Formally Verified Liveness with Synchronous Multiparty Session Types in Rocq

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

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* (also called *lock-freedom*).

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 contexts*, with global types using this projection and subtyping relations, and prove an *operational correspondence* between a local type context and its associated global type. We then utilize this association relation to prove the safety and liveness of associated local type contexts and, consequently, the multiparty sessions typed by these contexts.

Besides clarifying the often informal proofs of liveness found in the MPST literature, our Rocq mechanisation also enables the certification of lock-freedom properties of communication protocols. Our contribution amounts to around 12K lines of Rocq code.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements Anonymous acknowledgements

1 Introduction

Multiparty session types [19] 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* [13]) (it is possible for the session to progress so long as it has at least one active participant), and *liveness* (also called *lock-freedom* [41] or *starvation-freedom* [8]) (if a process is waiting to send and 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 context*, and the properties of the session is examined by model-checking this local type context. 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:

© Anonymous author(s);
licensed under Creative Commons License CC-BY 4.0
42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$\begin{tikzcd}
\mathsf{G} \arrow[rr, dotted, "\rightarrow"] \arrow[d, dotted, "\upharpoonright"] \arrow[dr, "\sqsubseteq"] & & \mathsf{G}' \arrow[d, "\sqsubseteq"] \\
\mathsf{T} \arrow[r] \arrow[d, dotted, "\vdash_{\mathsf{P}}"] & \{(p_i, T_i) \mid i \in I\} = \Gamma \arrow[r, "\rightarrow"] \arrow[d, "\vdash_{\mathcal{M}}"] & \Gamma' \arrow[d, "\vdash_{\mathcal{M}}"] \\
\mathsf{P} \arrow[r] & \prod_i \mathsf{p}_i \triangleleft \mathsf{P}_i = \mathcal{M} \arrow[r, "\rightarrow"] & \mathcal{M}'
\end{tikzcd}$$

Figure 1 Design overview. The dotted lines correspond to relations inherited from [13] while the solid lines denote relations that are new, or substantially rewritten, in this paper.

the bottom-up approach is generally able to type more sessions, while type-checking and type-inferring in the top-down approach tend to be more efficient than model-checking the bottom-up system [40].

In this work, we present the Rocq [4] formalisation of a synchronous MPST that that ensures the aforementioned properties for typed sessions. Our type system uses an *association* relation ($\sqsubseteq$) [44, 32] defined using (coinductive plain) projection [38] and subtyping, in order to relate local type contexts and global types. This association relation ensures *operational correspondence* between the labelled transition system (LTS) semantics we define for local type contexts and global types. We then type ($\vdash_{\mathcal{M}}$) sessions using local type contexts that are associated with global types, which ensure that the local type context, and hence the session, is well-behaved in some sense. Whenever an associated local type context Γ types a session $\mathcal{M}$, our type system guarantees safety (Theorem 6.5), deadlock-freedom (Theorem 6.6) and liveness (Theorem 6.9). To our knowledge, this work presents the first mechanisation of liveness for multiparty session types in a proof assistant.

Our Rocq implementation builds upon the recent formalisation of subject reduction for MPST by Ekici et. al. [13], which itself is based on [17]. The methodology in [13] 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 [17]. We extensively use these definitions and the lemmas concerning them, but we still depart from and extend [13] in numerous ways by introducing local typing contexts, their correspondence with global types and a new typing relation. Our addition to the code amounts to around 12K lines of Rocq code.

As with [13], our implementation heavily uses the parameterized coinduction technique of the paco [20] 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)[33]. This approach, together with the proof techniques provided by paco, 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 [13]. In Section 4 we give LTS semantics to local type contexts and global types, and detail the association relation between them. In Section 5 we define safety and liveness for local type contexts, 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 these typable sessions.

2 The Session Calculus

We introduce the simple synchronous session calculus that our type system will be used on.

2.1 Processes and Sessions

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

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

where e *is an expression that can be a variable, a value such as* `true`*, 0 or −3, or a term built from expressions by applying the operators* `succ`*,* `neg`*, ¬, non-deterministic choice ⊕ and >.*

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

Processes can be composed in parallel into sessions.

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

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

$\mathsf{p} \triangleleft \mathsf{P}$ denotes that participant p is running the process P, $|$ indicates parallel compositon. 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`.

```
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.
Notation "p '←-' P"  ≜ (s_ind p P) (at level 50, no
    associativity).
Notation "s1 '|||' s2" ≜ (s_par s1 s2) (at level 50, no
    associativity).
```

2.2 Structural Congruence and Operational Semantics

We define a structural congruence relation $\equiv$ on sessions which expresses the commutativity, associativity and unit of the parallel composition operator.

[SC-SYM]	[SC-ASSOC]	[SC-O]
$\mathsf{p} \triangleleft \mathsf{P} \mid \mathsf{q} \triangleleft \mathsf{Q} \equiv \mathsf{q} \triangleleft \mathsf{Q} \mid \mathsf{p} \triangleleft \mathsf{P}$	$(\mathsf{p} \triangleleft \mathsf{P} \mid \mathsf{q} \triangleleft \mathsf{Q}) \mid \mathsf{r} \triangleleft \mathsf{R} \equiv \mathsf{p} \triangleleft \mathsf{P} \mid (\mathsf{q} \triangleleft \mathsf{Q} \mid \mathsf{r} \triangleleft \mathsf{R})$	$\mathsf{p} \triangleleft \mathsf{P} \mid \mathcal{O} \equiv \mathsf{p} \triangleleft \mathsf{P}$

Table 1 Structural Congruence over Sessions

We omit the semantics for expressions, they are standard and can be found in e.g. [17]. We now give the operational semantics for sessions by the means of a labelled transition system. We use labelled *reactive* semantics [41, 6] 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 [13, 17, 41]. For the advantages of our approach see Remark 6.4.

In reactive semantics silent transitions are captured by an *unfolding* relation ($\Rrightarrow$), and β reductions are defined up to this unfolding (Table 2).

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

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

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

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

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

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

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

Table 2 Unfolding and Reductions of Sessions

In Table 2, $\mathcal{M} \Rrightarrow \mathcal{N}$ 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 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 [17]. 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 [13] and hence are omitted here.

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 atomic sorts:* `int`, `bool` *and* `nat`. *Local type trees are then defined coinductively with the following syntax:*

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

```
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.
```

In the above definition, `end` represents a role that has finished communicating. $\mathsf{p}\oplus\{\ell_i(S_i).\mathsf{T}\}_{i\in I}$ denotes a role that may, from any $i \in I$, receive a value of sort S_i with message label ℓ_i and continue with $\mathbb{T}_i$. Similarly, $\mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ 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 $i \in I$.

In Rocq we represent the continuations using a `list` of `option` types. In a continuation `gcs : list (option(sort*ltt))`, index `k` (using zero-indexing) being equal to `Some (s_k, T_k)` means that $\ell_k(S_k).\mathsf{T}_k$ 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.

▶ Remark 3.2. Note that Rocq allows us to create types such as `ltt_send q []` which don't correspond to well-formed local types as the continuation is empty. In our implementation we define a predicate `wfltt : ltt → Prop` capturing that all the continuations in the local type tree are non-empty. Henceforth we assume that all local types we mention satisfy this property.

3.2 Subtyping

We define the subsorting relation on sorts and the process-oriented [16] subtyping relation on local type trees.

▶ **Definition 3.3** (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:*

$$\frac{}{\mathsf{end} \leqslant \mathsf{end}}\;[\textsc{sub-end}] \qquad \frac{\forall i \in I: \quad S_i' \leq S_i \quad \mathsf{T}_i \leqslant \mathsf{T}_i'}{\mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I\cup J} \leqslant \mathsf{p}\&\{\ell_i(S_i').\mathsf{T}_i'\}_{i\in I}}\;[\textsc{sub-in}]$$

$$\frac{\forall i \in I: \quad S_i \leq S_i' \quad \mathsf{T}_i \leqslant \mathsf{T}_i'}{\mathsf{p}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \leqslant \mathsf{p}\oplus\{\ell_i(S_i').\mathsf{T}_i'\}_{i\in I\cup J}}\;[\textsc{sub-out}]$$

Intutively, $\mathsf{T}_1 \leqslant \mathsf{T}_2$ 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 contraviance 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 [20], for details of we refer to [17].

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 [13].

▶ **Definition 3.4** (Global type trees). *We define global type trees coinductively as follows:*

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

```
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 $i \in I$, 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:

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

We extend the function pt onto trees by defining $\mathtt{pt}(\mathsf{G}) = \mathtt{pt}(\mathbb{G})$ where the global type $\mathbb{G}$ corresponds to the global type tree G. Technical details of this definition such as well-definedness can be found in [13, 17]. 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 [40] for a survey of other notions of merge).

▶ **Definition 3.5** (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* $\forall i \in I, \mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{T}_i$ [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* $\forall i \in I, \mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{T}_i$ [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]

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 as the relation `projectionC : gtt → part → ltt → Prop`.

We further have the following fact about projections that lets us regard it as a partial function:

▶ **Lemma 3.6** ([13]). *If* `projectionC G p T` *and* `projectionC G p T'` *then* `T = T'`.

We write $\mathsf{G} \upharpoonright \mathsf{r} = \mathsf{T}$ when $\mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{T}$. Furthermore we will be 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 [17] and [13].

▶ **Definition 3.7** (Balanced Global Type Trees). *A global 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' *of length at least* k*.*

[1] Here we adopt a simplified presentation as $\mathtt{pt}(\mathsf{G})$ is actually defined by extending it from an inductively defined function on syntactic types, we refer to [13] for details.

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.

▶ **Definition 3.8** (Global Type Tree Contexts and Grafting). *Global type tree contexts are defined inductively with the following syntax:*

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

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

Given a global type tree context $\mathcal{G}$ whose holes are in the indexing set I and a set of global types $\{\mathsf{G}_i\}_{i\in I}$, the grafting $\mathcal{G}[\mathsf{G}_i]_{i\in I}$ denotes the global type tree obtained by substituting $[\;]_i$ with G_i in $\mathcal{G}$.

In Rocq the indexed set $\{\mathsf{G}_i\}_{i\in I}$ 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 context* `gcx` *results in the tree* `gt`. *We additionally define* `pt` *and* `ishParts` *on global type tree contexts analogously to* `pt` *and* `isgPartsC` *on trees.*

A global type tree context can be thought of as the finite prefix of a global type tree, where holes $[\;]_i$ indicate the cutoff points. Global type tree contexts are related to global type trees with the *grafting* operation that fills in the holes with type trees. The following lemma relates global type tree contexts to balanced global type trees. In particular, it allows us to turn proofs by coinduction on infinite trees to proofs by induction on the grafting context.

▶ **Lemma 3.9** (Proper Grafting Lemma, [13]). *If* `G` *is a balanced global type tree and* `isgPartsC p G`, *then there is a global type tree context* `Gctx` *and an option list of global type trees* `gs` *such that* `typ_gtth gs Gctx G`, `~ ishParts p Gctx` *and every* `Some` *element of* `gs` *is of shape* `gtt_end`, `gtt_send p q` *or* `gtt_send q p`. *We refer to* `Gctx` *and* `gs` *as the* `p`*-grafting of* `G`. *When we don't care about* `gs` *we may just say that* `G` *is* `p`*-grafted by* `Gctx`.

▶ Remark 3.10. 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 some global type and that `G` is balanced. 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 Types

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

4.1 Local Type Contexts and Reductions

We start by defining typing contexts as finite mappings of participants to local type trees.

▶ **Definition 4.1** (Typing Contexts).

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

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

Intuitively, p : 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 occuring in Γ. 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 typing contexts as finite maps of participants, which are represented as natural numbers, and local type trees. We use the red-black tree based finite map implementation of the MMaps library [27].

▶ Remark 4.2. From now on, we assume the all the types in the local type contexts always have non-empty continuations. In Rocq terms, if `T` is in context `gamma` then `wfltt T` holds. This is expressed by the predicate `tctx_wf: tctx → Prop`.

We now give LTS semantics to local typing contexts, for which we first define the transition labels.

▶ **Definition 4.3** (Transition labels). *A transition label α has the following form:*

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

Next we define labelled transitions for local type contexts.

▶ **Definition 4.4** (Typing context reductions). *The typing context 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 $\Gamma \to \Gamma'$ that holds 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 $\to$.

$[\,\Gamma\text{-}\oplus]$ and $[\,\Gamma\text{-}\&]$, express a single participant sending or receiving. $[\Gamma\text{-}\oplus\&]$ expresses a synchronised communication where one participant sends while another receives, and they both progress with their continuation. $[\Gamma\text{-},]$ shows how to extend a context.

In Rocq typing context reductions are defined with the predicate `tctxR`.

```
Notation opt_lbl ≜ nat.
Inductive label: Type ≜
  | lrecv: part → part → option sort → opt_lbl → label
  | lsend: part → part → option sort → opt_lbl → label
  | lcomm: part → part → opt_lbl → label.
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` corresponds to the rules in Definition 4.4, and `Rstruct` expresses the indistinguishability of local contexts under the `M.Equal` predicate from the MMaps library.

We illustrate typing context reductions with an example.

▶ **Example 4.5.** Let $\Gamma = \{\mathsf{p} : \mathsf{T_p},\ \mathsf{q} : \mathsf{T_q},\ \mathsf{r} : \mathsf{T_r}\}$ where $\mathsf{T_p} = \mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathsf{T_p}, \ell_1(\mathtt{int}).\mathtt{end}\}$ $\mathsf{T_q} = \mathsf{p}\&\{\ell_0(\mathtt{int}).\mathsf{T_q}, \ell_1(\mathtt{int}).\mathsf{r}\oplus\{\ell_2(\mathtt{int}).\mathtt{end}\}\}$ and $\mathsf{T_r} = \mathsf{q}\&\{\ell_2(\mathtt{int}).\mathtt{end}\}$. We have the reductions $\Gamma \xrightarrow{\mathsf{p:q}\oplus\ell_0(\mathtt{int})} \Gamma$ and $\Gamma \xrightarrow{\mathsf{q:p}\&\ell_0(\mathtt{int})} \Gamma$, 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 $\{\mathsf{p} : \mathtt{end},\ \mathsf{q} : \mathsf{r}\oplus\{\ell_2(\mathtt{int}).\mathtt{end}\}, \mathsf{r} : \mathsf{T_r}\}$. 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 (lsend prt_p prt_q (Some sint) 0) gamma`.

4.2 Global Type Reductions

As with local typing contexts, we can also define reductions for global types.

▶ **Definition 4.6** (Global type reductions). *The global type transition $\xrightarrow{\alpha}$ is defined coinductively as follows.*

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

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

[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 choosen 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 [13] for details.

4.3 Association Between Local Type Contexts and Global Types

We have defined local type contexts which specifies 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 now relate these local and global definitions by defining *association* between local type context and global types.

▶ **Definition 4.7** (Association). *A local typing context* Γ *is associated with a global type tree* G*, written* $\Gamma \sqsubseteq \mathsf{G}$*, if the following hold:*

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

In Rocq this is defined with the following:

```
Definition assoc (g: tctx) (gt:gtt) ≜
    ∀ p, (isgPartsC p gt  →  ∃ Tp, M.find p g=Some Tp ∧
       issubProj Tp gt p) ∧
        (~ isgPartsC p gt  →  ∀ Tpx, M.find p g = Some Tpx  →  Tpx=ltt_end).
```

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

▶ **Example 4.8.** In Example 4.5, we have that $\Gamma \sqsubseteq \mathsf{G}$ where $\mathsf{G} := \mathsf{p} \to \mathsf{q} : \{\ell_0(\mathtt{int}).\mathsf{G}, \ell_1(\mathtt{int}).\mathsf{q} \to \mathsf{r} : \{\ell_2(\mathtt{int}).\mathtt{end}\}\}$. 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 context and global type reductions, that is, when one of the associated constructs "takes a step" so should the other. We formalise this property with soundness and completeness theorems.

▶ **Theorem 4.9** (Soundness of Association). *If* `assoc gamma G` *and* `gttstepC G G' p q ell`*, then there is a local type context* `gamma'`*, a global type tree* `G''` *and a message label* `ell'` *such that* `gttStepC G G'' p q ell'`*,* `assoc gamma' G''` *and* `tctxR gamma (lcomm p q ell') gamma'`*.*

▶ **Theorem 4.10** (Completeness of Association). *If* `assoc gamma G` *and* `tctxR gamma (lcomm p q ell) gamma'`*, then there exists a global type tree* `G'` *such that* `assoc gamma' G'` *and* `gttstepC G G' p q ell`*.*

▶ Remark 4.11. Note that in the statement of soundness we allow the message label for the local type context reduction to be different to the message label for the global type reduction. This is because our use of subtyping in association causes the entries in the local type context 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.

5 Properties of Local Type Contexts

We now use the LTS semantics to define some desirable properties on type contexts and their reduction sequences. Namely, we formulate safety, fairness and liveness properties based on the definitions in [44].

5.1 Safety

We start by defining the *safety* property that plays an important role in bottom-up session type systems [35]:

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

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

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

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

> Safety says that if p and q 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 typing context reductions.

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 as we have $\Gamma \xrightarrow{\mathsf{p:q}\oplus\ell_0}$ and $\Gamma \xrightarrow{\mathsf{q:p}\&\ell_0}$ but we don't have $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0}$ as $\mathtt{int} \not\leq \mathtt{nat}$.

Consider Γ from Example 4.5. 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'  → R c')  →  safe R c.
Definition safeC c ≜ paco1 safe bot1 c.
```

`weak_safety` corresponds [S-&⊕] where `tctxRE l c` is shorthand for `∃ c', tctxR c l c'`. In the inductive `safe`, the constructor `safety_red` corresponds to [S-→] . Then `safeC` is defined as the greatest fixed point of `safe`.

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

▶ **Theorem 5.3** (Safety by Association). *If* `assoc gamma g` *then* `safeC gamma`.

5.2 Fairness and Liveness

We now focus our attention to fairness and liveness. We first restate the definition of fairness and liveness for local type context paths from [44].

▶ **Definition 5.4** (Fair, Live Paths). *A local type context reduction path (also called executions or runs) 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 context reduction path $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_2} ..$ *is fair if, for all* $n \in N : \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ *implies* $\exists k, \ell'$ *such that* $N \ni k \geq n$ *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. $\forall n \in N : \Gamma_n \xrightarrow{\mathsf{p:q}\oplus\ell(S)}$ *implies* $\exists k, \ell'$ *such that* $N \ni k \geq n$ *and* $\Gamma_k \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_{k+1}$
2. $\forall n \in N : \Gamma_n \xrightarrow{\mathsf{q:p}\&\ell(S)}$ *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 Context). *A local type context* Γ *is live if whenever* $\Gamma \to^* \Gamma'$*, every fair path starting from* Γ' *is also live.*

In general, fairness assumptions are used so that only the reduction sequences that are "well-behaved" in some sense are considered when formulating other properties [42]. We define fairness such that, in a fair path, whenever a synchronous transition $(\mathsf{p}, \mathsf{q})\ell$ is enabled, a communication between p and q is eventually executed. Then live paths are defined to be paths such that whenever p attempts to send to q *or* q attempts to receive from p, eventually a p to q communication takes place. Informally, this means that every communication request is eventually answered. Live typing contexts 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 contexts Γ, Γ' and Γ_{end} from Example 4.5. One possible reduction path is $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \ldots$. 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 $\Gamma_1 \xrightarrow{\mathsf{r}:\mathsf{q}\&\ell_2(\mathsf{int})}$ 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 context.

Now consider the reduction path $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma' \xrightarrow{(\mathsf{q},\mathsf{r})\ell_2} \Gamma_{\mathsf{end}}$. 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 the existential statements it contains. It would be ideal 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 context paths in Linear Temporal Logic (LTL) [33]. The LTL operators *eventually* ($\Diamond$) and *always* ($\Box$) can be characterised as least and greatest fixed points using their expansion laws [2, Chapter 5.14]. Hence they can be 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 contexts 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 3.7). Indeed association with non-balanced global types doesn't guarantee liveness. As an example, consider Γ from Example 4.5, which is associated with G from Example 4.8. Yet we have shown in Example 5.6 that Γ is not a live type context. This is not surprising as G is not balanced.

▶ **Theorem 5.8** (Liveness by Association). *If* `assoc gamma g` *then* `gamma` *is live.*

Proof. (Simplified, Outline) Our proof proceeds in two steps. First, we prove that the typing context obtained by direct projections [2] of g, that is, `gamma_proj` = $\{\mathsf{p}_i : \mathsf{G} \upharpoonright_{\mathsf{p}_i} \mid \mathsf{p}_i \in \mathtt{pt}\{\mathsf{G}\}\}$, is live. We then leverage Theorem 4.10 to show that if `gamma_proj` is live, so is `gamma`.

Suppose `gamma_proj` $\xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)}$ (the case for the receive is similar and omitted), and `xs` is a fair local type context reduction path beginning with `gamma_proj`. To show that `xs` is live we need to show the existence of a $(\mathsf{p},\mathsf{q})\ell$ transition in `xs`. We achieve this by taking the height of the p-grafting of the global type associated with the head of `xs` as our induction invariant. We show (, ,) that this invariant keeps decreasing until a $(\mathsf{p},\mathsf{q})\ell$ transition is enabled on the path, at which point our fairness assumption forces that transition to fire .

In the second step of the proof we extend association on to paths to get, for each local type context reduction path `xs` that begins with `gamma`, another local type context reduction path `ys` beginning with `gamma_proj` such that the elements of `xs` are subtypes (subtyping on contexts defined pointwise) of the corresponding elements of `ys`. This is obtained from Theorem 4.10, however the statement of Theorem 4.10 is implemented as an ∃ statement that lives in `Prop`, hence we need to use the `constructive_indefinite_description` axiom to construct a `CoFixpoint` returning the desired cosequence `ys`. The proof then follows by the definition of subtyping (Definition 3.3). ◀

6 Properties of Sessions

We give typing rules for the session calculus introduced in 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 context 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 [17] and [13]. We have two kinds of typing judgements and type contexts. $\Theta_{\mathsf{T}}, \Theta_{\mathsf{e}} \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}$ says that the single process P can be typed with local type T using expression and type variables from $\Theta_{\mathsf{T}}, \Theta_{\mathsf{e}}$. On the other hand, $\Gamma \vdash_{\mathcal{M}} \mathcal{M}$ expresses that session $\mathcal{M}$ can be typed by the local type context (Definition 4.1) Typing rules for expressions are standard and can be found in e.g. [17], and are therefore omitted. Γ.

$$\frac{}{\Theta \vdash_{\mathsf{P}} \mathbf{0} : \mathtt{end}}\;[\text{T-END}] \qquad \frac{}{\Theta, \mathbf{X} : \mathsf{T} \vdash_{\mathsf{P}} \mathbf{X} : \mathsf{T}}\;[\text{T-VAR}] \qquad \frac{\Theta, \mathbf{X} : \mathsf{T} \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}}{\Theta \vdash_{\mathsf{P}} \mu\mathbf{X}.\mathsf{P} : \mathsf{T}}\;[\text{T-REC}] \qquad \frac{\Theta \vdash_{\mathsf{P}} e : \mathtt{bool} \quad \Theta \vdash_{\mathsf{P}} \mathsf{P}_1 : \mathsf{T} \quad \Theta \vdash_{\mathsf{P}} \mathsf{P}_2 : \mathsf{T}}{\Theta \vdash_{\mathsf{P}} \mathtt{if}\ e\ \mathtt{then}\ \mathsf{P}_1\ \mathtt{else}\ \mathsf{P}_2 : \mathsf{T}}\;[\text{T-IF}]$$

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

Table 3 Typing processes

[2] Note that 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.

Table 3 states the standard [13, 17] typing rules for processes, which we don't elaborate on. We additionally have a single rule for typing sessions:

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

[T-SESS] says 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 context Γ if the local type of participant p_i in Γ types the process

6.2 Properties of Typed Sessions

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

▶ **Lemma 6.1** (Typing after Unfolding). *If* `gamma` $\vdash_{\mathcal{M}}$ `M` *and* `M` $\Rrightarrow$ `M'` *then* `typ_sess M' gamma`.

▶ **Theorem 6.2** (Subject Reduction). *If* `gamma` $\vdash_{\mathcal{M}}$ `M` *and* `M` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ `M'`*, then there exists a typing context* `gamma'` *such that* `gamma` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ `gamma'` *and* `gamma` $\vdash_{\mathcal{M}}$ `M'` .

▶ **Theorem 6.3** (Session Fidelity). *If* `gamma` $\vdash_{\mathcal{M}}$ `M` *and* `gamma` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ `gamma'`*, there exists a message label* ℓ'*, a context* `gamma''` *and a session* `M'` *such that* `M` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell'}$ `M'` , `gamma` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell'}$ `gamma''` *and* `typ_sess M' gamma''`.

> Lemma 6.1 says that typing is preserved after unfolding. Theorem 6.2 shows that the typing context reduces along with the session it types. Theorem 6.3 is an analogue of Theorem 6.2 in the opposite direction.

▶ Remark 6.4. Note that in Theorem 6.2 one transition between sessions corresponds to exactly one transition between local type contexts 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* [29], while it turns into a strong simulation with reactive semantics. For our Rocq implementation working with the strong simulation turns out be more convenient.

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

▶ **Theorem 6.5** (Communication Safety). *If* `gamma` $\vdash_{\mathcal{M}}$ `M` *and* `M` $\rightarrow^*$ `M'` $\Rrightarrow$ `(p` $\leftarrow$ `p_send q ell P ||| q` $\leftarrow$ `p_recv p xs ||| M'')`*, then* `onth ell xs` $\neq$ `None`.

> Theorem 6.5 means that typed sessions evolve to sessions where if participant `p` wants to send to `q` with label ℓ, and `q` is listening to receive from `p`, then `q` is able to receive with label ℓ.

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

1. *Either* `M` $\Rrightarrow$ `M_inact` *where every process making up* `M_inact` *is inactive, i.e.* `M_inact` $\equiv \prod_{i=1}^{n} \mathsf{p}_i \triangleleft \mathbf{0}$ *for some* n.
2. *Or there is a* `M'` *such that* `M` $\rightarrow$ `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). *Session $\mathcal{M}$ is live iff*

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

In Rocq this is expressed with the predicate `live_sess`*:*

```
Definition live_sess Mp ≜ ∀ M, betaRtc Mp M  ⟶
  (∀ p q ell e P' M', p ≠q  ⟶  unfoldP M ( (p ⟵- p_send q ell e P') ||| M')  ⟶  ∃ M'',
  betaRtc M ((p ⟵- P')\|\|\|M''))
  ∧
  (∀ p  q llp M', p ≠q  ⟶  unfoldP M ( (p ⟵- p_recv q llp) ||| M')  ⟶
    ∃ M'' P' e k, onth k llp = Some P' ∧ betaRtc M ((p ⟵- subst_expr_proc P' e 0 0) |||M'')).
```

Session liveness, analogous to liveness for typing contexts (Definition 5.4), 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 related work ([41, 30]).

We now detail the proof that typed sessions are live. First we prove the following lemma:

▶ **Lemma 6.8** (Fair Extension of Typed Sessions). *If* `typ_sess M gamma`*, then there exists a session reduction path* `xs` *starting from* `M` *such that the following fairness property holds:*

- *On* `xs`*, 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'*.*

Proof. The desired path can be constructed by repeatedly cycling through all participants, checking if there is a transition involving that participant, and executing that transition if there is. As in the proof of Theorem 5.8, the construction in Lemma 6.8 uses the `constructive_indefinite_description` axiom to construct a cosequence as a `CoFixpoint`. Additionally, we use the axiom `excluded_middle_informative` for the "check if there is a transition involving a participant" part of the scheduling algorithm. The use of this axiom is probably not necessary but it makes the proof easier. Correctness of the algorithm follows from Theorem 6.2 and Theorem 6.3. ◀

Lemma 6.8 defines a "fairness" property for sessions analogous to Definition 5.4. It then shows that there exists a fair path from any typable session. This resembles the *feasibility* property expected from sensible notions of fairness [42], which states that any partial path can be extended into a fair one [3].

▶ **Theorem 6.9** (Liveness by Typing). *For a session* `Mp`*, if* `∃ gamma gamma` $\vdash_{\mathcal{M}}$ `Mp` *then* `live_sess Mp`.

[3] Note that this fairness property for sessions is not actually feasible as there are partial paths starting with an untypable session that can't be extended into a fair one. Nevertheless, Lemma 6.8 turns out to be enough to prove our liveness property.

Proof. We detail the proof for the send case of Definition 6.7, the case for the receive is similar. Suppose that `Mp` $\to^*$ `M` and `M` $\Rrightarrow$ `((p` $\leftarrow$ `p_send q ell e P’) ||| M’)`. Our goal is to show that there exists a `M’’` such that `M` $\to^*$ `((p` $\leftarrow$ `P’)|||M’’)`. First, observe that by [R-UNFOLD] it suffices to show that `((p` $\leftarrow$ `p_send q ell e P’) ||| M’)` $\to^*$ `M’’` for some `M’’`. Also note that `gamma` $\vdash_{\mathcal{M}}$ `M` for some `gamma` by Theorem 6.2, therefore `gamma` $\vdash_{\mathcal{M}}$ `((p` $\leftarrow$ `p_send q ell e P’) ||| M’)` by Lemma 6.1.

Now let `xs` be a fair session reduction path starting from `((p` $\leftarrow$ `p_send q ell e P’) ||| M’)`, which exists by Lemma 6.8. By Theorem 6.2, let `ys` be a local type context reduction path starting with `gamma` such that every session in `xs` is typed by the context at the corresponding index of `ys`, and the transitions of `xs` and `ys` at every step match. Now it can be shown that `ys` is fair. Therefore by Theorem 5.8 `ys` is live, so a `lcomm p q ell’` transition eventually occurs in `ys` for some `ell’`. Therefore `ys` $=$ `gamma` $\to^*$ `gamma_0` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell'}$ `gamma_1` $\to$ .. for some `gamma_0`, `gamma_1`. Now consider the session `M_0` typed by `gamma_0` in `xs`. We have `((p` $\leftarrow$ `p_send q ell e P’) ||| M’’)` $\to^*$ `M_0` by `M_0` being on `xs`. We also have that `M_0` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell''}$ `M_1` for some ℓ'', `M_1` by Theorem 6.3. Now observe that `M_0` $\equiv$ `((p` $\leftarrow$ `p_send q ell e P’) ||| M’’)` for some `M’’` as no transitions involving `p` have happened on the reduction path to `M_0`. Therefore $\ell = \ell''$, so `M_1` $\equiv$ `((p` $\leftarrow$ `P’) ||| M’’)` for some `M’’`, as needed. ◀

7 Conclusion and Related 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 simple message-passing calculus. To our knowledge, our liveness result is the first mechanised one of its kind, and 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. In particular, the induction on the tree context height used in Theorem 5.8 requires some care to set up, and is not the most obvious way of implementing the proof in Rocq. Our earlier unsuccesful attempts at that proof included one which proceeded by induction on the grafting (Definition 3.8) of local type trees, which turned out to be a defective induction variable. Still, our work illustrates 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.

Related Work. Examinations of liveness, also called *lock-freedom*, guarantees of multiparty session types abound in literature, e.g. [31, 23, 44, 35, 3]. Most of these papers use the definition liveness proposed by Padovani [30], which doesn’t make the fairness assumptions that characterize the property [15] explicit. Contrastingly, van Glabbeek et. al. [41] examine several notions of fairness and the liveness properties induced by them, and devise a type system with flexible choices [6] that captures the strongest of these properties, the one induced by the *justness* [42] assumption. In their terminology, Definition 6.7 corresponds to liveness under strong fairness of transitions (ST), 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. Indeed, our type system is not complete for Definition 6.7, even if we restrict our attention to safe and race-free sessions. For example, the session described in [41, Example 9] is live but not typable by a context associated with a balanced global type in our system.

Fairness assumptions are also made explicit in recent work by Ciccone et. al [10, 11] which use generalized inference systems with coaxioms [1] to characterize *fair termination*, which is stronger 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. [13] which uses a coinductive representation of global and local types to prove subject reduction and progress. Their work uses a typing relation between global types and sessions while ours uses one between associated local type contexts and sessions. This necessiates the rewriting of subject reduction and progress proofs in addition to the novel operational correspondence, safety and liveness properties we have proved. Other recent results mechanised in Rocq include Ekici and Yoshida's [14] work on the completeness of asynchronous subtyping, and Tirore's work [37, 39, 38] on projections and subject reduction for π-calculus.

Castro-Perez et. al. [8] 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. Ciccone's PhD thesis [9] presents an Agda formalisation of fair termination for binary session types. Binary session types were also implemented in Agda by Thiemann [36] and in Idris by Brady[5]. Several implementations of binary session types are also present for Haskell [24, 28, 34].

Implementations of session types that are more geared towards practical verification include the Actris framework [18, 21] which enriches the seperation logic of Iris [22] with binary session types to certify deadlock-freedom. In general, verification of liveness properties, with or without session types, in concurrent seperation logic is an active research area that has produced tools such as TaDa [12], FOS [25] and LiLo [26] in the past few years. Further verification tools employing multiparty session types are Jacobs's Multiparty GV [21] based on the functional language of Wadler's GV [43], and Castro-Perez et. al's Zooid [7], which supports the extraction of certifiably safe and live protocols.

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, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
2. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The 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. arXiv:2308.10653 [cs]. URL: http://arxiv.org/abs/2308.10653, doi:10.4204/EPTCS.383.2.
4. Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.
5. Edwin Charles Brady. Type-driven Development of Concurrent Communicating Systems. *Computer Science*, 18(3), July 2017. URL: https://journals.agh.edu.pl/csci/article/view/1413, doi:10.7494/csci.2017.18.3.1413.
6. 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.
7. 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, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454041.

8 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.

9 Luca Ciccone. Concerto grosso for sessions: Fair termination of sessions, 2023. URL: https://arxiv.org/abs/2307.05539, arXiv:2307.05539.

10 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 139:100964, 2024. URL: https://www.sciencedirect.com/science/article/pii/S2352220824000221, doi:10.1016/j.jlamp.2024.100964.

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

12 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.

13 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, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.19, doi:10.4230/LIPIcs.ITP.2025.19.

14 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, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.13, doi:10.4230/LIPIcs.ITP.2024.13.

15 Nissim Francez. *Fairness*. Springer US, New York, NY, 1986. URL: http://link.springer.com/10.1007/978-1-4612-4886-6, doi:10.1007/978-1-4612-4886-6.

16 Simon J. Gay. *Subtyping Supports Safe Session Substitution*, pages 95–108. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-30936-1_5.

17 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. URL: https://www.sciencedirect.com/science/article/pii/S2352220817302237, doi:10.1016/j.jlamp.2018.12.002.

18 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.

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

20 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.

21 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.

22 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.

23 Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177(2):122–159, September 2002. URL: https://www.sciencedirect.com/science/article/pii/S0890540102931718, doi:10.1006/inco.2002.3171.

24 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, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471874.3472979.

25 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.

26 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.

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

28 Sam Lindley and J Garrett Morris. Embedding session types in haskell. *ACM SIGPLAN Notices*, 51(12):133–145, 2016.

29 Robin MILNER. Chapter 19 - 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, Amsterdam, 1990. URL: https://www.sciencedirect.com/science/article/pii/B978044488074150024X, doi:10.1016/B978-0-444-88074-1.50024-X.

30 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, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603116.

31 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, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

32 Kai Pischke and Nobuko Yoshida. *Asynchronous Global Protocols, Precisely*, pages 116–133. Springer Nature Switzerland, Cham, 2026. doi:10.1007/978-3-031-99717-4_7.

33 Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.

34 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.

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

36 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, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3354166.3354184.

37 Dawit Tirore. A mechanisation of multiparty session types, 2024.

38 Dawit Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*, pages 28–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.

39 Dawit Tirore, Jesper Bengtson, and Marco Carbone. Multiparty asynchronous session types: A mechanised proof of subject reduction. In *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, pages 31–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

40 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.

41 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, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1109/LICS52264.2021.9470531.

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

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

44 Nobuko Yoshida and Ping Hou. Less is more revisited, 2024. URL: https://arxiv.org/abs/2402.16741, arXiv:2402.16741.