

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 *safety* (the labels and types of senders' payloads cohere with the capabilities of the receivers), *deadlock-freedom* (also called *progress* or *non-stuck property* [14]) (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:

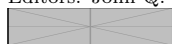


© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

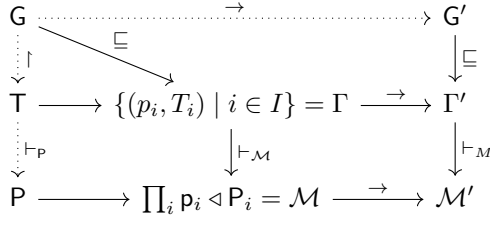
42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Design overview. The dotted lines correspond to relations inherited from [14] 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 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 the following properties:

1. **Subject Reduction** (Theorem 6.2): If \mathcal{M} can progress into \mathcal{M}' , then Γ can progress into Γ' such that Γ' types \mathcal{M}' .
2. **Session Fidelity** (Theorem 6.5): If Γ can progress into Γ' , then \mathcal{M} can progress into \mathcal{M}' such that \mathcal{M}' is typable by Γ' .
3. **Safety** (Theorem 6.7): If \mathcal{M} can progress into \mathcal{M}' by one or more communications, participant p in \mathcal{M}' sends to participant q and q receives from p , then the labels of p and q cohere.
4. **Deadlock-Freedom** (Theorem 6.4): Either every participant in \mathcal{M} has terminated, or \mathcal{M} can progress.
5. **Liveness** (Theorem 6.11): If participant p attempts to communicate with participant q in \mathcal{M} , then \mathcal{M} can progress (in possibly multiple steps) into a session \mathcal{M}' where that communication has occurred.

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. [14], which itself is based on [17]. The methodology in [14] 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 [14] 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 [14], 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 using linear temporal logic (LTL)[33]. The LTL modalities eventually (\diamond) and always (\Box) can be expressed as least and greatest fixpoints respectively using expansion laws. This allows us to represent the properties that use these modalities as inductive and coinductive predicates in Rocq. 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 introduce local and global type trees. 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:*

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

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`, \neg , non-deterministic choice \oplus and $>$.

$p!\ell(e).P$ is a process that sends the value of expression e with label ℓ to participant p , and continues with process P . $\sum_{i \in I} p?\ell_i(x_i).P_i$ 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. X is a recursion variable, $\mu X.P$ is a recursive process, if e then P else P is a conditional and 0 is a terminated process.

Processes can be composed in parallel into sessions.

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

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

$p \triangleleft P$ denotes that participant p is running the process P , \mid indicates parallel composition.

We write $\prod_{i \in I} p_i \triangleleft P_i$ 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).
```

117 2.2 Structural Congruence and Operational Semantics

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

$$\begin{array}{l}
\text{[SC-SYM]} \quad p \triangleleft P \mid q \triangleleft Q \equiv q \triangleleft Q \mid p \triangleleft P \quad \text{[SC-ASSOC]} \quad (p \triangleleft P \mid q \triangleleft Q) \mid r \triangleleft R \equiv p \triangleleft P \mid (q \triangleleft Q \mid r \triangleleft R) \\
\text{[SC-O]} \quad p \triangleleft P \mid \mathcal{O} \equiv p \triangleleft P
\end{array}$$

■ **Table 1** Structural Congruence over Sessions

120 We omit the semantics for expressions, they are standard and can be found in e.g. [17].
121 We now give the operational semantics for sessions by the means of a labelled transition
122 system. We use labelled *reactive* semantics [41, 6] which doesn't contain explicit silent τ
123 actions for internal reductions (that is, evaluation of if expressions and unfolding of recursion)
124 while still considering β reductions up to those internal reductions by using an unfolding
125 relation. This stands in contrast to the more standard semantics used in [14, 17, 41]. For
126 the advantages of our approach see Remark 6.3.

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

$$\begin{array}{l}
\text{[UNF-STRUCT]} \quad \frac{\mathcal{M} \equiv \mathcal{N}}{\mathcal{M} \Rightarrow \mathcal{N}} \quad \text{[UNF-REC]} \quad p \triangleleft \mu \mathbf{X}.P \mid \mathcal{N} \Rightarrow p \triangleleft P[\mu \mathbf{X}.P/\mathbf{X}] \mid \mathcal{N} \quad \text{[UNF-CONDT]} \quad \frac{e \downarrow \text{true}}{p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathcal{N} \Rightarrow p \triangleleft P \mid \mathcal{N}} \\
\text{[UNF-CONDF]} \quad \frac{e \downarrow \text{false}}{p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathcal{N} \Rightarrow p \triangleleft Q \mid \mathcal{N}} \quad \text{[UNF-TRANS]} \quad \frac{\mathcal{M} \Rightarrow \mathcal{M}' \quad \mathcal{M}' \Rightarrow \mathcal{N}}{\mathcal{M} \Rightarrow \mathcal{N}}
\end{array}$$



■ **Table 2** Unfolding of Sessions

128 $\mathcal{M} \Rightarrow \mathcal{N}$ means that \mathcal{M} can transition to \mathcal{N} through some internal actions, that is, a
129 reduction that doesn't involve a communication. We say that \mathcal{M} *unfolds* to \mathcal{N} . In Rocq it's
130 captured by the predicate `unfoldP : session → session → Prop` 🐼.

$$\begin{array}{l}
\text{[R-COMM]} \quad \frac{j \in I \quad e \downarrow v}{p \triangleleft \sum_{i \in I} q! \ell_i(x_i).P_i \mid q \triangleleft p! \ell_j(e).Q \mid \mathcal{N} \xrightarrow{(p,q)\ell_j} p \triangleleft P_j[v/x_j] \mid q \triangleleft Q \mid \mathcal{N}} \\
\text{[R-UNFOLD]} \quad \frac{\mathcal{M} \Rightarrow \mathcal{M}' \quad \mathcal{M}' \xrightarrow{\lambda} \mathcal{N}' \quad \mathcal{N}' \Rightarrow \mathcal{N}}{\mathcal{M} \xrightarrow{\lambda} \mathcal{N}}
\end{array}$$

■ **Table 3** Reactive Semantics of Sessions

131 Table 3 illustrates the rules for communicating transits. [R-COMM] captures commu-
132 nications between processes, and [R-UNFOLD] lets us consider reductions up to unfoldings.
133

134 In Rocq, `betaP_lbl M lambda M'`  denotes $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$. We write $\mathcal{M} \rightarrow \mathcal{M}'$ if $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$ for
 135 some λ , which is written `betaP M M'` in Rocq. We write \rightarrow^* to denote the reflexive transitive
 136 closure of \rightarrow , which is called `betaRtc`  in Rocq.

137 3 The Type System

138 We briefly recap the core definitions of local and global type trees, subtyping and projection
 139 from [17]. We take an equirecursive approach and work directly on the possibly infinite local
 140 and global type trees obtained by unfolding the recursion in guarded syntactic types, details
 141 of this approach can be found in [14] and hence are omitted here.

142 3.1 Local Type Trees

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

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

147
$$\begin{aligned} T ::= & \text{end} \\ & | p\&\{\ell_i(S_i).T_i\}_{i \in I} \\ & | p\oplus\{\ell_i(S_i).T_i\}_{i \in I} \end{aligned}$$

```
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.
```

148 In the above definition, `end` represents a role that has finished communicating.
 149 $p\oplus\{\ell_i(S_i).T_i\}_{i \in I}$ denotes a role that may, from any $i \in I$, receive a value of sort S_i with
 150 message label ℓ_i and continue with T_i . Similarly, $p\&\{\ell_i(S_i).T_i\}_{i \in I}$ represents a role that may
 151 choose to send a value of sort S_i with message label ℓ_i and continue with T_i for any $i \in I$.

152 In Rocq we represent the continuations using a `list` of `option` types. In a continuation
 153 `gcs : list (option(sort*ltt))`, index `k` (using zero-indexing) being equal to `Some (s_k,`
 154 `T_k)` means that $\ell_k(S_k).T_k$ is available in the continuation. Similarly index `k` being equal to
 155 `None` or being out of bounds of the list means that the message label ℓ_k is not present in the
 156 continuation.

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

162 3.2 Subtyping

163 We define the subsorting relation on sorts and the subtyping relation on local type trees.

164 ► **Definition 3.3** (Subsorting and Subtyping). *Subsorting \leq is the least reflexive binary
 165 relation that satisfies $\text{nat} \leq \text{int}$. Subtyping \leq is the largest relation between local type trees*

166 *coinductively defined by the following rules:*

$$\begin{array}{c}
 \text{167} \quad \frac{}{\text{end} \leq \text{end}} \text{ [SUB-END]} \quad \frac{\forall i \in I : \quad S'_i \leq S_i \quad T_i \leq T'_i}{p \& \{\ell_i(S_i).T_i\}_{i \in I \cup J} \leq p \& \{\ell_i(S'_i).T'_i\}_{i \in I}} \text{ [SUB-IN]} \\
 \\
 \frac{\forall i \in I : \quad S_i \leq S'_i \quad T_i \leq T'_i}{p \oplus \{\ell_i(S_i).T_i\}_{i \in I} \leq p \oplus \{\ell_i(S'_i).T'_i\}_{i \in I \cup J}} \text{ [SUB-OUT]}
 \end{array}$$

168 Intutively, $T_1 \leq T_2$ means that a role of type T_1 can be supplied anywhere a role of type T_2
 169 is needed. [SUB-IN] captures the fact that we can supply a role that is able to receive more
 170 labels than specified, and [SUB-OUT] captures that we can supply a role that has fewer labels
 171 available to send. Note the contravariance of the sorts in [SUB-IN], if the supertype demands
 172 the ability to receive an **nat** then the subtype can receive **nat** or **int**.

173 In Rocq, the subtyping relation `subtypeC : ltt → ltt → Prop` is expressed as a greatest
 174 fixpoint using the `Paco` library [20], for details of we refer to [17].

175 3.3 Global Types and Type Trees

176 We now define global types which give a bird's eye view of the whole protocol. As before, we
 177 work directly on infinite trees and omit the details which can be found in [14]. **end** denotes
 178 a protocol that has ended, $p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$ denotes a protocol where for any $i \in I$,
 179 participant p may send a value of sort S_i to another participant q via message label ℓ_i , after
 180 which the protocol continues as G_i .

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

182 $G ::= \text{end} \mid p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$

```

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

```

183 We further define the function `pt(G)` that denotes the participants of the global type G as
 184 the least solution ¹ to the following equations:

$$\begin{array}{l}
 \text{185} \quad \text{pt}(\text{end}) = \emptyset \\
 \text{186} \quad \text{pt}(p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}) = \{p, q\} \cup \bigcup_{i \in I} \text{pt}(G_i)
 \end{array}$$

187 We extend the function `pt` onto trees by defining $\text{pt}(G) = \text{pt}(\mathbb{G})$ where the global type
 188 \mathbb{G} corresponds to the global type tree G . Technical details of this definition such as well-
 189 definedness can be found in [14, 17].

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

192 3.4 Projection

193 We now define coinductive projections with plain merging (see [40] for a survey of other
 194 notions of merge).

¹ Here we adopt a simplified presentation as `pt(G)` is actually defined by extending it from an inductively defined function on syntactic types, we refer to [14] for details.

195 ► **Definition 3.5** (Projection). *The projection of a global type tree onto a participant r is the*
 196 *largest relation \vdash_r between global type trees and local type trees such that, whenever $G \vdash_r T$:*
 197 $\blacksquare r \notin \text{pt}\{G\}$ *implies* $T = \text{end}$; [PROJ-END]
 198 $\blacksquare G = p \rightarrow r : \{\ell_i(S_i).G_i\}_{i \in I}$ *implies* $T = p \& \{\ell_i(S_i).T_i\}_{i \in I}$ *and* $\forall i \in I, G \vdash_r T_i$ [PROJ-IN]
 199 $\blacksquare G = r \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$ *implies* $T = q \oplus \{\ell_i(S_i).T_i\}_{i \in I}$ *and* $\forall i \in I, G \vdash_r T_i$ [PROJ-OUT]
 200 $\blacksquare G = p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$ *and* $r \notin \{p, q\}$ *implies that there are* $T_i, i \in I$ *such that*
 201 $T = \prod_{i \in I} T_i$ *and* $\forall i \in I, G \vdash_r T_i$ [PROJ-CONT]
 202 *where \prod is the plain merging operator, defined as*

$$203 \quad T_1 \prod T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

204 Informally, the projection of a global type tree G onto a participant r extracts a specification
 205 for participant r from the protocol whose bird's-eye view is given by G . [PROJ-END]
 206 expresses that if r is not a participant of G then r does nothing in the protocol. [PROJ-IN]
 207 and [PROJ-OUT] handle the cases where r is involved in a communication in the root of G .
 208 [PROJ-CONT] says that, if r is not involved in the root communication of G , then the only
 209 way it knows its role in the protocol is if there is a role for it that works no matter what
 210 choices p and q make in their communication. This "works no matter the choices of the other
 211 participants" property is captured by the merge operations.

212 In Rocq, projection is defined as a Paco greatest fixpoint as the relation `projectionC` :
 213 `gtt → part → ltt → Prop`.

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

216 ► **Lemma 3.6** ([14]). *If `projectionC G p T` and `projectionC G p T'` then $T = T'$.*

217 We write $G \vdash_r T$ when $G \vdash_r T$. Furthermore we will be frequently be making assertions
 218 about subtypes of projections of a global type e.g. $T \leq G \vdash_r$. In our Rocq implementation
 219 we define the predicate `issubProj : ltt → gtt → part → Prop` as a shorthand for this.

220 3.5 Balancedness, Global Tree Contexts and Grafting

221 We introduce an important constraint on the types of global type trees we will consider,
 222 balancedness.

223 ► **Definition 3.7** (Balanced Global Type Trees). *A global tree G is balanced if for any subtree*
 224 *G' of G , there exists k such that for all $p \in \text{pt}(G')$, p occurs on every path from the root of*
 225 *G' of length at least k .*

226 We omit the technical details of this definition and the Rocq implementation, they can be
 227 found in [17] and [14].

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

233 ► **Definition 3.8** (Global Type Tree Context). *Global type tree contexts are defined inductively*
 234 *with the following syntax:*

235

$$\mathcal{G} ::= p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I} \mid []_i$$

```

Inductive gttth: Type :=
| gttth_hol   : fin → gttth

| gttth_send  : part → part → list (option (sort *
gttth)) → gttth.

```

236 We additionally define `pt` and `ishParts` on contexts analogously to `pt` and `isgPartsC` on
 237 trees.

238 A global type tree context can be thought of as the finite prefix of a global type tree, where
 239 holes $[]_i$ indicate the cutoff points. Global type tree contexts are related to global type trees
 240 with the grafting operation.

241 ► **Definition 3.9** (Grafting). *Given a global type tree context \mathcal{G} whose holes are in the*
 242 *indexing set I and a set of global types $\{G_i\}_{i \in I}$, the grafting $\mathcal{G}[G_i]_{i \in I}$ denotes the global type*
 243 *tree obtained by substituting $[]_i$ with G_i in \mathcal{G} .*

244 In Rocq the indexed set $\{G_i\}_{i \in I}$ is represented using a list `(option gtt)`. Grafting is
 245 expressed with the inductive relation `typ_gttth : list (option gtt) → gttth → gtt →`
 246 **Prop.** `typ_gttth gs gcx gt` means that the grafting of the set of global type trees `gs` onto the
 247 context `gcx` results in the tree `gt`.

248 Furthermore, we have the following lemma that relates global type tree contexts to
 249 balanced global type trees.

250 ► **Lemma 3.10** (Proper Grafting Lemma, [14]). *If G is a balanced global type tree and*
 251 *`isgPartsC p G`, then there is a global type tree context `Gctx` and an option list of global type*
 252 *trees `gs` such that `typ_gttth gs Gctx G`, \sim `ishParts p Gctx` and every `Some` element of `gs` is of*
 253 *shape `gtt_end`, `gtt_send p q` or `gtt_send q p`.*

254 3.10 enables us to represent a coinductive global type tree featuring participant `p` as the
 255 grafting of a context that doesn't contain `p` with a list of trees that are all of a certain
 256 structure. If `typ_gttth gs Gctx G`, \sim `ishParts p Gctx` and every `Some` element of `gs` is of shape
 257 `gtt_end`, `gtt_send p q` or `gtt_send q p`, then we call the pair `gs` and `Gctx` as the `p`-grafting
 258 of `G`, expressed in Rocq as `typ_p_gttth gs Gctx p G`. When we don't care about the contents
 259 of `gs` we may just say that `G` is `p`-grafted by `Gctx`.

260 ► **Remark 3.11.** From now on, all the global type trees we will be referring to are assumed
 261 to be balanced. When talking about the Rocq implementation, any `G : gtt` we mention
 262 is assumed to satisfy the predicate `wfgC G`, expressing that `G` corresponds to some global
 263 type and that `G` is balanced. Furthermore, we will often require that a global type is
 264 projectable onto all its participants. This is captured by the predicate `projectableA G = ∀`
 265 `p, ∃ T, projectionC G p T`. As with `wfgC`, we will be assuming that all types we mention
 266 are projectable.

267 4 Semantics of Types

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

4.1 Typing Contexts

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

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

$$\Gamma ::= \emptyset \mid \Gamma, p : T$$

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

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

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 $\text{wfltt } T$ holds. This is expressed by the predicate $\text{wfltt}: \text{tctx} \rightarrow \text{Prop}$.

4.2 Local Type Context Reductions

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:*

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

In Rocq they are defined as follows:

```
Notation opt_lbl  $\triangleq$  nat.
Inductive label: Type  $\triangleq$ 
| lrecv: part  $\rightarrow$  part  $\rightarrow$  option sort  $\rightarrow$  opt_lbl  $\rightarrow$  label
| lsend: part  $\rightarrow$  part  $\rightarrow$  option sort  $\rightarrow$  opt_lbl  $\rightarrow$  label
| lcomm: part  $\rightarrow$  part  $\rightarrow$  opt_lbl  $\rightarrow$  label.
```

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:*

$$\begin{aligned} & \frac{k \in I}{p : q \& \{\ell_i(S_i).T_i\}_{i \in I} \xrightarrow{p:q \& \ell_k(S_k)} p : T_k} [\Gamma-\&] \\ & \frac{k \in I}{p : q \oplus \{\ell_i(S_i).T_i\}_{i \in I} \xrightarrow{p:q \oplus \ell_k(S_k)} p : T_k} [\Gamma-\oplus] \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, p : T \xrightarrow{\alpha} \Gamma', p : T} [\Gamma-,] \\ & \frac{\Gamma_1 \xrightarrow{p:q \oplus \ell(S)} \Gamma'_1 \quad \Gamma_2 \xrightarrow{q:p \& \ell(S')} \Gamma'_2 \quad S \leq S'}{\Gamma_1, \Gamma_2 \xrightarrow{(p,q)\ell} \Gamma'_1, \Gamma'_2} [\Gamma-\oplus\&] \end{aligned}$$

23:10 Dummy short title

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

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

In Rocq typing context reductions are defined the following way:

```
Inductive tctxR: tctx → label → tctx → Prop ≜
| Rsend: ∀ p q xs n s T,
  p ≠ q →
  onth n xs = Some (s, T) →
  tctxR (M.add p (ltsend q xs) M.empty) (lsend p q (Some s) n) (M.add p T M.empty)
| Rrecv: ...
| Rcomm: ∀ p q g1' g2' s s' n (H1: MF.Disjoint g1 g2) (H2: MF.Disjoint g1' g2'),
  p ≠ q →
  tctxR g1 (lsend p q (Some s) n) g1' →
  tctxR g2 (lrecv q p (Some s') n) g2' →
  subort s s' →
  tctxR (disj_merge g1 g2 H1) (lcomm p q n) (disj_merge g1' g2' H2)
| RvarI: ∀ g l g' p T,
  tctxR g l g' →
  M.mem p g = false →
  tctxR (M.add p T g) l (M.add p T g')
| Rstruct: ∀ g1 g1' g2 g2' l, tctxR g1' l g2' →
  M.Equal g1 g1' →
  M.Equal g2 g2' →
  tctxR g1 l g2.
```

Rsend, **Rrecv** and **RvarI** are straightforward translations of $[\Gamma \&]$, $[\Gamma \oplus]$ and $[\Gamma \cdot]$. **Rcomm** captures $[\Gamma \oplus \&]$ using the **disj_merge** function we defined for the compositions, and requires a proof that the contexts given are disjoint to be applied. **Rstruct** captures the indistinguishability of local contexts under the **M.Equal** predicate from the **MMaps** library. We give an example to illustrate typing context reductions.

► **Example 4.5.** Let

$$\begin{aligned} T_p &= q \oplus \{\ell_0(\text{int}).T_p, \ell_1(\text{int}).\text{end}\} \\ T_q &= p \& \{\ell_0(\text{int}).T_q, \ell_1(\text{int}).r \oplus \{\ell_2(\text{int}).\text{end}\}\} \\ T_r &= q \& \{\ell_2(\text{int}).\text{end}\} \end{aligned}$$

and $\Gamma = \{p : T_p, q : T_q, r : T_r\}$. We have the reductions $\Gamma \xrightarrow{p:q \oplus \ell_0(\text{int})} \Gamma$ and $\Gamma \xrightarrow{q:p \& \ell_0(\text{int})} \Gamma$, which synchronise to give the reduction and $\Gamma \xrightarrow{(p,q)\ell_0} \Gamma$. Similarly via synchronised communication of p and q via message label ℓ_1 we get $\Gamma \xrightarrow{(p,q)\ell_1} \Gamma'$ where Γ' is defined as $\{p : \text{end}, q : r \oplus \{\ell_2(\text{int}).\text{end}\}, r : T_r\}$. We further have that $\Gamma' \xrightarrow{(q,r)\ell_2} \Gamma_{\text{end}}$ where Γ_{end} is defined as $\{p : \text{end}, q : \text{end}, r : \text{end}\}$.

In Rocq, Γ is defined the following way:

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

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

4.3 Global Type Reductions

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

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

$$\begin{array}{c}
 \frac{k \in I}{\frac{}{\frac{}{\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \xrightarrow{(p,q)\ell_k} G_k} \text{ [GR-}\oplus\&]}} \\
 \frac{\forall i \in I \ G_i \xrightarrow{\alpha} G'_i \quad \text{subject}(\alpha) \cap \{\mathbf{p}, \mathbf{q}\} = \emptyset \quad \forall i \in I \ \{\mathbf{p}, \mathbf{q}\} \subseteq \text{pt}\{G_i\}}{\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \xrightarrow{\alpha} \mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G'_i\}_{i \in I}} \text{ [GR-CTX]}
 \end{array}$$

326 [GR- $\oplus\&$] says that a global type tree with root $\mathbf{p} \rightarrow \mathbf{q}$ can transition to any of its children
 327 corresponding to the message label choosen by \mathbf{p} . [GR-CTX] says that if the subjects of α
 328 are disjoint from the root and all its children can transition via α , then the whole tree can
 329 also transition via α , with the root remaining the same and just the subtrees of its children
 330 transitioning.

331 In Rocq global type reductions are expressed using the coinductively defined predicate
 332 `gttstepC`. For example, $G \xrightarrow{(p,q)\ell_k} G'$ translates to `gttstepC G G' p q k`. We refer to [14] for
 333 details.

334 4.4 Association Between Local Type Contexts and Global Types

335 We have defined local type contexts which specifies protocols bottom-up by directly describing
 336 the roles of every participant, and global types, which give a top-down view of the whole
 337 protocol, and the transition relations on them. We now relate these local and global definitions
 338 by defining *association* between local type context and global types.

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

- 341 ■ For all $\mathbf{p} \in \text{pt}(G)$, $\mathbf{p} \in \text{dom}(\Gamma)$ and $\Gamma(\mathbf{p}) \leq G \upharpoonright \mathbf{p}$.
- 342 ■ For all $\mathbf{p} \notin \text{pt}(G)$, either $\mathbf{p} \notin \text{dom}(\Gamma)$ or $\Gamma(\mathbf{p}) = \text{end}$.

343 In Rocq this is defined with the following:

```

Definition assoc (g: tctx) (gt:gtt)  $\triangleq$ 
   $\forall p, (\text{isgPartsC } p \text{ gt} \rightarrow \exists Tp, M.\text{find } p \text{ g} = \text{Some } Tp \wedge$ 
     $\text{issubProj } Tp \text{ gt } p) \wedge$ 
     $(\neg \text{isgPartsC } p \text{ gt} \rightarrow \forall Tpx, M.\text{find } p \text{ g} = \text{Some } Tpx \rightarrow Tpx = \text{ltt\_end}).$ 

```

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

347 ► **Example 4.8.** In Example 4.5, we have that $\Gamma \sqsubseteq G$ where

$$348 \quad G := \mathbf{p} \rightarrow \mathbf{q} : \{\ell_0(\text{int}).G, \ell_1(\text{int}).\mathbf{q} \rightarrow \mathbf{r} : \{\ell_2(\text{int}).\text{end}\}\}$$

349 In fact, we have $\Gamma(s) = G \upharpoonright s$ for $s \in \{\mathbf{p}, \mathbf{q}, \mathbf{r}\}$. Similarly, we have $\Gamma' \sqsubseteq G'$ where

$$350 \quad G' := \mathbf{q} \rightarrow \mathbf{r} : \{\ell_2(\text{int}).\text{end}\}$$

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

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

► **Theorem 4.10** (Completeness of Association). *If $\text{assoc } \text{gamma } G$ and $\text{tctxR } \text{gamma} \ (\text{lcomm } p \ q \ \text{ell}) \ \text{gamma}'$, then there exists a global type tree G' such that $\text{assoc } \text{gamma}' \ G'$ and $\text{gttstepC } G \ G' \ p \ q \ \text{ell}$.*

► **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

$$\Gamma = p : q \oplus \{\ell_0(\text{int}).\text{end}\}, q : p \& \{\ell_0(\text{int}).\text{end}, \ell_1(\text{int}).\text{end}\}$$

and

$$G = p \rightarrow q : \{\ell_0(\text{int}).\text{end}, \ell_1(\text{int}).\text{end}\}$$

We have $\Gamma \sqsubseteq G$ and $G \xrightarrow{(p,q)\ell_1}$. However $\Gamma \xrightarrow{(p,q)\ell_1}$ is not a valid transition. Note that soundness still requires that $\Gamma \xrightarrow{(p,q)\ell_x}$ for some x , which is satisfied in this case by the valid transition $\Gamma \xrightarrow{(p,q)\ell_0}$.

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, liveness and fairness properties based on the definitions in [44].

5.1 Safety

We start by defining safety:

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

$$\begin{aligned} \Gamma \xrightarrow{p:q \oplus \ell(S)} \text{ and } \Gamma \xrightarrow{q:p \& \ell'(S')} \text{ implies } \Gamma \xrightarrow{(p,q)\ell} & \quad [\text{S-}\&\oplus] \\ \Gamma \rightarrow \Gamma' \text{ implies } \Gamma' \in \text{safe} & \quad [\text{S-}\rightarrow] \end{aligned}$$

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

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

► **Example 5.2.** Let $\Gamma_A = p : \text{end}$, then Γ_A is safe: the set of reducts is $\{\Gamma_A\}$ and this set respects $[\text{S-}\&\oplus]$ as its elements can't reduce, and it respects $[\text{S-}\rightarrow]$ as it's closed with respect to \rightarrow .

391 Let $\Gamma_B = p : q \oplus \{\ell_0(\text{int}).\text{end}\}, q : p \& \{\ell_0(\text{nat}).\text{end}\}$. Γ_B is not safe as as we have
 392 $\Gamma_B \xrightarrow{p:q \oplus \ell_0}$ and $\Gamma_B \xrightarrow{q:p \& \ell_0}$ but we don't have $\Gamma_B \xrightarrow{(p,q)\ell_0}$ as $\text{int} \not\leq \text{nat}$.

393 Let $\Gamma_C = p : q \oplus \{\ell_1(\text{int}).q \oplus \{\ell_0(\text{int}).\text{end}\}\}, q : p \& \{\ell_1(\text{int}).p \& \{\ell_0(\text{nat}).\text{end}\}\}$. Γ_C is not
 394 safe as we have $\Gamma_C \xrightarrow{(p,q)\ell_1} \Gamma_B$ and Γ_B is not safe.

395 Consider Γ from Example 4.5. All the reducts satisfy $[S-\&\oplus]$, hence Γ is safe.

396 Being a coinductive property, **safe** can be expressed in Rocq using Paco:

```

Definition weak_safety (c: tctx)  $\triangleq$ 
 $\forall p\ q\ s\ s'\ k\ k', \text{tctxRE } (\text{lend } p\ q\ (\text{Some } s)\ k)\ c \rightarrow \text{tctxRE } (\text{lrecv } q\ p\ (\text{Some } s')\ k')\ c \rightarrow$ 
 $\text{tctxRE } (\text{lcomm } p\ q\ k)\ c.$ 

Inductive safe (R: tctx  $\rightarrow$  Prop): tctx  $\rightarrow$  Prop  $\triangleq$ 
| safety_red :  $\forall c, \text{weak\_safety } c \rightarrow (\forall p\ q\ c' k,$ 
 $\text{tctxR } c\ (\text{lcomm } p\ q\ k)\ c' \rightarrow R\ c')$ 
 $\rightarrow \text{safe } R\ c.$ 

Definition safeC c  $\triangleq$  paco1 safe bot1 c.

```

397
 398 **weak_safety** corresponds $[S-\&\oplus]$ where $\text{tctxRE } 1\ c$ is shorthand for $\exists c', \text{tctxR } c\ 1\ c'$. In
 399 the inductive **safe**, the constructor **safety_red** corresponds to $[S-\rightarrow]$. Then **safeC** is defined
 400 as the greatest fixed point of **safe**.

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

402 ► **Theorem 5.3** (Safety by Association 🐼). *If $\text{assoc } \text{gamma } g$ then $\text{safeC } \text{gamma}$.*

403 5.2 Fairness and Liveness

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

406 ► **Definition 5.4** (Fair, Live Paths). *A local type context reduction path (also called executions
 407 or runs) is a possibly infinite sequence of transitions $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_1} \dots$ such that λ_i is a
 408 synchronous transition label, that is, of the form $(p,q)\ell$, for all i .*

409 *We say that a local type context reduction path $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_1} \dots$ is fair if, for all
 410 $n \in \mathbb{N} : \Gamma_n \xrightarrow{(p,q)\ell} \dots$ implies $\exists k, \ell'$ such that $N \ni k \geq n$ and $\lambda_k = (p,q)\ell'$, and therefore
 411 $\Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$. We say that a path $(\Gamma_n)_{n \in \mathbb{N}}$ is live iff, $\forall n \in \mathbb{N}$:*

- 412 1. $\forall n \in \mathbb{N} : \Gamma_n \xrightarrow{p:q \oplus \ell(S)} \dots$ implies $\exists k, \ell'$ such that $N \ni k \geq n$ and $\Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$
- 413 2. $\forall n \in \mathbb{N} : \Gamma_n \xrightarrow{q:p \& \ell(S)} \dots$ implies $\exists k, \ell'$ such that $N \ni k \geq n$ and $\Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$

414 ► **Definition 5.5** (Live Local Type Context). *A local type context Γ is live if whenever $\Gamma \rightarrow^* \Gamma'$,
 415 every fair path starting from Γ' is also live.*

416 In general, fairness assumptions are used so that only the reduction sequences that are
 417 "well-behaved" in some sense are considered when formulating other properties [42]. For our
 418 purposes we define fairness such that, in a fair path, if at any point p attempts to send to q
 419 and q attempts to send to p then eventually a communication between p and q takes place.
 420 Then live paths are defined to be paths such that whenever p attempts to send to q or q
 421 attempts to send to p , eventually a p to q communication takes place. Informally, this means
 422 that every communication request is eventually answered. Then live typing contexts are
 423 defined to be the Γ where all fair paths that start from Γ are also live.

424 ► **Example 5.6.** Consider the contexts Γ, Γ' and Γ_{end} from Example 4.5. One possible
 425 reduction path is $\Gamma \xrightarrow{(p,q)\ell_0} \Gamma \xrightarrow{(p,q)\ell_0} \dots$. Denote this path as $(\Gamma_n)_{n \in \mathbb{N}}$, where $\Gamma_n = \Gamma$

23:14 Dummy short title

for all $n \in \mathbb{N}$. We have $\forall n, \Gamma_n \xrightarrow{(p,q)\ell_0}$ and $\Gamma_n \xrightarrow{(p,q)\ell_1}$ as the only possible synchronised reductions from Γ_n . Accordingly, we also have $\forall n, \Gamma_n \xrightarrow{(p,q)\ell_0} \Gamma_{n+1}$ in the path so this path is fair. However, this path is not live as we have $\Gamma_1 \xrightarrow{r;q\&\ell_2(\text{int})}$ but there is no n, ℓ' with $\Gamma_n \xrightarrow{(q,r)\ell'} \Gamma_{n+1}$ in the path. Consequently, Γ is not a live type context.

Now consider the reduction path $\Gamma \xrightarrow{(p,q)\ell_0} \Gamma \xrightarrow{(p,q)\ell_0} \Gamma' \xrightarrow{(q,r)\ell_2} \Gamma_{\text{end}}$. 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 contained in them. 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]. \Diamond and \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. We now detail the Rocq Proof that associated local type contexts are also live.

► **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 projections² of g , that is, $\text{gamma_proj} = \{p_i : G \vdash_{p_i} \mid p_i \in \text{pt}\{G\}\}$, is live. We then leverage Theorem 4.10 to show that if gamma_proj is live, so is gamma .

The proof that gamma_proj is live proceeds by well-founded induction on the tree height [12] of the grafting (Lemma 3.10) of the global type g . Suppose $\text{gamma_proj} \xrightarrow{p;q\oplus\ell(S)}$ (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 $(p, q)\ell$ transition in xs . We prove the following helper lemmas:

■ The height of the p -grafting of g is not smaller than the q -grafting.

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.

460 ■ If the p -grafting and q -grafting of a global type g' have the same height, then any fair
 461 path beginning with the direct projection context of g' eventually contains a $(p, q)\ell$
 462 transition .

463 ■ The height of the p -grafting of g strictly decreases with every transition involving q ,
 464 and doesn't increase with the transitions not involving q .

465 These lemmas followed by well-founded induction on the height of the p -grafting of the global
 466 type the head of xs is projected from gives the desired transition.

467 In the second step of the proof we extend association on to paths to get, for each local
 468 type context reduction path xs that begins with γ , another local type context reduction
 469 path ys beginning with γ_{proj} such that the elements of xs are subtypes (subtyping
 470 on contexts defined pointwise) of the corresponding elements of ys . This is obtained from
 471 Theorem 4.10, however the statement of Theorem 4.10 is implemented as an \exists statement
 472 that lives in **Prop**, hence we need to use the `constructive_indefinite_description` axiom to
 473 construct a `CoFixpoint` returning the desired cosequence ys . The proof then follows by the
 474 definition of subtyping (Definition 3.3). ◀

475 6 Properties of Sessions

476 We give typing rules for the session calculus introduced in 2, and prove subject reduction and
 477 progress for them. Then we define a liveness property for sessions, and show that processes
 478 typable by a local type context that's associated with a global type tree are guaranteed to
 479 satisfy this liveness property.

480 6.1 Typing rules

481 We give typing rules for our session calculus based on [17] and [14].

482 We distinguish between two kinds of typing judgements and type contexts.

483 1. A local type context Γ associates participants with local type trees, as defined in `cdef-`
 484 `type-ctx`. Local type contexts are used to type sessions (Definition 2.2) i.e. a set of pairs
 485 of participants and single processes composed in parallel. We express such judgements as
 486 $\Gamma \vdash_{\mathcal{M}} \mathcal{M}$, or as `typ_sess M gamma` or $\gamma \vdash M$ in Rocq.

487 2. A process variable context Θ_T associates process variables with local type trees, and an
 488 expression variable context Θ_e assigns sorts to expression variables. Variable contexts
 489 are used to type single processes and expressions (Definition 2.1). Such judgements are
 490 expressed as $\Theta_T, \Theta_e \vdash_P P : T$, or in Rocq as `typ_proc theta_T theta_e P T` or $\text{theta}_T,$
 491 $\text{theta}_e \vdash P : T$.

$\Theta \vdash_P n : \text{nat}$	$\Theta \vdash_P i : \text{int}$	$\Theta \vdash_P \text{true} : \text{bool}$	$\Theta \vdash_P \text{false} : \text{bool}$	$\Theta, x : S \vdash_P x : S$
$\frac{\Theta \vdash_P e : \text{nat}}{\Theta \vdash_P \text{succ } e : \text{nat}}$	$\frac{\Theta \vdash_P e : \text{int}}{\Theta \vdash_P \text{neg } e : \text{int}}$	$\frac{\Theta \vdash_P e : \text{bool}}{\Theta \vdash_P \neg e : \text{bool}}$		
$\frac{\Theta \vdash_P e_1 : S \quad \Theta \vdash_P e_2 : S}{\Theta \vdash_P e_1 \oplus e_2 : S}$	$\frac{\Theta \vdash_P e_1 : \text{int} \quad \Theta \vdash_P e_2 : \text{int}}{\Theta \vdash_P e_1 > e_2 : \text{bool}}$	$\frac{\Theta \vdash_P e : S \quad S \leq S'}{\Theta \vdash_P e : S'}$		

■ Table 4 Typing expressions

$$\begin{array}{c}
\frac{[T\text{-END}]}{\Theta \vdash_P \mathbf{0} : \text{end}} \quad \frac{[T\text{-VAR}]}{\Theta, \mathbf{X} : T \vdash_P \mathbf{X} : T} \quad \frac{[T\text{-REC}]}{\Theta, \mathbf{X} : T \vdash_P P : T} \quad \frac{[T\text{-IF}]}{\Theta \vdash_P e : \text{bool} \quad \Theta \vdash_P P_1 : T \quad \Theta \vdash_P P_2 : T} \\
\frac{[T\text{-SUB}]}{\Theta \vdash_P P : T \quad T \leq T'} \quad \frac{[T\text{-IN}]}{\Theta \vdash_P \sum_{i \in I} p_i ? \ell_i(x_i). P_i : p_i \& \{ \ell_i(S_i). T_i \}_{i \in I}} \quad \frac{[T\text{-OUT}]}{\Theta \vdash_P e : S \quad \Theta \vdash_P P : T} \\
\Theta \vdash_P p_i ! \ell(e). P : p_i \oplus \{ \ell(S). T \}
\end{array}$$

■ **Table 5** Typing processes

Table 4 and Table 5 state the standard typing rules for expressions and processes which we don't elaborate on. We have a single rule for typing sessions:

$$\frac{[T\text{-SESS}]}{\forall i \in I : \quad \vdash_P P_i : \Gamma(p_i) \quad \Gamma \sqsubseteq G} \quad \Gamma \vdash_{\mathcal{M}} \prod_i p_i \triangleleft P_i$$


[T-SESS] says that a session made of the parallel composition of processes $\prod_i p_i \triangleleft P_i$ 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


give theorem
no

The subject reduction, progress and non-stuck theorems from [14] also hold in this setting, with minor changes in their statements and proofs. We won't discuss these proofs in detail.

► **Lemma 6.1.** *If $\text{gamma} \vdash_{\mathcal{M}} M$ and $M \Rightarrow M'$ then $\text{typ_sess } M' \text{ gamma}$.*


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

► **Remark 6.3.** 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 to be more convenient.

► **Theorem 6.4** (Deadlock Freedom ). *If $\text{gamma} \vdash_{\mathcal{M}} M$, one of the following hold :*

1. *Either $M \Rightarrow M_{\text{inact}}$ where every process making up M_{inact} is inactive, i.e. $M_{\text{inact}} \equiv \prod_{i=1}^n p_i \triangleleft \mathbf{0}$ for some n .*
2. *Or there is a M' such that $M \rightarrow M'$.*

We can also prove the following correspondence result in the reverse direction to Theorem 6.2, analogous to Theorem 4.9.

► **Theorem 6.5** (Session Fidelity ). *If $\text{gamma} \vdash_{\mathcal{M}} M$ and $\text{gamma} \xrightarrow{(p,q)\ell} \text{gamma}'$, there exists a message label ℓ' , a context gamma'' and a session M' such that $M \xrightarrow{(p,q)\ell'} M'$, $\text{gamma} \xrightarrow{(p,q)\ell'} \text{gamma}''$ and $\text{typ_sess } M' \text{ gamma}''$.*

520 ▶ **Remark 6.6.** Again we note that by Theorem 6.5 a single-step context reduction induces a
 521 single-step session reduction on the type. With the τ -semantics the session reduction induced
 522 by the context reduction would be multistep.

523 Now the following type safety property follows from the above theorems:

524 ▶ **Theorem 6.7 (Type Safety 🦋).** *If $\text{gamma} \vdash_{\mathcal{M}} M$ and $M \rightarrow^* M' \Rightarrow (p \leftarrow p_send\ q\ \text{ell}\ P$
 525 $||| q \leftarrow p_recv\ p\ \text{xs} ||| M'')$, then $\text{onth}\ \text{ell}\ \text{xs} \neq \text{None}$.*

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

527 ▶ **Definition 6.8 (Session Liveness).** *Session \mathcal{M} is live iff*

- 528 1. $\mathcal{M} \rightarrow^* \mathcal{M}' \Rightarrow q \triangleleft p!_{\ell_i}(x_i).Q \mid \mathcal{N}$ implies $\mathcal{M}' \rightarrow^* \mathcal{M}'' \Rightarrow q \triangleleft Q \mid \mathcal{N}'$ for some $\mathcal{M}'', \mathcal{N}'$
- 529 2. $\mathcal{M} \rightarrow^* \mathcal{M}' \Rightarrow q \triangleleft \bigwedge_{i \in I} p?_{\ell_i}(x_i).Q_i \mid \mathcal{N}$ implies $\mathcal{M}' \rightarrow^* \mathcal{M}'' \Rightarrow q \triangleleft Q_i[v/x_i] \mid \mathcal{N}'$ for some
 530 $\mathcal{M}'', \mathcal{N}', i, v$.

531 In Rocq this is expressed with the predicate `live_sess` 🦋:

```

Definition live_sess Mp  $\triangleq \forall M, \text{betaRtc}\ Mp\ M \rightarrow$ 
  ( $\forall p\ q\ \text{ell}\ e\ P'\ M', p \neq q \rightarrow \text{unfoldP}\ M\ ((p \leftarrow p\_send\ q\ \text{ell}\ e\ P') ||| M') \rightarrow \exists M'',$ 
   betaRtc M ((p  $\leftarrow$  P')\(|\backslash|\backslash|\backslash|M'')))
  /\
  ( $\forall p\ q\ \text{llp}\ M', p \neq q \rightarrow \text{unfoldP}\ M\ ((p \leftarrow p\_recv\ q\ \text{llp}) ||| M') \rightarrow$ 
    $\exists M'', P'\ e\ k, \text{onth}\ k\ \text{llp} = \text{Some}\ P' \wedge \text{betaRtc}\ M\ ((p \leftarrow \text{subst\_expr\_proc}\ P'\ e\ 0\ 0) ||| M''))$ ).

```

533 Session liveness, analogous to liveness for typing contexts (Definition 5.4), says that when
 534 \mathcal{M} is live, if \mathcal{M} reduces to a session \mathcal{M}' containing a participant that's attempting to send
 535 or receive, then \mathcal{M}' reduces to a session where that communication has happened. It's also
 536 called *lock-freedom* in related work ([41, 30]).

537 We can now prove that typed sessions are live. First we prove the following lemma:

538 ▶ **Lemma 6.9 (Fair Extension of Typed Sessions 🦋).** *If $\text{typ_sess}\ M\ \text{gamma}$, then there exists a
 539 session reduction path xs starting from M such that the following fairness property holds:*
 540 ■ *On xs , whenever a transition with label $(p, q)\ell$ is enabled, a transition with label $(p, q)\ell'$
 541 eventually occurs for some ℓ' .*

542 **Proof.** The desired path can be constructed by repeatedly cycling through all participants,
 543 checking if there is a transition involving that participant, and executing that transition if
 544 there is. Correctness follows from Theorem 6.2 and Theorem 6.5. ◀

545 Lemma 6.9 defines a "fairness" property for sessions analogous to Definition 5.4. It then
 546 shows that there exists a fair path from any typable session. This resembles the *feasibility*
 547 property expected from sensible notions of fairness [42], which states that any partial path
 548 can be extended into a fair one³.

549 ▶ **Remark 6.10.** As in the proof of Theorem 5.8, the construction in Lemma 6.9 uses the
 550 `constructive_indefinite_description` axiom to construct a `CoFixpoint`. Additionally, we
 551 use the axiom `excluded_middle_informative` for the "check if there is a transition involving a
 552 participant" part of the scheduling algorithm. The use of this axiom is probably not necessary
 553 but it makes the proof easier.

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 6.9 turns out to be enough to prove our liveness property.

► **Theorem 6.11** (Liveness by Typing). *For a session M_p , if $\exists \text{ gamma } \text{ gamma} \vdash_{\mathcal{M}} M_p$ then $\text{live_sess } M_p$.*

Proof. We detail the proof for the send case of Definition 6.8, the case for the receive is similar. Suppose that $M_p \rightarrow^* M$ and $M \Rightarrow ((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M')$. Our goal is to show that there exists a M'' such that $M \rightarrow^* ((p \leftarrow P') \mid \mid M'')$. First, observe that by [R-UNFOLD] it suffices to show that $((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M') \rightarrow^* M''$ for some M'' . Also note that $\text{gamma} \vdash_{\mathcal{M}} M$ for some gamma by Theorem 6.2, therefore $\text{gamma} \vdash_{\mathcal{M}} ((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M')$ by Lemma 6.1.

Now let xs be a fair session reduction path starting from $((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M')$, which exists by Lemma 6.9. 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 $\text{lcomm } p\ q\ \text{ell}'$ transition eventually occurs in ys for some ell' . Therefore $ys = \text{gamma} \rightarrow^* \text{gamma}_0 \xrightarrow{(p,q)\ell'} \text{gamma}_1 \rightarrow \dots$ for some $\text{gamma}_0, \text{gamma}_1$. Now consider the session M_0 typed by gamma_0 in xs . We have $((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M') \rightarrow^* M_0$ by M_0 being on xs . We also have that $M_0 \xrightarrow{(p,q)\ell''} M_1$ for some ℓ'', M_1 by Theorem 6.5. Now observe that $M_0 \equiv ((p \leftarrow p_send\ q\ \text{ell}\ e\ P') \mid \mid M'')$ for some M'' as no transitions involving p have happened on the reduction path to M_0 . Therefore $\ell = \ell''$, so $M_1 \equiv ((p \leftarrow P') \mid \mid M'')$ for some M'' , as needed. ◀

7 Conclusion and Related Work

Liveness Properties. 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 [16] 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 6.8 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 6.8, 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 6.8, but enjoys good composition properties.

Mechanisation. Mechanisation of session types in proof assistants is a relatively new effort. Our formalisation is built on recent work by Ekici et. al. [14] 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 necessitates the rewriting of subject reduction and progress proofs in addition to the operational correspondence, safety and liveness properties we have proved. Other recent results mechanised in Rocq include Ekici and Yoshida's [15] work on the completeness of asynchronous subtyping, and Tiore'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 separation 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 separation logic is an active research area that has produced tools such as TaDa [13], 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.

- 648 12 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction*
649 *to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 650 13 Emanuele D’Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live:
651 Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans.*
652 *Program. Lang. Syst.*, 43(4), November 2021. doi:10.1145/3477082.
- 653 14 Burak Ekici, Tadayoshi Kamegai, and Nobuko Yoshida. Formalising Subject Reduction and
654 Progress for Multiparty Session Processes. In Yannick Forster and Chantal Keller, editors, *16th*
655 *International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz*
656 *International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:23, Dagstuhl, Germany,
657 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.19)
658 [de/entities/document/10.4230/LIPIcs.ITP.2025.19](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.19), doi:10.4230/LIPIcs.ITP.2025.19.
- 659 15 Burak Ekici and Nobuko Yoshida. Completeness of Asynchronous Session Tree Subtyping
660 in Coq. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International*
661 *Conference on Interactive Theorem Proving (ITP 2024)*, volume 309 of *Leibniz International*
662 *Proceedings in Informatics (LIPIcs)*, pages 13:1–13:20, Dagstuhl, Germany, 2024. Schloss
663 Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. URL: [https://drops.dagstuhl.](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.13)
664 [de/entities/document/10.4230/LIPIcs.ITP.2024.13](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.13), doi:10.4230/LIPIcs.ITP.2024.13.
- 665 16 Nissim Francez. *Fairness*. Springer US, New York, NY, 1986. URL: [http://link.springer.](http://link.springer.com/10.1007/978-1-4612-4886-6)
666 [com/10.1007/978-1-4612-4886-6](http://link.springer.com/10.1007/978-1-4612-4886-6), doi:10.1007/978-1-4612-4886-6.
- 667 17 Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida.
668 Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Meth-*
669 *ods in Programming*, 104:127–173, 2019. URL: [https://www.sciencedirect.com/science/](https://www.sciencedirect.com/science/article/pii/S2352220817302237)
670 [article/pii/S2352220817302237](https://www.sciencedirect.com/science/article/pii/S2352220817302237), doi:10.1016/j.jlamp.2018.12.002.
- 671 18 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type
672 based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*,
673 4(POPL):1–30, 2019.
- 674 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types.
675 *SIGPLAN Not.*, 43(1):273–284, January 2008. doi:10.1145/1328897.1328472.
- 676 20 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization
677 in coinductive proof. *SIGPLAN Not.*, 48(1):193–206, January 2013. doi:10.1145/2480359.
678 2429093.
- 679 21 Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Deadlock-free separation
680 logic: Linearity yields progress for dependent higher-order message passing. *Proceedings of the*
681 *ACM on Programming Languages*, 8(POPL):1385–1417, 2024.
- 682 22 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek
683 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
684 logic. *Journal of Functional Programming*, 28:e20, 2018.
- 685 23 Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*,
686 177(2):122–159, September 2002. URL: [https://www.sciencedirect.com/science/article/](https://www.sciencedirect.com/science/article/pii/S0890540102931718)
687 [pii/S0890540102931718](https://www.sciencedirect.com/science/article/pii/S0890540102931718), doi:10.1006/inco.2002.3171.
- 688 24 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proceedings of*
689 *the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 1–13, New
690 York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471874.3472979.
- 691 25 Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur.
692 Fair operational semantics. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/
693 3591253.
- 694 26 Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur.
695 Lilo: A higher-order, relational concurrent separation logic for liveness. *Proceedings of the*
696 *ACM on Programming Languages*, 9(OOPSLA1):1267–1294, 2025.
- 697 27 Pierre Letouzey and Andrew W. Appel. Modular Finite Maps over Ordered Types. URL:
698 <https://github.com/rocq-community/mmmaps>.

- 699 28 Sam Lindley and J Garrett Morris. Embedding session types in haskell. *ACM SIGPLAN*
700 *Notices*, 51(12):133–145, 2016.
- 701 29 Robin MILNER. Chapter 19 - operational and algebraic semantics of concurrent pro-
702 cesses. In JAN VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook
703 of Theoretical Computer Science, pages 1201–1242. Elsevier, Amsterdam, 1990. URL:
704 <https://www.sciencedirect.com/science/article/pii/B978044488074150024X>, doi:10.
705 1016/B978-0-444-88074-1.50024-X.
- 706 30 Luca Padovani. Deadlock and lock freedom in the linear pi-calculus. In *Proceedings of the*
707 *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic*
708 *(CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science*
709 *(LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.
710 doi:10.1145/2603088.2603116.
- 711 31 Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. Typing Liveness in
712 Multiparty Communicating Systems. In Eva Kühn and Rosario Pugliese, editors, *Coordination*
713 *Models and Languages*, pages 147–162, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 714 32 Kai Pischke and Nobuko Yoshida. *Asynchronous Global Protocols, Precisely*, pages 116–133.
715 Springer Nature Switzerland, Cham, 2026. doi:10.1007/978-3-031-99717-4_7.
- 716 33 Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of*
717 *computer science (sfcs 1977)*, pages 46–57. iee, 1977.
- 718 34 Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. In *Proceedings*
719 *of the first ACM SIGPLAN symposium on Haskell*, pages 25–36, 2008.
- 720 35 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc.*
721 *ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290343.
- 722 36 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In *Proceedings*
723 *of the 21st International Symposium on Principles and Practice of Declarative Programming*,
724 PPDP '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/
725 3354166.3354184.
- 726 37 Dawit Tiore. A mechanisation of multiparty session types, 2024.
- 727 38 Dawit Tiore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for
728 global types. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*,
729 pages 28–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.
- 730 39 Dawit Tiore, Jesper Bengtson, and Marco Carbone. Multiparty asynchronous session types:
731 A mechanised proof of subject reduction. In *39th European Conference on Object-Oriented*
732 *Programming (ECOOP 2025)*, pages 31–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik,
733 2025.
- 734 40 Thien Udomsrirungruang and Nobuko Yoshida. Top-down or bottom-up? complexity analyses
735 of synchronous multiparty session types. *Proceedings of the ACM on Programming Languages*,
736 9(POPL):1040–1071, 2025.
- 737 41 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make
738 session types complete for lock-freedom. In *Proceedings of the 36th Annual ACM/IEEE*
739 *Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021. Association
740 for Computing Machinery. doi:10.1109/LICS52264.2021.9470531.
- 741 42 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Computing*
742 *Surveys*, 52(4):1–38, August 2019. URL: <http://dx.doi.org/10.1145/3329125>, doi:10.1145/
743 3329125.
- 744 43 Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012.
745 doi:10.1145/2398856.2364568.
- 746 44 Nobuko Yoshida and Ping Hou. Less is more revisited, 2024. URL: [https://arxiv.org/abs/](https://arxiv.org/abs/2402.16741)
747 [2402.16741](https://arxiv.org/abs/2402.16741), arXiv:2402.16741.