. Now the session state after the communication of p happens is $\mathsf{p} \triangleleft \mu\mathsf{X}.\mathsf{q}!\ell(0).\mathsf{X} \mid \mathsf{q} \triangleleft \mathbf{0}$ which just happens to equal $\mathcal{M}$. We further have that $\mathcal{M} \rightarrow^* \mathcal{M}$, hence $\mathcal{M}$ satisfies the liveness criteria. Our definition avoids this problem as we require a transition from the unfolded session $\mathsf{p} \triangleleft \mathsf{q}!\ell(0).\mu\mathsf{X}.\mathsf{q}!\ell(0).\mathsf{X} \mid \mathsf{q} \triangleleft \mathbf{0}$.

▶ **Theorem 6.9** (Liveness by Typing). *If* $\Gamma \vdash \mathcal{M}$ *then* $\mathcal{M}$ *is live.*

Proof. We detail the proof for the send case of Definition 6.7; the case for the receive is similar. Suppose that $\mathcal{M} \rightarrow^*\Rrightarrow \mathcal{N}$ and $\mathcal{N} \equiv \mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}'$. Our goal is to show that there exists a N'' such that $\mathcal{N} \equiv \mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}' \rightarrow^* \mathsf{p} \triangleleft \mathsf{P}' \mid \mathcal{N}''$. By Theorem 6.2, we also have that $\Gamma \vdash \mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}'$ for some Γ.

Now let ρ be a session reduction path starting from $\mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}'$, which has the following fairness property: whenever a transition with label $(\mathsf{p}, \mathsf{q})\ell$ is enabled, a transition with label $(\mathsf{p}, \mathsf{q})\ell'$ eventually occurs for some ℓ'. It is shown that such a path always exists, and that path can be constructed using the axioms `constructive_indefinite_description` and `excluded_middle_informative`. We now show that the existence of this path implies the liveness of the session.

By extending Theorem 6.2 onto paths, let ρ' be a local type environment reduction path starting with Γ such that every session in ρ is typed by the environment at the corresponding index of ρ', and the transitions of ρ and ρ' at every step match. Now we can show that ρ' is fair. Therefore by Theorem 5.8, path ρ' is live, so transition $(\mathsf{p}, \mathsf{q})\ell'$ eventually occurs in ρ' for some ℓ'. Therefore $\rho' = \Gamma \rightarrow^* \Gamma_0 \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_1 \rightarrow ..$ for some Γ_0, Γ_1.

Now consider the session $\mathcal{N}_0$ typed by Γ_0 in ρ. We have $\mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}' \rightarrow^* \mathcal{N}_0$ by $\mathcal{N}_0$ being on ρ. We have that $\mathcal{N}_0 \xrightarrow{(\mathsf{p},\mathsf{q})\ell''} \mathcal{N}_1$ for some $\ell'', \mathcal{N}_1$ by Theorem 6.3. Observe that $\mathcal{N}_0 \equiv \mathsf{p} \triangleleft \mathsf{q}!\ell(e).\mathsf{P}' \mid \mathcal{N}''$ for some $\mathcal{N}''$ as no transitions involving p have happened on the reduction path to $\mathcal{N}_0$. Therefore $\ell = \ell''$, so $\mathcal{N}_1 \equiv \mathsf{p} \triangleleft \mathsf{P}' \mid \mathcal{N}''$ for some $\mathcal{N}''$, as needed. ◀

Theorem 6.9 shows that typable sessions are live.

7 Related Work

Examinations of liveness, also called *lock-freedom*, guarantees of multiparty session types abound in the literature, e.g. [37, 41, 49, 3]. Most of these papers use the definition of liveness proposed by Padovani [36], which does not make the fairness assumptions that characterise the property [18] explicit. Contrastingly, van Glabbeek et al. [21] examine several notions of fairness and the liveness properties induced by them, and devise a type system with flexible choices [7] that captures the strongest of these properties, the one induced by the *justness* [22] assumption. In their terminology, Definition 6.7 roughly corresponds to liveness under full fairness, which is the weakest of the properties considered in that paper. They also show that their type system is complete, i.e. every live process can be typed. We haven't presented any completeness results in this paper. Fairness assumptions are also made explicit in recent work by Ciccone et al. [13, 14], which use generalised inference systems with coaxioms [1] to characterise *fair termination*, which is a stronger property than Definition 6.7, but enjoys good compositionality properties.

Mechanisation of session types in proof assistants is a relatively new effort. Our formalisation is built on recent work by Ekici et al. [16], which uses a coinductive representation of global and local types to prove subject reduction and deadlock-freedom. Their work uses a typing relation between global types and sessions while ours uses one between associated local type environments and sessions. This necessitates the rewriting of subject reduction and deadlock-freedom proofs in addition to the novel operational correspondence, safety and liveness properties we have proved.

In other work, Ekici and Yoshida [17] present a mechanisation of the completeness of asynchronous subtyping. Tirore et al. [45, 43, 44] give the Rocq mechanisation of a projection function for coinductive plain merge projection, and a proof of subject reduction for an asynchronous π-calculus.

Implementations of session types that are more geared towards practical verification include Castro-Perez et al.'s Zooid [8], which is a DSL that supports the extraction of verified protocols via asynchronous MPST. Similar to our work, Zooid mechanises LTS semantics for local type environments and global types, and proves an operational correspondence between a global type and its projections. They then exhibit that the trace of a well-typed session can be embedded in the trace of the global type that types it. Unlike our work, Zooid does not formalise the properties of the type environments, and does not prove type safety and other properties of processes with respect to a typing system. In contrast, our implementation directly certifies that typable sessions are communication safe, deadlock-free and live.

The Actris framework [23] enriches the separation logic of Iris [28] with binary session types to allow reasoning about message passing programs. LinearActris [27] further employs linearity to certify deadlock-free programs. Jacobs et al.'s Multiparty GV [26], based on the functional language of Wadler's GV [47], proves deadlock-freedom of interleaved sessions using multiparty session types and by reasoning on the communication topology of different sessions. None of these works address liveness. In general, verification of liveness properties, with or without session types, in concurrent separation logic is an active research area that has produced tools such as TaDa [15], FOS [30] and LiLo [31] in the past few years.

Castro-Perez et al. [9] devise a multiparty session type system that dispenses with projections and local types by defining the typing relation directly on the LTS specifying the global protocol, and formalise the results in Agda. Li and Weis [33] present a Rocq formalisation of a characterisation of *implementability* for asynchronous global protocols given by a top-down specification. Implementable global protocols are those corresponding to a set of deadlock-free locally specified processes, which is guaranteed in our work by the existence

of an associated global type. Ciccone's PhD thesis [12] presents an Agda formalisation of fair termination for binary session types. Binary session types were also implemented in Agda by Thiemann [42] and in Idris by Brady [6]. Several implementations of binary session types are also present for Haskell [29, 34, 40]. None of the above works formalises liveness.

8 Conclusion and Future Work

In this work we have mechanised the semantics of local and global types, proved a correspondence between them, and used this correspondence to prove safety, deadlock-freedom and liveness for the typed sessions in a simple message-passing calculus. To our knowledge, our liveness result is the first mechanised one of its kind; it is the most challenging of the theorems we formalised. Our implementation illustrates some of the difficulties encountered when mechanising liveness properties in general. These include the use of mixed inductive-coinductive reasoning and the absence of a clear general proof technique.

A proof method for liveness in multiparty session processes using association was proposed in [50], though it was proven by pencil-and-paper based on the inductive full-merging projection. This association technique offers a general proof method for establishing type soundness of the top-down approach, extending to crash-stop failures [4] and asynchronous subtyping [38]. The present paper provides a rigorous Rocq formalisation of the type system based on the association relation for the first time. The mechanisation of the association is not trivial: for instance, the induction on the g-context height used in the proof of Theorem 5.8 requires a careful setup. The proof proceeds smoothly after this setup, as the p-grafting of a global type neatly encodes information about the enabled transition of p. Our work further demonstrates the power of parameterised coinduction in the verification of liveness properties, and provides a framework for the verification of further linear time properties on session types.

Future Work. There are a couple avenues for extension for our work. One apparent addition would be to extend the library to support coinductive full-merge projection [46]. None of the proofs included in this work explicitly use the properties of the merge operation; therefore we expect that if the results of [16] can be generalised to full-merge, so can our work. Another possible variation on our work would be to adapt it to formalise liveness of different process calculi such as the π-calculus, as done on paper in [49]. The design of our proofs provides clear layers of separation between the association of local type environments to global types, which makes the local type environments well-behaved, and the typing relation of sessions by local type environments, which makes the sessions well-behaved. Therefore, we conjecture that any process calculus that satisfies Lemma 6.1, Theorem 6.2 and Theorem 6.3 could have its liveness analysed within our framework without significantly changing the remaining proofs. However, our requirement that the transitions match one-to-one in subject reduction and session fidelity (Remark 6.4) is probably too stringent for most calculi, hence the proofs would have to consider weak simulation relations.

Another possible extension would be to vary our fairness assumptions and examine the liveness properties induced by them, as done in [21]. In this paper the liveness property we targeted for our sessions is called *Padovani lock-freedom* or *lock-freedom under full fairness* in [21]. However, we could have targeted stronger liveness properties parameterised by other fairness assumptions such as *fairness of instructions* or *justness*. As in [21], targeting a different liveness property would likely require us to use a different formulation of global types and a different projection relation, but the code we used to reason about the linear-time properties of the LTS of sessions could be re-used.

Declaration. We confirm that no AI generated text or code is present in this work.

References

1 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing Inference Systems by Coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.

2 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

3 Franco Barbanera and Mariangiola Dezani-Ciancaglini. Partially Typed Multiparty Sessions. *Electronic Proceedings in Theoretical Computer Science*, 383:15–34, August 2023. doi:10.4204/EPTCS.383.2.

4 Adam Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Crash-Stop Failures in Asynchronous Multiparty Session Types. *Logical Methods in Computer Science*, 2025. doi:10.46298/lmcs-21(2:5)2025.

5 Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 1st edition, 2010.

6 Edwin Charles Brady. Type-driven Development of Concurrent Communicating Systems. *Computer Science*, 18(3), July 2017. doi:10.7494/csci.2017.18.3.1413.

7 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Reversible sessions with flexible choices. *Acta Informatica*, 56(7):553–583, November 2019. doi:10.1007/s00236-019-00332-y.

8 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 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 2021, page 237–251. Association for Computing Machinery, 2021. doi:10.1145/3453483.3454041.

9 David Castro-Perez, Francisco Ferreira, and Sung-Shik Jongmans. A synthetic reconstruction of multiparty session types. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi:10.1145/3776692.

10 Nicolas Chappe. A family of sims with diverging interests. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi:10.1145/3776714.

11 Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. Stuttering for free. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622857.

12 Luca Ciccone. Concerto grosso for sessions: Fair termination of sessions, 2023. arXiv:2307.05539.

13 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 139:100964, 2024. doi:10.1016/j.jlamp.2024.100964.

14 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498666.

15 Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), November 2021. doi:10.1145/3477082.

16 Burak Ekici, Tadayoshi Kamegai, and Nobuko Yoshida. Formalising Subject Reduction and Progress for Multiparty Session Processes. In Yannick Forster and Chantal Keller, editors, *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ITP.2025.19.

17 Burak Ekici and Nobuko Yoshida. Completeness of Asynchronous Session Tree Subtyping in Coq. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving (ITP 2024)*, volume 309 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. ISSN: 1868-8969, a long version will appear in ACM Transactions on Computational Logic. doi:10.4230/LIPIcs.ITP.2024.13.

18 Nissim Francez. *Fairness*. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

19 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of LNCS, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

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

21 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21. Association for Computing Machinery, 2021. doi:10.1109/LICS52264.2021.9470531.

22 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4):1–38, August 2019. doi:10.1145/3329125.

23 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019. doi:10.1145/3371074.

24 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008. doi:10.1145/1328897.1328472.

25 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. *SIGPLAN Not.*, 48(1):193–206, January 2013. doi:10.1145/2480359.2429093.

26 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

27 Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Deadlock-free separation logic: Linearity yields progress for dependent higher-order message passing. *Proceedings of the ACM on Programming Languages*, 8(POPL):1385–1417, 2024. doi:10.1145/3632889.

28 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.

29 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 1–13. Association for Computing Machinery, 2021. doi:10.1145/3471874.3472979.

30 Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. Fair operational semantics. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591253.

31 Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. Lilo: A higher-order, relational concurrent separation logic for liveness. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1267–1294, 2025. doi:10.1145/3720525.

32 Pierre Letouzey and Andrew W. Appel. Modular Finite Maps over Ordered Types. URL: https://github.com/rocq-community/mmaps.

33 Elaine Li and Thomas Wies. Certified Implementability of Global Multiparty Protocols. In Yannick Forster and Chantal Keller, editors, *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ITP.2025.15.

34 Sam Lindley and J Garrett Morris. Embedding session types in Haskell. *ACM SIGPLAN Notices*, 51(12):133–145, 2016. doi:10.1145/3241625.2976018.

35 Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 1201–1242. Elsevier, 1990. doi:10.1016/B978-0-444-88074-1.50024-X.

36 Luca Padovani. Deadlock and lock freedom in the linear pi-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14. Association for Computing Machinery, 2014. doi:10.1145/2603088.2603116.

37 Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. Typing Liveness in Multiparty Communicating Systems. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, pages 147–162. Springer, 2014. doi:10.1007/978-3-662-43376-8_10.

38 Kai Pischke and Nobuko Yoshida. Asynchronous Global Protocols, Precisely. In *Components Operationally: Reversibility and System Engineering*, volume 16065 of LNCS, pages 116–133. Springer, 2025. doi:10.1007/978-3-031-99717-4_7.

39 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 1977*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

40 Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36, 2008. doi:10.1145/1411286.1411290.

41 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290343.

42 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19. Association for Computing Machinery, 2019. doi:10.1145/3354166.3354184.

43 Dawit Tirore. *A Mechanisation of Multiparty Session Types*. PhD thesis, IT-Universitetet i København, 2024.

44 Dawit Tirore, Jesper Bengtson, and Marco Carbone. Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECOOP.2025.31.

45 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, Białystok, Poland, July 31 - August 4, 2023*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.28.

46 Thien Udomsrirungruang and Nobuko Yoshida. Top-down or bottom-up? Complexity analyses of synchronous multiparty session types. *Proceedings of the ACM on Programming Languages*, 9(POPL):1040–1071, 2025. doi:10.1145/3704872.

47 Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012. doi:10.1145/2398856.2364568.

48 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D´Souza, editors, *Distributed Computing and Internet Technology*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.

49 Nobuko Yoshida and Ping Hou. Less is more revisited. In Ana Cavalcanti and James Baxter, editors, *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, pages 268–291. Springer, 2024. doi:10.1007/978-3-031-66673-5_14.

50 Nobuko Yoshida, Ping Hou, and Iona Kuhn. Less is More Revisited: Association with Global Protocols and Multiparty Sessions. *Theoretical Computer Science*, 2026. doi:10.1016/J.TCS.2026.115873.