

# Formally Verified Liveness with Synchronous Multiparty Session Types in Rocq

Anonymous author

Anonymous affiliation

Anonymous author

Anonymous affiliation

## Abstract

We mechanise a synchronous multiparty session type framework that guarantees liveness for typed processes. We type sessions using a context of local types, and use "association" with global types to denote a set of well-behaved local type contexts. We give LTS semantics to local contexts and global types and prove operational correspondences between the LTSs local context and their associated global types. We then prove that sessions typed by a local context that's associated with a global type are live.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

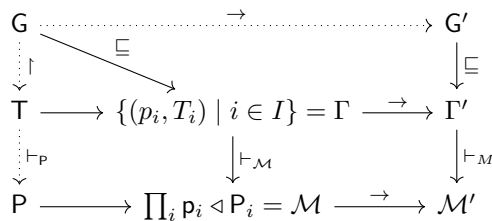
Acknowledgements Anonymous acknowledgements

## 1 Introduction

In this work we present the Rocq formalisation of a session type system for a simple session calculus, and prove that sessions typable in this system are *safe*, *deadlock-free*, and *live*. The approach we take in our type system is very similar to the one followed by Hou and Yoshida in [16]. Namely, we proceed by defining local and global type trees, and relate them using projections. We then extend this projection relation to an *association* relation between local type contexts i.e. collections of local types paired with participants, and global type trees. Next we give LTS semantics to local type contexts and global type trees, and prove an operational correspondence between them. We then proceed to formulate safety and liveness properties for local type contexts, and show that local type contexts associated with global type trees enjoy these properties. We relate associated local type contexts to sessions via typing rules, and demonstrate an operational correspondence between contexts and sessions via *subject reduction*, *progress* and *session fidelity* theorems. Finally we show, using the liveness properties we defined on local type contexts, that typable sessions are live.

Our Rocq implementation builds upon the recent formalisation of subject reduction for MPST by Ekici et. al. [4], which itself is based on [5]. The methodology in [4] takes an

Session types  
introduction  
Liveness intro-  
duction



**Figure 1** Design overview. The dotted lines correspond to relations inherited from [4] while the solid lines denote relations that are new, or substantially rewritten, in this paper.

specifics of  
the project

36 equirecursive approach where an inductive syntactic global or local type is identified with  
37 the coinductive tree obtained by fully unfolding the recursion. It then defines a coinductive  
38 projection relation between global and local type trees, the LTS semantics for global type  
39 trees, and typing rules for the session calculus outlined in [5]. We extensively make use of  
40 these definitions and the lemmas concerning them.

41 **Outline.** In Section 2 we define our session calculus and its LTS semantics. In Section 3  
42 we introduce local and global type trees. In Section 4 we give LTS semantics to local type  
43 contexts and global types, and detail the association relation between them. In Section 5  
44 we define safety and liveness for local type contexts, and prove that they hold for contexts  
45 associated with a global type tree. In Section 6 we give the typing rules for our session  
46 calculus, and prove *non-stuck* and *liveness* properties for typable sessions.  
47

## 48 2 The Session Calculus

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

### 51 2.1 Processes and Sessions

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

$$53 \quad P ::= p!\ell(e).P \mid \sum_{i \in I} p?\ell_i(x_i).P_i \mid \text{if } e \text{ then } P \text{ else } P \mid \mu X.P \mid X \mid 0$$

54 where  $e$  is an expression that can be a variable, a value such as *true*, 0 or  $-3$ , or a term  
55 built from expressions by applying the operators *succ*, *neg*,  $\neg$ , non-deterministic choice  $\oplus$   
56 and  $>$ .

57  $p!\ell(e).P$  is a process that sends the value of expression  $e$  with label  $\ell$  to participant  $p$ , and  
58 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  
59 any label  $\ell_i$  where  $i \in I$ , binding the result to  $x_i$  and continuing with  $P_i$ , depending on  
60 which  $\ell_i$  the value was received from.  $X$  is a recursion variable,  $\mu X.P$  is a recursive process,  
61 if  $e$  then  $P$  else  $P$  is a conditional and  $0$  is a terminated process.

62 Processes can be composed in parallel into sessions.

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

$$64 \quad \mathcal{M} ::= p \triangleleft P \mid (\mathcal{M} \mid \mathcal{M}) \mid \mathcal{O}$$

65  $p \triangleleft P$  denotes that participant  $p$  is running the process  $P$ ,  $\mid$  indicates parallel composition. We  
66 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

67 an empty session with no participants, that is, the unit of parallel composition.

68 ► **Remark 2.3.** Note that  $\mathcal{O}$  is different than  $p \triangleleft 0$  as  $p$  is a participant in the latter but not  
69 the former. This differs from previous work, e.g. in [5] the unit of parallel composition is  
70  $p \triangleleft 0$  while in [4] there is no unit. The unitless approach of [4] results in a lot of repetition  
71 in the code, for an example see their definition of `unfoldP` which contains two of every  
72 constructor: one for when the session is composed of exactly two processes, and one for  
73 when it's composed of three or more. Therefore we chose to add an unit element to parallel  
74 composition. However, we didn't make that unit  $p \triangleleft 0$  in order to reuse some of the lemmas  
75 from [4] that use the fact that structural congruence preserves participants.

76 In Rocq processes and sessions are expressed in the following way

```

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).

```

77

## 2.2 Structural Congruence and Operational Semantics

78

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

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

■ **Table 1** Structural Congruence over Sessions

81 We now give the operational semantics for sessions by the means of a labelled transition  
 82 system. We will be giving two types of semantics: one which contains silent  $\tau$  transitions,  
 83 and another, *reactive* semantics [15] which doesn't contain explicit  $\tau$  reductions while still  
 84 considering  $\beta$  reductions up to silent actions. We will mostly be using the reactive semantics  
 85 throughout this paper, for the advantages of this approach see Remark 6.4.

### 2.2.1 Semantics With Silent Transitions

86

87 We have two kinds of transitions, *silent* ( $\tau$ ) and *observable* ( $\beta$ ). Correspondingly, we have  
 88 two kinds of *transition labels*,  $\tau$  and  $(p, q)\ell$  where  $p, q$  are participants and  $\ell$  is a message  
 89 label. We omit the semantics of expressions, they are standard and can be found in [5, Table  
 90 1]. We write  $e \downarrow v$  when expression  $e$  evaluates to value  $v$ .

$$\begin{array}{c}
 \text{[R-COMM]} \\
 \frac{j \in I \quad e \downarrow v}{p \triangleleft \sum_{i \in I} q? \ell_i(x_i).P_i \mid q \triangleleft p! \ell_j(e).Q \mid \mathcal{N} \xrightarrow{(p,q)\ell_j} p \triangleleft P_j[v/x_j] \mid q \triangleleft Q \mid \mathcal{N}} \\
 \\ 
 \begin{array}{ll}
 \text{[R-REC]} & \text{[R-CONDT]} \\
 p \triangleleft \mu \mathbf{X}.P \mid \mathcal{N} \xrightarrow{\tau} p \triangleleft P[\mu \mathbf{X}.P/\mathbf{X}] \mid \mathcal{N} & \frac{e \downarrow \text{true}}{p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathcal{N} \xrightarrow{\tau} p \triangleleft P \mid \mathcal{N}}
 \end{array} \\
 \\ 
 \begin{array}{ll}
 \text{[R-CONF]} & \text{[R-STRUCT]} \\
 \frac{e \downarrow \text{false}}{p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathcal{N} \xrightarrow{\tau} p \triangleleft Q \mid \mathcal{N}} & \frac{\mathcal{N}'_1 \equiv \mathcal{N}_1 \quad \mathcal{N}_1 \xrightarrow{\lambda} \mathcal{N}_2 \quad \mathcal{N}_2 \equiv \mathcal{N}'_2}{\mathcal{N}'_1 \xrightarrow{\lambda} \mathcal{N}'_2}
 \end{array}
 \end{array}$$

■ **Table 2** Operational Semantics of Sessions

In Table 2, [R-COMM] describes a synchronous communication from  $p$  to  $q$  via message label  $\ell_j$ . [R-REC] unfolds recursion, [R-CONDT] and [R-CONDF] express how to evaluate conditionals, and [R-STRUCT] shows that the reduction respects the structural pre-congruence. We write  $\mathcal{M} \rightarrow \mathcal{N}$  if  $\mathcal{M} \xrightarrow{\lambda} \mathcal{N}$  for some transition label  $\lambda$ . We write  $\rightarrow^*$  to denote the reflexive transitive closure of  $\rightarrow$ .

## 2.3 Reactive Semantics

In reactive semantics  $\tau$  transitions are captured by an *unfolding* relation ( $\Rightarrow$ ), and  $\beta$  reductions are defined up to this unfolding.

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

■ **Table 3** Unfolding of Sessions

$\mathcal{M} \Rightarrow \mathcal{N}$  means that  $\mathcal{M}$  can transition to  $\mathcal{N}$  through some internal actions, or  $\tau$  transitions in the semantics of Section 2.2.1. We say that  $\mathcal{M}$  *unfolds* to  $\mathcal{N}$ . In Rocq it's captured by the predicate `unfoldP : session → session → Prop`.

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

■ **Table 4** Reactive Semantics of Sessions

[R-COMM] captures communications between processes, and [R-UNFOLD] lets us consider reductions up to unfoldings. 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 some  $\lambda$ , which is written `betaP M M'` in Rocq. We write  $\rightarrow^*$  to denote the reflexive transitive closure of  $\rightarrow$ , which is called `betaRtc` in Rocq.

## 3 The Type System

We introduce local and global types and trees and the subtyping and projection relations based on [5]. We start by defining the sorts that will be used to type expressions, and local types that will be used to type single processes.

### 3.1 Local Types and Type Trees

► **Definition 3.1** (Sorts). We define sorts as follows:

$$S ::= \text{int} \mid \text{bool} \mid \text{nat}$$

and the corresponding Rocq

```
Inductive sort: Type ≡
| sbool: sort
| sint : sort
| snat : sort.
```

► **Definition 3.2.** Local types are defined inductively with the following syntax:

$$\mathbb{T} ::= \text{end} \mid \mathbf{p}\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid \mathbf{p}\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid \mathbf{t} \mid \mu\mathbf{t}.\mathbb{T}$$

Informally, in the above definition, **end** represents a role that has finished communicating.  $\mathbf{p}\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$  denotes a role that may, from any  $i \in I$ , receive a value of sort  $S_i$  with message label  $\ell_i$  and continue with  $\mathbb{T}_i$ . Similarly,  $\mathbf{p}\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$  represents a role that may choose to send a value of sort  $S_i$  with message label  $\ell_i$  and continue with  $\mathbb{T}_i$  for any  $i \in I$ .  $\mu\mathbf{t}.\mathbb{T}$  represents a recursive type where  $\mathbf{t}$  is a type variable. We assume that the indexing sets  $I$  are always non-empty. We also assume that recursion is always guarded.

We employ an equirecursive approach based on the standard techniques from [11] where  $\mu\mathbf{t}.\mathbb{T}$  is considered to be equivalent to its unfolding  $\mathbb{T}[\mu\mathbf{t}.\mathbb{T}/\mathbf{t}]$ . This enables us to identify a recursive type with the possibly infinite local type tree obtained by fully unfolding its recursive subterms.

► **Definition 3.3.** Local type trees are defined coinductively with the following syntax:

$$\mathbb{T} ::= \text{end} \mid \mathbf{p}\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid \mathbf{p}\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$$

The corresponding Rocq definition is given below.

```
CoInductive ltt: Type ≡
| ltt_end : ltt
| ltt_recv: part → list (option(sort*ltt)) → ltt
| ltt_send: part → list (option(sort*ltt)) → ltt.
```

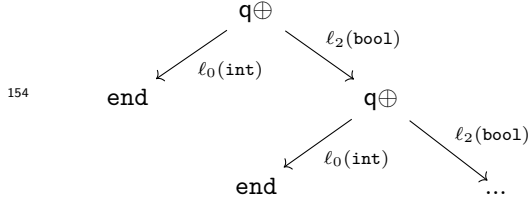
Note that in Rocq we represent the continuations using a `list` of `option` types. In a continuation `gcs : list (option(sort*ltt))`, index `k` (using zero-indexing) being equal to `Some (sk, Tk)` means that  $\ell_k(S_k).T_k$  is available in the continuation. Similarly index `k` being equal to `None` or being out of bounds of the list means that the message label  $\ell_k$  is not present in the continuation. Below are some of the constructions we use when working with option lists.

these may go

1. **SList xs**: A function that is equal to `True` if `xs` represents a continuation that has at least one element that is not `None`, and `False` otherwise.
2. **on<sub>th</sub> k xs**: A function that returns `Some x` if the element at index `k` (using 0-indexing) of `xs` is `Some x`, and returns `None` otherwise. Note that the function returns `None` if `k` is out of bounds for `xs`.
3. **Forall**, **Forall2** and **Forall2R** : **Forall** and **Forall2** are predicates from the Rocq Standard Library [14, List] that are used to quantify over elements of one list and pairwise elements of two lists, respectively. **Forall2R** is a weaker version of **Forall2** that might hold even if one parameter is shorter than the other. We frequently use **Forall2R** to express subset relations on continuations.

► Remark 3.4. 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.

► Example 3.5. Let local type  $T = \mu t. q \oplus \{ \ell_0(\text{int}).\text{end}, \ell_2(\text{bool}).t \}$ . This is equivalent to the following infinite local type tree:



and the following Rocq code

```
CoFixpoint T ≜ ltt_send q [Some (sint, ltt_end), None, Some (sbool, T)]
```

We omit the details of the translation between local types and local type trees, the technicalities of our approach is explained in [5], and the Rocq implementation of translation is detailed in [4]. From now on we work exclusively on local type trees.

► Remark 3.6. We will occasionally be talking about equality (=) between coinductively defined trees in Rocq. Rocq's Leibniz equality is not strong enough to treat as equal the types that we will deem to be the same. To do that, we define a coinductive predicate `lttIsoC` that captures isomorphism between coinductive trees and take as an axiom that `lttIsoC T1 T2 → T1=T2`. Technical details can be found in [4].

## 3.2 Subtyping

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

► Definition 3.7 (Subsorting and Subtyping). *Subsorting  $\leq$  is the least reflexive binary relation that satisfies  $\text{nat} \leq \text{int}$ . Subtyping  $\leq$  is the largest relation between local type trees coinductively defined by the following rules:*

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

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

In Rocq we express coinductive relations such as subtyping using the Paco library [7]. The idea behind Paco is to formulate the coinductive predicate as the greatest fixpoint of an inductive relation parameterised by another relation  $R$  representing the "accumulated

179 knowledge" obtained during the course of the proof. Hence our subtyping relation looks like  
 180 the following:

```

Inductive subtype (R: ltt → ltt → Prop): ltt → ltt → Prop ≡
| sub_end: subtype R ltt_end ltt_end
| sub_in : ∀ p xs ys,
    wfrec subsort R ys xs →
    subtype R (litt_recv p xs) (litt_recv p ys)
| sub_out : ∀ p xs ys,
    wfsend subsort R xs ys →
    subtype R (litt_send p xs) (litt_send p ys).

Definition subtypeC l1 l2 ≡ paco2 subtype bot2 l1 l2.

```

181

182 In definition of the inductive relation `subtype`, constructors `sub_in` and `sub_out` correspond  
 183 to [SUB-IN] and [SUB-OUT] with `wfrec` and `wfsend` expressing the premises of those rules. Then  
 184 `subtypeC` defines the coinductive subtyping relation as a greatest fixed point. Given that the  
 185 relation `subtype` is monotone (proven in [4]), `paco2 subtype bot2` generates the greatest fixed  
 186 point of `subtype` with the "accumulated knowledge" parameter set to the empty relation `bot2`.  
 187 The 2 at the end of `paco2` and `bot2` stands for the arity of the predicates.

### 188 3.3 Global Types and Type Trees

189 While local types specify the behaviour of one role in a protocol, global types give a bird's  
 190 eye view of the whole protocol.

191 ► **Definition 3.8** (Global type). *We define global types inductively as follows:*

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

193 *We further inductively define the function  $\text{pt}(\mathbb{G})$  that denotes the participants of type  $\mathbb{G}$ :*

194  $\text{pt}(\text{end}) = \text{pt}(t) = \emptyset$

195  $\text{pt}(p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}) = \{p, q\} \cup \bigcup_{i \in I} \text{pt}(G_i)$

196  $\text{pt}(\mu t. \mathbb{G}) = \text{pt}(\mathbb{G})$

197 `end` denotes a protocol that has ended,  $p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$  denotes a protocol where for  
 198 any  $i \in I$ , participant  $p$  may send a value of sort  $S_i$  to another participant  $q$  via message  
 199 label  $\ell_i$ , after which the protocol continues as  $G_i$ .

200 As in the case of local types, we adopt an equirecursive approach and work exclusively  
 201 on possibly infinite global type trees.

202 ► **Definition 3.9** (Global type trees). *We define global type trees coinductively as follows:*

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

204 *with the corresponding Rocq code*

```

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

```

205

206 We extend the function  $\text{pt}$  onto trees by defining  $\text{pt}(\mathbb{G}) = \text{pt}(\mathbb{G})$  where the global type  
 207  $\mathbb{G}$  corresponds to the global type tree  $G$ . Technical details of this definition such as well-  
 208 definedness can be found in [4, 5].

209 In Rocq  $\text{pt}$  is captured with the predicate  $\text{isgPartsC} : \text{part} \rightarrow \text{gtt} \rightarrow \text{Prop}$ , where  
 210  $\text{isgPartsC } p \ G$  denotes  $p \in \text{pt}(G)$ .

### 3.4 Projection

We give definitions of projections with plain merging.

► **Definition 3.10** (Projection). *The projection of a global type tree onto a participant  $r$  is the largest relation  $\vdash_r$  between global type trees and local type trees such that, whenever  $G \vdash_r T$ :*

- $r \notin \text{pt}\{G\}$  implies  $T = \text{end}$ ; [PROJ-END]
- $G = p \rightarrow r : \{\ell_i(S_i).G_i\}_{i \in I}$  implies  $T = p \& \{\ell_i(S_i).T_i\}_{i \in I}$  and  $\forall i \in I, G \vdash_r T_i$  [PROJ-IN]
- $G = r \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$  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]
- $G = p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$  and  $r \notin \{p, q\}$  implies that there are  $T_i, i \in I$  such that  $T = \sqcap_{i \in I} T_i$  and  $\forall i \in I, G \vdash_r T_i$  [PROJ-CONT]

where  $\sqcap$  is the merging operator. We also define plain merge  $\sqcap$  as

$$T_1 \sqcap T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

► **Remark 3.11.** In the MPST literature there exists a more powerful merge operator named full merging, defined as

$$T_1 \sqcap T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = p \& \{\ell_i(S_i).T_i\}_{i \in I} & \text{and} \\ T_2 = p \& \{\ell_j(S_j).T_j\}_{j \in J} & \text{and} \\ T_3 = p \& \{\ell_k(S_k).T_k\}_{k \in I \cup J} \end{cases} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Indeed, one of the papers we base this work on [16] uses full merging. However we used plain merging in our formalisation and consequently in this work as it was already implemented in [4]. Generally speaking, the results we proved can be adapted to a full merge setting, see the proofs in [16].

Informally, the projection of a global type tree  $G$  onto a participant  $r$  extracts a specification 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  $G$ , then the only way it knows its role in the protocol is if there is a role for it that works no matter what choices  $p$  and  $q$  make in their communication. This "works no matter the choices of the other participants" property is captured by the merge operations.

In Rocq these constructions are expressed with the inductive `isMerge` and the coinductive `projectionC`.

```
Inductive isMerge : ltt → list (option ltt) → Prop ≜
| matm : ∀ t, isMerge t (Some t :: nil)
| mconsn : ∀ t xs, isMerge t xs → isMerge t (None :: xs)
| mconss : ∀ t xs, isMerge t xs → isMerge t (Some t :: xs).
```

`isMerge t xs` holds if the plain merge of the types in `xs` is equal to `t`.

```
Variant projection (R: gtt → part → ltt → Prop): gtt → part → ltt → Prop ≜
| proj_end : ∀ g r,
  (isgPartsC r g → False) →
  projection R g r (lts_end)
| proj_in : ∀ p r xs ys,
  p ≠ r →
  (isgPartsC r (gts_send p r xs)) →
  List.Forall12 (fun u v => (u = None ∧ v = None) ∨ (∃ s g t, u = Some(s, g) ∧ v = Some(s, t) ∧ R g r t)) xs ys →
```



```

    projection R (grr_send p r xs) r (lrr_recv p ys)
  | proj_out : ...
  | proj_cont : ∀ p q r xs ys t,
    p ≠ q →
    q ≠ r →
    p ≠ r →
    (isPartsC r (grr_send p q xs)) →
    List.Forall2 (fun u v => (u = None ∧ v = None) ∨
      (∃ s g t, u = Some(s, g) ∧ v = Some t ∧ R g r t)) xs ys →
    isMerge t ys →
    projection R (grr_send p q xs) r t.
Definition projectionC g r t ≜ paco3 projection bot3 g r t.

```

242

243 As in the definition of `subtypeC`, `projectionC` is defined as a parameterised greatest fixed point  
 244 using Paco. The premises of the rules [PROJ-IN], [PROJ-OUT] and [PROJ-CONT] are captured  
 245 using the Rocq standard library predicate `List.Forall2` :  $\forall A B : \text{Type}, (P:A \rightarrow B \rightarrow$   
 246  $\text{Prop}) (xs:\text{list } A) (ys:\text{list } B) : \text{Prop}$  that holds if  $P\ x\ y$  holds for every  $x, y$  where the  
 247 index of  $x$  in  $xs$  is the same as the index of  $y$  in the index of  $ys$ .

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

249 ► **Lemma 3.12.** *If `projectionC G p T` and `projectionC G p T'` then  $T = T'$ .*

250 We write  $G \upharpoonright r = T$  when  $G \upharpoonright_r T$ . Furthermore we will be frequently be making assertions  
 251 about subtypes of projections of a global type e.g.  $T \leq G \upharpoonright r$ . In our Rocq implementation  
 252 we define the predicate `issubProj` as a shorthand for this.

```

Definition issubProj (t:ltt) (g:grr) (p:part) ≜
  ∃ tg, projectionC g p tg ∧ subtypeC t tg.

```

253

### 254 3.5 Balancedness, Global Tree Contexts and Grafting

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

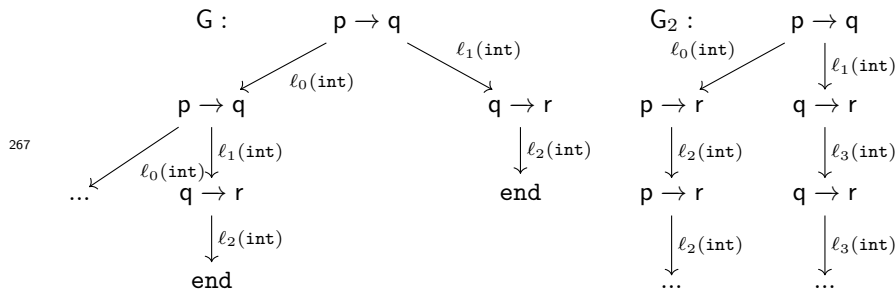
257 ► **Definition 3.13** (Balanced Global Type Trees). *A global tree  $G$  is balanced if for any subtree*  
 258  *$G'$  of  $G$ , there exists  $k$  such that for all  $p \in \text{pt}(G')$ ,  $p$  occurs on every path from the root of*  
 259  *$G'$  of length at least  $k$ .*

260 *In Rocq balancedness is expressed with the predicate `balancedG` ( $G : \text{grr}$ )*

261 We omit the technical details of this definition and the Rocq implementation, they can be  
 262 found in [5] and [4].

263 ► **Example 3.14.** The global type tree  $G$  given below is unbalanced as constantly following  
 264 the left branch gives an infinite path where  $r$  doesn't occur despite being a participant of the  
 265 tree. There is no such path for  $G_2$ , hence  $G_2$  is balanced.

266



268 Intuitively, balancedness is a regularity condition that imposes a notion of *liveness* on  
 269 the protocol described by the global type tree. For example,  $G$  in Example 3.14 describes

a defective protocol as it possible for  $p$  and  $q$  to constantly communicate through  $\ell_0$  and leave  $r$  waiting to receive from  $q$  a communication that will never come. We will be exploring these liveness properties from Section 4 onwards.

One other 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 3.15** (Global Type Tree Context). *Global type tree contexts are defined inductively with the following syntax:*

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

In Rocq global type tree contexts are represented by the type `gtth`

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

We additionally define `pt` and `ishParts` on 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.

► **Definition 3.16** (Grafting). *Given a global type tree context  $\mathcal{G}$  whose holes are in the 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 tree obtained by substituting  $[ ]_i$  with  $G_i$  in  $Gcx$ .*

In Rocq the indexed set  $\{G_i\}_{i \in I}$  is represented using a list `(option gtt)`. Grafting is expressed by the following inductive relation:

```
Inductive 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`.

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

► **Lemma 3.17** (Proper Grafting Lemma, [4]). *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`,  $\sim$  `ishParts`  $p$  `Gctx` and every `Some` element of `gs` is of shape `gtt_end`, `gtt_send`  $p$   $q$  or `gtt_send`  $q$   $p$ .*

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

► **Remark 3.18.** 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  $G : \text{gtt}$  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 =  $\forall p, \exists T, \text{projectionC } G \ p \ T$` . 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 Typing Contexts

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

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

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

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  $\Gamma$ . We write  $\Gamma(p)$  for the type of  $p$  in  $\Gamma$ . We define the composition  $\Gamma_1, \Gamma_2$  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.

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

In our implementation, we extensively use the MMaps library [8], which defines finite maps using red-black trees and provides many useful functions and theorems about them. We give some of the most important ones below:

- `M.add p t g`: Adds value  $t$  with the key  $p$  to the finite map  $g$ .
- `M.find p g`: If the key  $p$  is in the finite map  $g$  and is associated with the value  $t$ , returns `Some t`, else returns `None`.
- `M.In p g`: A `Prop` that holds iff  $p$  is in  $g$ .
- `M.mem p g`: A `bool` that is equal to `true` if  $p$  is in  $g$ , and `false` otherwise.
- `M.Equal g1 g2`: Unfolds to  $\forall p, \text{M.find } p \ g1 = \text{M.find } p \ g2$ . For our purposes, if `M.Equal g1 g2` then  $g1$  and  $g2$  are indistinguishable. This is made formal in the MMaps library with the assertion that `M.Equal` forms a setoid, and theorems asserting that most functions on maps respect `M.Equal` by showing that they form `Proper` morphisms [13, Generalized Rewriting].
- `M.merge f g1 g2` where  $f: \text{key} \rightarrow \text{option value} \rightarrow \text{option value} \rightarrow \text{option value}$ : Creates a finite map whose keys are the keys in  $g1$  or  $g2$ , where the value of the key  $p$  is defined as  $f \ p \ (\text{M.find } p \ g1) \ (\text{M.find } p \ g2)$ .
- `MF.Disjoint g1 g2`: A `Prop` that holds iff the keys of  $g1$  and  $g2$  are disjoint.
- `M.Eqdom g1 g2`: A `Prop` that holds iff  $g1$  and  $g2$  have the same domains.

One important function that we define is `disj_merge`, which merges disjoint maps and is used to represent the composition of typing contexts.

```
Definition both (z: nat) (o:option ltt) (o':option ltt)  $\triangleq$ 
match o,o' with
| Some _, None  $\Rightarrow$  o
| None, Some _  $\Rightarrow$  o'
| _, _  $\Rightarrow$  None
end.
```

this section  
might go

```

Definition disj_merge (g1 g2:tctx) (H:MF.Disjoint g1 g2) : tctx ≡
  M.merge both g1 g2.

```

345

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

347 ► **Definition 4.2** (Transition labels). *A transition label  $\alpha$  has the following form:*

348  $\alpha ::= p : q \& \ell(S)$  ( $p$  receives  $\ell(S)$  from  $q$ )  
 349  $\quad \mid p : q \oplus \ell(S)$  ( $p$  sends  $\ell(S)$  to  $q$ )  
 350  $\quad \mid (p, q) \ell$  ( $\ell$  is transmitted from  $p$  to  $q$ )

351

352 and in Rocq

```

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.

```

353

354 We also define the function  $\text{subject}(\alpha)$  as  $\text{subject}(p : q \& \ell(S)) = \text{subject}(p : q \oplus \ell(S)) = \{p\}$   
 355 and  $\text{subject}((p, q) \ell) = \{p, q\}$ .

356 In Rocq we represent  $\text{subject}(\alpha)$  with the predicate `ispSubj1 p alpha` that holds iff  $p \in \text{subject}(\alpha)$ .  
 357

```

Definition ispSubj1 r l ≡
  match l with
  | lsend p q _ _ => p=r
  | lrecv p q _ _ => p=r
  | lcomm p q _ => p=r ∨ q=r
  end.

```

358

359 ► **Remark 4.3.** From now on, we assume the all the types in the local type contexts always  
 360 have non-empty continuations. In Rocq terms, if  $\Gamma$  is in context `gamma` then `wfltt T` holds.  
 361 This is expressed by the predicate `wfltt: tctx → Prop`.

## 362 4.2 Local Type Context Reductions

363 Next we define labelled transitions for local type contexts.

364 ► **Definition 4.4** (Typing context reductions). *The typing context transition  $\xrightarrow{\alpha}$  is defined inductively by the following rules:*

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

366

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

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

373 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 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.
  
```

374

375 **Rsend**, **Rrecv** and **RvarI** are straightforward translations of  $[\Gamma - \&]$ ,  $[\Gamma - \oplus]$  and  $[\Gamma -,]$ .  
 376 **Rcomm** captures  $[\Gamma - \oplus\&]$  using the **disj\_merge** function we defined for the compositions, and  
 377 requires a proof that the contexts given are disjoint to be applied. **Rstruct** captures the  
 378 indistinguishability of local contexts under **M.Equal**.

379 We give an example to illustrate typing context reductions.

this can be  
cut

380 ▶ **Example 4.5.** Let

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

384

385 and  $\Gamma = p : T_p, q : T_q, r : T_r$ . We have the following one step reductions from  $\Gamma$ :

$$386 \quad \Gamma \xrightarrow{p:q \oplus \ell_0(\text{int})} \Gamma \quad (1)$$

$$387 \quad \Gamma \xrightarrow{q:p \& \ell_0(\text{int})} \Gamma \quad (2)$$

$$388 \quad \Gamma \xrightarrow{(p,q)\ell_0} \Gamma \quad (3)$$

$$389 \quad \Gamma \xrightarrow{r:q \& \ell_2(\text{int})} p : T_p, q : T_q, r : \text{end} \quad (4)$$

$$390 \quad \Gamma \xrightarrow{p:q \oplus \ell_1(\text{int})} p : \text{end}, q : T_q, r : T_r \quad (5)$$

$$391 \quad \Gamma \xrightarrow{q:p \& \ell_1(\text{int})} p : T_p, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : T_r \quad (6)$$

$$392 \quad \Gamma \xrightarrow{(p,q)\ell_1} p : \text{end}, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : T_r \quad (7)$$

393 and by (3) and (7) we have the synchronized reductions  $\Gamma \rightarrow \Gamma$  and

394  $\Gamma \rightarrow \Gamma' = p : \text{end}, q : r \oplus \{\ell_2(\text{int}).\text{end}\}, r : T_r$ . Further reducing  $\Gamma'$  we get

$$\Gamma' \xrightarrow{q:r \oplus \ell_2(\text{int})} p : \text{end}, q : \text{end}, r : T_r \quad (8)$$

$$\Gamma' \xrightarrow{r:q \& \ell_2(\text{int})} p : \text{end}, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : \text{end} \quad (9)$$

$$\Gamma' \xrightarrow{(q,r)\ell_2} p : \text{end}, q : \text{end}, r : \text{end} \quad (10)$$

and by (10) we have the reduction  $\Gamma' \rightarrow p : \text{end}, q : \text{end}, r : \text{end} = \Gamma_{\text{end}}$ , which results in a context that can't be reduced any further.

In Rocq,  $\Gamma$  is defined the following way:

```

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

```

Now Equation (1) can be stated with the following piece of Rocq

```

Lemma red_1 : 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.

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

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

In Rocq  $G \xrightarrow{(p,q)\ell_k} G'$  is expressed with the coinductively defined (via Paco) predicate `gttstepC`

$G \ G' \ p \ q \ k$ .

[GR- $\oplus\&$ ] says that a global type tree with root  $p \rightarrow q$  can transition to any of its children corresponding to the message label chosen by  $p$ . [GR-Ctx] says that if the subjects of  $\alpha$  are disjoint from the root and all its children can transition via  $\alpha$ , then the whole tree can also transition via  $\alpha$ , with the root remaining the same and just the subtrees of its children transitioning.

### 4.4 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.

421 ► **Definition 4.7** (Association). *A local typing context  $\Gamma$  is associated with a global type tree*  
 422  *$G$ , written  $\Gamma \sqsubseteq G$ , if the following hold:*  
 423 ■ *For all  $p \in \text{pt}(G)$ ,  $p \in \text{dom}(\Gamma)$  and  $\Gamma(p) \leq G \upharpoonright p$ .*  
 424 ■ *For all  $p \notin \text{pt}(G)$ , either  $p \notin \text{dom}(\Gamma)$  or  $\Gamma(p) = \text{end}$ .*  
 425 *In Rocq this is defined with the following:*

```

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

```

426  
 427 Informally,  $\Gamma \sqsubseteq G$  says that the local type trees in  $\Gamma$  obey the specification described by the  
 428 global type tree  $G$ .

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

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

431 Note that  $G$  is the global type that was shown to be unbalanced in Example 3.14. In fact,  
 432 we have  $\Gamma(s) = G \upharpoonright s$  for  $s \in \{p, q, r\}$ . Similarly, we have  $\Gamma' \sqsubseteq G'$  where

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

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

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

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

443 ► **Remark 4.11.** Note that in the statement of soundness we allow the message label for the  
 444 local type context reduction to be different to the message label for the global type reduction.  
 445 This is because our use of subtyping in association causes the entries in the local type context  
 446 to be less expressive than the types obtained by projecting the global type. For example  
 447 consider

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

449 and

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

451 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  
 452 soundness still requires that  $\Gamma \xrightarrow{(p,q)\ell_x}$  for some  $x$ , which is satisfied in this case by the valid  
 453 transition  $\Gamma \xrightarrow{(p,q)\ell_0}$ .

## 454 5 Properties of Local Type Contexts

455 We now use the LTS semantics to define some desirable properties on type contexts and their  
 456 reduction sequences. Namely, we formulate safety, liveness and fairness properties based on  
 457 the definitions in [16].

## 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  $\Gamma \in \text{safe}$ :*

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

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  $\ell$ , 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  $\varphi$  such that  $\Gamma \in \varphi$  and  $\varphi$  satisfies  $[S-\&\oplus]$  and  $[S-\rightarrow]$ . This amounts to showing that every element of  $\Gamma'$  of the set of reducts of  $\Gamma$ , defined  $\varphi := \{\Gamma' \mid \Gamma \rightarrow^* \Gamma'\}$ , satisfies  $[S-\&\oplus]$ . We illustrate this with some examples:

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

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

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

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

Being a coinductive property, **safe** can be expressed in Rocq using 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' → (∀ p q c' k,
      → safe R c.

Definition safeC c := paco1 safe bot1 c.
```

**weak\_safety** corresponds  $[S-\&\oplus]$  where  $\text{tctxRE } l \ c$  is shorthand for  $\exists c', \text{tctxR } c \ l \ c'$ . In the inductive **safe**, the constructor **safety\_red** corresponds to  $[S-\rightarrow]$ . 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  $\text{assoc } \text{gamma } g$  then  $\text{safeC } \text{gamma}$ .*

**Proof.**  $[S-\&\oplus]$  follows by inverting the projection and the subtyping, and  $[S-\rightarrow]$  holds by Theorem 4.10. ◀

## 5.2 Linear Time Properties

We now focus our attention to fairness and liveness. In this paper we have defined LTS semantics on three types of constructs: sessions, local type contexts and global types. We will appropriately define liveness properties on all three of these systems, so it will be convenient



to define a general notion of valid reduction paths (also known as *runs* or *executions* [1, 2.1.1]) along with a general statement of some Linear Temporal Logic [12] constructs.

We start by defining the general notion of a reduction path [1, Def. 2.6] using possibly infinite cosequences.

► **Definition 5.4** (Reduction Paths). *A finite reduction path is an alternating sequence of states and labels  $S_0 \lambda_0 S_1 \lambda_1 \dots S_n$  such that  $S_i \xrightarrow{\lambda_i} S_{i+1}$  for all  $0 \leq i < n$ . An infinite reduction path is an alternating sequence of states and labels  $S_0 \lambda_0 S_1 \lambda_1 \dots S_n$  such that  $S_i \xrightarrow{\lambda_i} S_{i+1}$  for all  $0 \leq i$ .*

We won't be distinguishing between finite and infinite reduction paths and refer to them both as just (*reduction*) *paths*. Note that the above definition is general for LTSs, by *state* we will be referring to local type contexts, global types or sessions, depending on the contexts.

In Rocq, we define reduction paths using possibly infinite cosequences of pairs of states (which will be *tctx*, *gtx* or *session* in this paper) and *option label*:

```
CoInductive coseq (A: Type): Type :=
| conil : coseq A
| cocons: A → coseq A → coseq A.
Notation local_path := (coseq (tctx*option label)).
Notation global_path := (coseq (gtx*option label)).
Notation session_path := (coseq (session*option label)).
```

Note the use of *option label*, where we employ *None* to represent transitions into the end of the list, *conil*. For example,  $S_0 \xrightarrow{\lambda_0} S_1 \xrightarrow{\lambda_1} S_2$  would be represented in Rocq as *cocons (s\_0, Some lambda\_0) (cocons (s\_1, Some lambda\_1) (cocons (s\_2, None) conil))*, and *cocons (s\_1, Some lambda) conil* would not be considered a valid path.

Note that this definition doesn't require the transitions in the *coseq* to actually be valid. We achieve that using the coinductive predicate *valid\_path\_GC*  $A:Type (V: A \rightarrow label \rightarrow A \rightarrow Prop)$ , where the parameter *V* is a *transition validity predicate*, capturing if a one-step transition is valid. For all *V*, *valid\_path\_GC* *V* *conil* and  $\forall x, \text{valid\_path\_GC } V (\text{cocons } (x, \text{None}) \text{ conil})$  hold, and *valid\_path\_GC* *V* *cocons* (*x*, *Some l*) (*cocons* (*y*, *l'*) *xs*) holds if the transition validity predicate  $V \ x \ l \ y$  and *valid\_path\_GC* *V* (*cocons* (*y*, *l'*) *xs*) hold. We use different *V* based on our application, for example in the context of local type context reductions the predicate is defined as follows:

```
Definition local_path_vcriteria := (fun x1 l x2 =>
match (x1,l,x2) with
| ((g1,lcomm p q ell),g2) => tctxR g1 (lcomm p q ell) g2
| _ => False
end
).
```

That is, we only allow synchronised communications in a valid local type context reduction path.

We can now define fairness and liveness on paths. We first restate the definition of fairness and liveness for local type context paths from [16], and use that to motivate our use of more general LTL constructs.

► **Definition 5.5** (Fair, Live Paths). *We say that a local type context path  $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_2} \dots$  is fair if, for all  $n \in N : \Gamma_n \xrightarrow{(p,q)\ell}$  implies  $\exists k, \ell'$  such that  $N \ni k \geq n$  and  $\lambda_k = (p,q)\ell'$ , and therefore  $\Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$ . We say that a path  $(\Gamma_n)_{n \in N}$  is live iff,  $\forall n \in N$ :*

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

► **Definition 5.6** (Live Local Type Context). *A local type context  $\Gamma$  is live if whenever  $\Gamma \rightarrow^* \Gamma'$ , every fair path starting from  $\Gamma'$  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 [6]. For our purposes we define fairness such that, in a fair path, if at any point  $p$  attempts to send to  $q$  and  $q$  attempts to send to  $p$  then eventually a communication between  $p$  and  $q$  takes place. Then live paths are defined to be paths such that whenever  $p$  attempts to send to  $q$  or  $q$  attempts to send to  $p$ , eventually a  $p$  to  $q$  communication takes place. Informally, this means that every communication request is eventually answered. Then live typing contexts are defined to be the  $\Gamma$  where all fair paths that start from  $\Gamma$  are also live.

► **Example 5.7.** Consider the contexts  $\Gamma, \Gamma'$  and  $\Gamma_{\text{end}}$  from Example 4.5. One possible 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  $\Gamma_n = \Gamma$  for all  $n \in \mathbb{N}$ . By reductions (3) and (7), 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  $\Gamma_n$ . 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 by reduction (4) that  $\Gamma_1 \xrightarrow{r:q\&\ell_2(\text{int})}$  but there is no  $n, \ell'$  with  $\Gamma_n \xrightarrow{(q,r)\ell'} \Gamma_{n+1}$  in the path. Consequently,  $\Gamma$  is not a live type context.

Now consider the reduction path  $\Gamma \xrightarrow{(p,q)\ell_0} \Gamma \xrightarrow{(p,q)\ell_0} \Gamma' \xrightarrow{(q,r)\ell_2} \Gamma_{\text{end}}$ , denoted by  $(\Gamma'_n)_{n \in \{1..4\}}$ . This path is fair with respect to reductions from  $\Gamma'_1$  and  $\Gamma'_2$  as shown above, and it's fair with respect to reductions from  $\Gamma'_3$  as reduction (10) is the only one available from  $\Gamma'_3$  and we have  $\Gamma'_3 \xrightarrow{(q,r)\ell_2} \Gamma'_4$  as needed. Furthermore, this path is live: the reduction  $\Gamma_1 \xrightarrow{r:q\&\ell_2(\text{int})}$  that causes  $(\Gamma_n)$  to fail liveness is handled by the reduction  $\Gamma'_3 \xrightarrow{(q,r)\ell_2} \Gamma'_4$  in this case.

Definition 5.5, 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 coinductive (via Paco) types. To do that, we turn to Linear Temporal Logic (LTL) [12].

these may go

► **Definition 5.8** (Linear Temporal Logic). *The syntax of LTL formulas  $\psi$  are defined inductively with boolean connectives  $\wedge, \vee, \neg$ , atomic propositions  $P, Q, \dots$ , and temporal operators  $\Box$  (always),  $\Diamond$  (eventually),  $\circ$  next and  $\mathcal{U}$ . Atomic propositions are evaluated over pairs of states and transitions  $(S, i, \lambda_i)$  (for the final state  $S_n$  in a finite reduction path we take that there is a null transition from  $S_n$ , corresponding to a **None** transition in Rocq) while LTL formulas are evaluated over reduction paths<sup>1</sup>. The satisfaction relation  $\rho \models \psi$  (where  $\rho = S_0 \xrightarrow{\lambda_0} S_1 \dots$  is a reduction path, and  $\rho_i$  is the suffix of  $\rho$  starting from index  $i$ ) is given by the following:*

- $\rho \models P \iff (S_0, \lambda_0) \models P.$
- $\rho \models \psi_1 \wedge \psi_2 \iff \rho \models \psi_1 \text{ and } \rho \models \psi_2$
- $\rho \models \neg\psi_1 \iff \text{not } \rho \models \psi_1$
- $\rho \models \circ\psi_1 \iff \rho_1 \models \psi_1$
- $\rho \models \Diamond\psi_1 \iff \exists k \geq 0, \rho_k \models \psi_1$

<sup>1</sup> These semantics assume that the reduction paths are infinite. In our implementation we do a sleight-of-hand and, for the purposes of the  $\Box$  operator, treat a terminating path as entering a dump state  $S_\perp$  (which corresponds to **conil** in Rocq) and looping there infinitely.

- 570  $\rho \models \Box \psi_1 \iff \forall k \geq 0, \rho_k \models \psi_1$   
 571  $\rho \models \psi_1 \mathcal{U} \psi_2 \iff \exists k \geq 0, \rho_k \models \psi_2 \text{ and } \forall j < k, \rho_j \models \psi_1$

572 Fairness and liveness for local type context paths Definition 5.5 can be defined in Linear  
 573 Temporal Logic (LTL). Specifically, define atomic propositions  $\text{enabledComm}_{p,q,\ell}$  such that  
 574  $(\Gamma, \lambda) \models \text{enabledComm}_{p,q,\ell} \iff \Gamma \xrightarrow{(p,q)\ell}$ , and  $\text{headComm}_{p,q}$  that holds iff  $\lambda = (p,q)\ell$  for some  
 575  $\ell$ . Then fairness can be expressed in LTL with: for all  $p, q$ ,

576  $\Box(\text{enabledComm}_{p,q,\ell} \implies \Diamond(\text{headComm}_{p,q}))$

577 Similarly, by defining  $\text{enabledSend}_{p,q,\ell,S}$  that holds iff  $\Gamma \xrightarrow{p:q \oplus \ell(S)}$  and analogously  
 578  $\text{enabledRecv}$ , liveness can be defined as

579  $\Box((\text{enabledSend}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{p,q})) \wedge$   
 580  $(\text{enabledRecv}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{q,p})))$

581 The reason we defined the properties using LTL properties is that the operators  $\Diamond$  and  $\Box$   
 582 can be characterised as least and greatest fixed points using their expansion laws [1, Chapter  
 583 5.14]:

- 584  $\Diamond P$  is the least solution to  $\Diamond P \equiv P \vee \Diamond(\Diamond P)$   
 585  $\Box P$  is the greatest solution to  $\Box P \equiv P \wedge \Box(\Box P)$   
 586  $PUQ$  is the least solution to  $PUQ \equiv Q \vee (P \wedge \Box(PUQ))$

587 Thus fairness and liveness correspond to greatest fixed points, which can be defined coin-  
 588 ductively.

589 In Rocq, we implement the LTL operators  $\Diamond$  and  $\Box$  inductively and coinductively (with  
 590 Paco), in the following way:

```

591 Inductive eventually {A: Type} (F: coseq A → Prop): coseq A → Prop ≡
  | evh: ∀ xs, F xs → eventually F xs
  | evc: ∀ x xs, eventually F xs → eventually F (cocons x xs).

Inductive until {A: Type} (F: coseq A → Prop) (G: coseq A → Prop) : coseq A → Prop ≡
  | untilh: ∀ xs, G xs → until F G xs
  | untilc: ∀ x xs, F (cocons x xs) → until F G xs → until F G (cocons x xs).

Inductive alwaysG {A: Type} (F: coseq A → Prop) (R: coseq A → Prop): coseq A → Prop ≡
  | alwn: F conil → alwaysG F R conil
  | alwc: ∀ x xs, F (cocons x xs) → R xs → alwaysG F R (cocons x xs).

Definition alwaysCG {A: Type} (F: coseq A → Prop) ≡ paco1 (alwaysG F) bot1.

```

592 Note the use of the constructor `alwn` in the definition `alwaysG` to handle finite paths.

593 Using these LTL constructs we can define fairness and liveness on paths.

```

594 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.

```

595 For instance, the fairness of the first reduction path for  $\Gamma$  given in Example 5.7 can be  
 596 expressed with the following:

```

597 CoFixpoint inf_pq_path ≡ cocons (gamma, (lcomm prt_p prt_q) 0) inf_pq_path.
Theorem inf_pq_path_fair : fairness inf_pq_path.

```

598

► Remark 5.9. Note that the LTS of local type contexts has the property that, once a transition between participants  $p$  and  $q$  is enabled, it stays enabled until a transition between  $p$  and  $q$  occurs. This makes `fair_path` equivalent to the standard formulas [1, Definition 5.25] for strong fairness ( $\Box \Diamond \text{enabledComm}_{p,q} \implies \Box \Diamond \text{headComm}_{p,q}$ ) and weak fairness ( $\Diamond \Box \text{enabledComm}_{p,q} \implies \Box \Diamond \text{headComm}_{p,q}$ ).

### 5.3 Rocq Proof of Liveness by Association

We now detail the Rocq Proof that associated local type contexts are also live.

► Remark 5.10. We once again emphasise that all global types mentioned are assumed to be balanced (Definition 3.13). Indeed association with non-balanced global types doesn't guarantee liveness. As an example, consider  $\Gamma$  from Example 4.5, which is associated with  $G$  from Example 4.8. Yet we have shown in Example 5.7 that  $\Gamma$  is not a live type context. This is not surprising as Example 3.14 shows that  $G$  is not balanced.

Our proof proceeds in the following way:

1. Formulate an analogue of fairness and liveness for global type reduction paths.
2. Prove that all global types are live for this notion of liveness.
3. Show that if  $G : \text{gtt}$  is live and `assoc gamma G`, then `gamma` is also live.

First we define fairness and liveness for global types, analogous to Definition 5.5.

► Definition 5.11 (Fairness and Liveness for Global Types). *We say that the label  $\lambda$  is enabled at  $G$  if the context  $\{p_i : G \vdash_{p_i} \mid p_i \in \text{pt}\{G\}\}$  can transition via  $\lambda$ . More explicitly, and in Rocq terms,*

```
Definition global_label_enabled l g  $\triangleq$  match l with
| lsend p q (Some s) n =>  $\exists$  xs g',
  projectionC g p (litt_send q xs)  $\wedge$  onth n xs=Some (s,g')
| lrecv p q (Some s) n =>  $\exists$  xs g',
  projectionC g p (litt_recv q xs)  $\wedge$  onth n xs=Some (s,g')
| lcomm p q n =>  $\exists$  g', gttstepC g g' p q n
| _ => False end.
```

With this definition of enabling, fairness and liveness are defined exactly as in Definition 5.5. A global type reduction path is fair if the following holds:

$$\Box(\text{enabledComm}_{p,q,\ell} \implies \Diamond(\text{headComm}_{p,q}))$$

and liveness is expressed with the following:

$$\Box((\text{enabledSend}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{p,q})) \wedge (\text{enabledRecv}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{q,p})))$$

where `enabledSend`, `enabledRecv` and `enabledComm` correspond to the match arms in the definition of `global_label_enabled` (Note that the names `enabledSend` and `enabledRecv` are chosen for consistency with Definition 5.5, there aren't actually any transitions with label  $p : q \oplus \ell(S)$  in the transition system for global types). A global type  $G$  is live if whenever  $G \rightarrow^* G'$ , any fair path starting from  $G'$  is also live.

Now our goal is to prove that all (well-formed, balanced, projectable)  $G$  are live under this definition. This is where the notion of grafting (Definition 3.13) becomes important, as the proof essentially proceeds by well-founded induction on the height of the tree obtained by grafting.

We first introduce some definitions on global type tree contexts (Definition 3.15).

636 ► **Definition 5.12** (Global Type Context Equality, Proper Prefixes and Height). *We consider*  
 637 *two global type tree contexts to be equal if they are the same up to the relabelling the indices*  
 638 *of their leaves. More precisely,*

```

Inductive gtth_eq : gtth → gtth → Prop ≜
| gtth_eq_hol : ∀ n m, gtth_eq (gtth_hol n) (gtth_hol m)
| gtth_eq_send : ∀ xs ys p q :
  Forall2 (fun u v => (u=None ∧ v=None) ∨ (∃ s g1 g2, u=Some (s,g1) ∧ v=Some (s,g2) ∧ gtth_eq g1 g2)) xs ys →
  gtth_eq (gtth_send p q xs) (gtth_send p q ys).

```

640 Informally, we say that the global type context  $\mathbb{G}'$  is a proper prefix of  $\mathbb{G}$  if we can obtain  $\mathbb{G}'$   
 641 by changing some subtrees of  $\mathbb{G}$  with context holes such that none of the holes in  $\mathbb{G}$  are present  
 642 in  $\mathbb{G}'$ . Alternatively, we can characterise it as akin to `gtth_eq` except where the context holes  
 643 in  $\mathbb{G}'$  are assumed to be "jokers" that can be matched with any global type context that's not  
 644 just a context hole. In Rocq:

```

Inductive is_tree_proper_prefix : gtth → gtth → Prop ≜
| tree_proper_prefix_hole : ∀ n p q xs, is_tree_proper_prefix (gtth_hol n) (gtth_send p q xs)
| tree_proper_prefix_tree : ∀ p q xs ys,
  Forall2 (fun u v => (u=None ∧ v=None)
    ∨ (∃ s g1 g2, u=Some (s, g1) ∧ v=Some (s, g2) ∧
      is_tree_proper_prefix g1 g2)) xs ys →
  is_tree_proper_prefix (gtth_send p q xs) (gtth_send p q ys).

```

646 We also define a function `gtth_height` : `gtth` → `Nat` that computes the height [3] of a  
 647 global type tree context. Context holes i.e. leaves have height 0, and the height of an internal  
 648 node is the maximum of the height of their children plus one.

give examples

```

Fixpoint gtth_height (gh : gtth) : nat ≜
match gh with
| gtth_hol n => 0
| gtth_send p q xs =>
  list_max (map (fun u => match u with
    | None => 0
    | Some (s,x) => gtth_height x end) xs) + 1 end.

```

651 `gtth_height`, `gtth_eq` and `is_tree_proper_prefix` interact in the expected way.

652 ► **Lemma 5.13.** *If `gtth_eq gx gx'` then `gtth_height gx = gtth_height gx'`.*

653 ► **Lemma 5.14.** *If `is_tree_proper_prefix gx gx'` then `gtth_height gx < gtth_height gx'`.*

654 Our motivation for introducing these constructs on global type tree contexts is the following  
 655 *multigrafting* lemma:

656 ► **Lemma 5.15** (Multigrafting). *Let `projectionC g p (ltx_send q xsp)` or `projectionC g`*  
 657 *`p (ltx_recv q xsp)`, `projectionC g q Tq`, `g` is `p`-grafted by `ctx_p` and `gs_p`, and `g` is `q`-*  
 658 *grafted by `ctx_q` and `gs_q`. Then either `is_tree_proper_prefix ctx_q ctx_p` or `gtth_eq`*  
 659 *`ctx_p ctx_q`. Furthermore, if `gtth_eq ctx_p ctx_q` then `projectionC g q (ltx_send p xsq)`*  
 660 *or `projectionC g q (ltx_recv p xsq)` for some `xsq`.*

661 **Proof.** By induction on the global type context `ctx_p`. ◀

example

663 We also have that global type reductions that don't involve participant `p` can't increase  
 664 the height of the `p`-grafting, established by the following lemma:

665 ► **Lemma 5.16.** *Suppose `g : gtt` is `p`-grafted by `gx : gtth` and `gs : list (option gtt)`, `gttstepC`  
 666 `g g' s t ell` where `p ≠ s` and `p ≠ t`, and `g'` is `p`-grafted by `gx'` and `gs'`. Then*

- 667 (i) If  $\text{ishParts } s \text{ } gx$  or  $\text{ishParts } t \text{ } gx$ , then  $\text{gtth\_height } gx' < \text{gtth\_height } gx$   
 668 (ii) In general,  $\text{gtth\_height } gx' \leq \text{gtth\_height } gx$

669 **Proof.** We define a inductive predicate  $\text{gttstepH} : \text{gtth} \rightarrow \text{part} \rightarrow \text{part} \rightarrow \text{part} \rightarrow$   
 670  $\text{gtth} \rightarrow \text{Prop}$  with the property that if  $\text{gttstepC } g \text{ } g' \text{ } p \text{ } q \text{ } \text{ell}$  for some  $r \neq p, q$ , and  
 671 tree contexts  $gx$  and  $gx'$   $r$ -graft  $g$  and  $g'$  respectively, then  $\text{gttstepH } gx \text{ } p \text{ } q \text{ } \text{ell } gx'$   
 672 ( $\text{gttstepH\_consistent}$ ). The results then follow by induction on the relation  $\text{gttstepH}$   
 673  $gx \text{ } s \text{ } t \text{ } \text{ell } gx'$ .  $\blacktriangleleft$

674 We can now prove the liveness of global types. The bulk of the work goes in to proving the  
 675 following lemma:

676 **► Lemma 5.17.** *Let  $xs$  be a fair global type reduction path starting with  $g$ .*

- 677 (i) If  $\text{projectionC } g \text{ } p \text{ } (\text{ltt\_send } q \text{ } xsp)$  for some  $xsp$ , then a  $\text{lcomm } p \text{ } q \text{ } \text{ell}$  transition  
 678 takes place in  $xs$  for some message label  $\text{ell}$ .  
 679 (ii) If  $\text{projectionC } g \text{ } p \text{ } (\text{ltt\_recv } q \text{ } xsp)$  for some  $xsp$ , then a  $\text{lcomm } q \text{ } p \text{ } \text{ell}$  transition  
 680 takes place in  $xs$  for some message label  $\text{ell}$ .

681 **Proof.** We outline the proof for (i), the case for (ii) is symmetric.

682 Rephrasing slightly, we prove the following: forall  $n : \text{nat}$  and global type reduction path  
 683  $xs$ , if the head  $g$  of  $xs$  is  $p$ -grafted by  $\text{ctx\_p}$  and  $\text{gtth\_height } \text{ctx\_p} = n$ , the lemma holds.  
 684 We proceed by strong induction on  $n$ , that is, the tree context height of  $\text{ctx\_p}$ .

685 Let  $(\text{ctx\_q}, \text{gs\_q})$  be the  $q$ -grafting of  $g$ . By Lemma 5.15 we have that either  $\text{gtth\_eq}$   
 686  $\text{ctx\_q } \text{ctx\_p}$  (a) or  $\text{is\_tree\_proper\_prefix } \text{ctx\_q } \text{ctx\_p}$  (b). In case (a), we have that  
 687  $\text{projectionC } g \text{ } q \text{ } (\text{ltt\_recv } p \text{ } xsq)$ , hence by (cite simul subproj or something here) and  
 688 fairness of  $xs$ , we have that a  $\text{lcomm } p \text{ } q \text{ } \text{ell}$  transition eventually occurs in  $xs$ , as required.

689 In case (b), by Lemma 5.14 we have  $\text{gtth\_height } \text{ctx\_q} < \text{gtth\_height } \text{ctx\_p}$ , so by the  
 690 induction hypothesis a transition involving  $q$  eventually happens in  $xs$ . Assume wlog that  
 691 this transition has label  $\text{lcomm } q \text{ } r \text{ } \text{ell}$ , or, in the pen-and-paper notation,  $(q, r)\ell$ . Now  
 692 consider the prefix of  $xs$  where the transition happens:  $g \xrightarrow{\lambda} g_1 \rightarrow \dots g' \xrightarrow{(q, r)\ell} g''$ . Let  
 693  $g'$  be  $p$ -grafted by the global tree context  $\text{ctx}'\_p$ , and  $g''$  by  $\text{ctx}''\_p$ . By Lemma 5.16,  
 694  $\text{gtth\_height } \text{ctx}''\_p < \text{gtth\_height } \text{ctx}'\_p \leq \text{gtth\_height } \text{ctx\_p}$ . Then, by the induction  
 695 hypothesis, the suffix of  $xs$  starting with  $g''$  must eventually have a transition  $\text{lcomm } p \text{ } q \text{ } \text{ell}'$   
 696 for some  $\text{ell}'$ , therefore  $xs$  eventually has the desired transition too.  $\blacktriangleleft$

697 Lemma 5.17 proves that any fair global type reduction path is also a live path, from which  
 698 the liveness of global types immediately follows.

699 **► Corollary 5.18.** *All global types are live.*

700 We can now leverage the simulation established by Theorem 4.10 to prove the liveness  
 701 (Definition 5.5) of local typing context reduction paths.

702 We start by lifting association (Definition 4.7) to reduction paths.

703 **► Definition 5.19 (Path Association).** *Path association is defined coinductively by the following*  
 704 *rules:*

- 705 (i) *The empty path is associated with the empty path.*  
 706 (ii) *If  $\Gamma \xrightarrow{\lambda_0} \rho$  is path-associated with  $G \xrightarrow{\lambda_1} \rho'$  where  $(\rho$  and  $\rho'$  are local and global reduction*  
 707 *paths, respectively), then  $\lambda_0 = \lambda_1$  and  $\rho$  is path-associated with  $\rho'$ .*

```

Variant path_assoc (R:local_path → global_path → Prop): local_path → global_path → Prop ≡
| path_assoc_nil : path_assoc R conil conil
| path_assoc_xs : ∀ g gamma l xs ys, assoc gamma g → R xs ys →
  path_assoc R (cocons (gamma, l) xs) (cocons (g, l) ys).
Definition path_assocC ≡ paco2 path_assoc bot2.

```

708

709 Informally, a local type context reduction path is path-associated with a global type reduction  
 710 path if their matching elements are associated and have the same transition labels.

711 We show that reduction paths starting with associated local types can be path-associated.  
 712

713 ► **Lemma 5.20.** *If assoc gamma g, then any local type context reduction path starting with*  
 714 *gamma is associated with a global type reduction path starting with g.*

715 **Proof.** Let the local reduction path be  $\text{gamma} \xrightarrow{\lambda} \text{gamma}_1 \xrightarrow{\lambda_1} \dots$ . We construct a path-  
 716 associated global reduction path. By Theorem 4.10 there is a  $g_1 : \text{gtt}$  such that  $g \xrightarrow{\lambda} g_1$   
 717 and  $\text{assoc gamma}_1 g_1$ , hence the path-associated global type reduction path starts with  $g$   
 718  $\xrightarrow{\lambda} g_1$ . We can repeat this procedure to the remaining path starting with  $\text{gamma}_1 \xrightarrow{\lambda_1} \dots$   
 719 to get  $g_2 : \text{gtt}$  such that  $\text{assoc gamma}_2 g_2$  and  $g_1 \xrightarrow{\lambda_1} g_2$ . Repeating this, we get  $g \xrightarrow{\lambda}$   
 720  $g_1 \xrightarrow{\lambda_1} \dots$  as the desired path associated with  $\text{gamma} \xrightarrow{\lambda} \text{gamma}_1 \xrightarrow{\lambda_1} \dots$  ◀

maybe just  
give the defin-  
ition as a  
cofixpoint?

721 ► **Remark 5.21.** In the Rocq implementation the construction above is implemented as a  
 722 **CoFixpoint** returning a **coseq**. Theorem 4.10 is implemented as an  $\exists$  statement that lives in  
 723 **Prop**, hence we need to use the **constructive\_indefinite\_description** axiom to obtain the  
 724 witness to be used in the construction.

725 We also have the following correspondence between fairness and liveness properties for  
 726 associated global and local reduction paths.

727 ► **Lemma 5.22.** *For a local reduction path xs and global reduction path ys, if path\_assocC*  
 728 *xs ys then*

- 729 (i) *If xs is fair then so is ys*
- 730 (ii) *If ys is live then so is xs*

731 As a corollary of Lemma 5.22, Lemma 5.20 and Lemma 5.17 we have the following:

732 ► **Corollary 5.23.** *If assoc gamma g, then any fair local reduction path starting from gamma is*  
 733 *live.*

734 **Proof.** Let xs be the fair local reduction path starting with gamma. By Lemma 5.20 there is  
 735 a global path ys associated with it. By Lemma 5.22 (i) ys is fair, and by Lemma 5.17 ys is  
 736 live, so by Lemma 5.22 (ii) xs is also live. ◀

737 Liveness of contexts follows directly from Corollary 5.23.

738 ► **Theorem 5.24 (Liveness by Association).** *If assoc gamma g then gamma is live.*

739 **Proof.** Suppose  $\text{gamma} \rightarrow^* \text{gamma}'$ , then by Theorem 4.10  $\text{assoc gamma}' g'$  for some  $g'$ , and  
 740 hence by Corollary 5.23 any fair path starting from  $\text{gamma}'$  is live, as needed. ◀

## 741 6 Properties of Sessions

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



## 6.1 Typing rules

We give typing rules for our session calculus based on [5] and [4].

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

1. A local type context  $\Gamma$  associates participants with local type trees, as defined in cdef-type-ctx. Local type contexts are used to type sessions (Definition 2.2) i.e. a set of pairs of participants and single processes composed in parallel. We express such judgements as  $\Gamma \vdash_{\mathcal{M}} \mathcal{M}$ , or as `typ_sess M gamma` or `gamma ⊢ M` in Rocq.
2. A process variable context  $\Theta_T$  associates process variables with local type trees, and an expression variable context  $\Theta_e$  assigns sorts to expression variables. Variable contexts are used to type single processes and expressions (Definition 2.1). Such judgements are expressed as  $\Theta_T, \Theta_e \vdash_P P : T$ , or in Rocq as `typ_proc theta_T theta_e P T` or `theta_T, theta_e ⊢ P : T`.

$$\Theta \vdash_P n : \text{nat} \quad \Theta \vdash_P i : \text{int} \quad \Theta \vdash_P \text{true} : \text{bool} \quad \Theta \vdash_P \text{false} : \text{bool} \quad \Theta, x : S \vdash_P x : S$$

$$\frac{\Theta \vdash_P e : \text{nat}}{\Theta \vdash_P \text{succ } e : \text{nat}} \quad \frac{\Theta \vdash_P e : \text{int}}{\Theta \vdash_P \text{neg } e : \text{int}} \quad \frac{\Theta \vdash_P e : \text{bool}}{\Theta \vdash_P \neg e : \text{bool}}$$

$$\frac{\Theta \vdash_P e_1 : S \quad \Theta \vdash_P e_2 : S}{\Theta \vdash_P e_1 \oplus e_2 : S} \quad \frac{\Theta \vdash_P e_1 : \text{int} \quad \Theta \vdash_P e_2 : \text{int}}{\Theta \vdash_P e_1 > e_2 : \text{bool}} \quad \frac{\Theta \vdash_P e : S \quad S \leq S'}{\Theta \vdash_P e : S'}$$

■ Table 5 Typing expressions

$$\frac{[T\text{-END}]}{\Theta \vdash_P 0 : \text{end}} \quad \frac{[T\text{-VAR}]}{\Theta, X : T \vdash_P X : T} \quad \frac{[T\text{-REC}]}{\Theta, X : T \vdash_P P : T} \quad \frac{[T\text{-IF}]}{\Theta \vdash_P e : \text{bool} \quad \Theta \vdash_P P_1 : T \quad \Theta \vdash_P P_2 : T}{\Theta \vdash_P \text{if } e \text{ then } P_1 \text{ else } P_2 : T}$$

$$\frac{[T\text{-SUB}]}{\Theta \vdash_P P : T \quad T \leq T'}{\Theta \vdash_P P : T'} \quad \frac{[T\text{-IN}]}{\Theta \vdash_P \sum_{i \in I} p_i ? \ell_i(x_i). P_i : p_i \& \{ \ell_i(S_i). T_i \}_{i \in I}} \quad \frac{[T\text{-OUT}]}{\Theta \vdash_P e : S \quad \Theta \vdash_P P : T}{\Theta \vdash_P p_i ! \ell(e). P : p_i \oplus \{ \ell(S). T \}}$$

■ Table 6 Typing processes

Table 5 and Table 6 state the standard typing rules for expressions and processes which we don't elaborate on. We have a single rule for typing sessions:

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

[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  $\Gamma$  if the local type of participant  $p_i$  in  $\Gamma$  types the process

## 6.2 Subject Reduction, Progress and Session Fidelity

give theorem  
no

The subject reduction, progress and non-stuck theorems from [4] also hold in this setting, with minor changes in their statements and proofs. We won't discuss these proofs in detail.



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

768 **Proof.** By induction on  $\text{unfoldP } M \ M'$ . ◀

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

771 ► **Theorem 6.3** (Progress). *If  $\text{gamma} \vdash_{\mathcal{M}} M$ , one of the following hold :*

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

774 ► **Remark 6.4.** Note that in Theorem 6.2 one transition between sessions corresponds to  
 775 exactly one transition between local type contexts with the same label. That is, every session  
 776 transition is observed by the corresponding type. This is the main reason for our choice of  
 777 reactive semantics (Section 2.3) as  $\tau$  transitions are not observed by the type in ordinary  
 778 semantics. In other words, with  $\tau$ -semantics the typing relation is a *weak simulation* [9],  
 779 while it turns into a strong simulation with reactive semantics. For our Rocq implementation  
 780 working with the strong simulation turns out to be more convenient.  
 781 We can also prove the following correspondence result in the reverse direction to Theorem 6.2,  
 782 analogous to Theorem 4.9.

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

786 **Proof.** By inverting the local type context transition and the typing. ◀

787 ► **Remark 6.6.** Again we note that by Theorem 6.5 a single-step context reduction induces a  
 788 single-step session reduction on the type. With the  $\tau$ -semantics the session reduction induced  
 789 by the context reduction would be multistep.

## 790 6.3 Session Liveness

791 We state the liveness property we are interested in proving, and show that typable sessions  
 792 have this property.

793 ► **Definition 6.7** (Session Liveness). *Session  $\mathcal{M}$  is live iff*

- 794 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  $\mathcal{M}'', \mathcal{N}'$*
- 795 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*  
 796  *$\mathcal{M}'', \mathcal{N}', i, v$ .*

797 *In Rocq we express this with the following:*

```

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) \\\ \\\ M'')).
  
```

799 Session liveness, analogous to liveness for typing contexts (Definition 5.5), says that when  
 800  $\mathcal{M}$  is live, if  $\mathcal{M}$  reduces to a session  $\mathcal{M}'$  containing a participant that's attempting to send  
 801 or receive, then  $\mathcal{M}'$  reduces to a session where that communication has happened. It's also  
 802 called *lock-freedom* in related work ([15, 10]).

803 We now prove that typed sessions are live. Our proof follows the following steps:

1. Formulate a "fairness" property for typable sessions, with the property that any finite session reduction path can be extended to a fair session reduction path.
  2. Lift the typing relation to reduction paths, and show that fair session reduction paths are typed by fair local type context reduction paths.
  3. Prove that a certain transition eventually happens in the local context reduction path, and that this means the desired transition is enabled in the session reduction path.
- We first state a "fairness" (the reason for the quotes is explained in Remark 6.9) property for session reduction paths, analogous to fairness for local type context reduction paths (Definition 5.5).

► **Definition 6.8** ("Fairness" of Sessions). *We say that a  $(p, q)\ell$  transition is enabled at  $\mathcal{M}$  if  $\mathcal{M} \xrightarrow{(p, q)\ell} \mathcal{M}'$  for some  $\mathcal{M}'$ . A session reduction path is fair if the following LTL property holds:*

$$\Box(\text{enabledComm}_{p, q, \ell} \implies \Diamond(\text{headComm}_{p, q}))$$

► **Remark 6.9.** Definition 6.8 is not actually a sensible fairness property for our reactive semantics, mainly because it doesn't satisfy the *feasibility* [6] property stating that any finite execution can be extended to a fair execution. Consider the following session:

$$\mathcal{M} = p \triangleleft \text{if}(\text{true} \oplus \text{false}) \text{ then } q! \ell_1(\text{true}) \text{ else } r! \ell_2(\text{true}).0 \mid q \triangleleft p? \ell_1(x).0 \mid r \triangleleft p? \ell_2(x).0$$

We have that  $\mathcal{M} \xrightarrow{(p, q)\ell_1} \mathcal{M}'$  where  $\mathcal{M}' = p \triangleleft 0 \mid q \triangleleft 0 \mid r \triangleleft p? \ell_2(x).0$ , and also  $\mathcal{M} \xrightarrow{(p, r)\ell_2} \mathcal{M}''$  for another  $\mathcal{M}''$ . Now consider the reduction path  $\rho = \mathcal{M} \xrightarrow{(p, q)\ell_1} \mathcal{M}'$ .  $(p, r)\ell_2$  is enabled at  $\mathcal{M}$  so in a fair path it should eventually be executed, however no extension of  $\rho$  can contain such a transition as  $\mathcal{M}'$  has no remaining transitions. Nevertheless, it turns out that there is a fair reduction path starting from every typable session (Lemma 6.13), and this will be enough to prove our desired liveness property.

We can now lift the typing relation to reduction paths, just like we did in Definition 5.19.

► **Definition 6.10** (Path Typing). *Path typing is a relation between session reduction paths and local type context reduction paths, defined coinductively by the following rules:*

- (i) *The empty session reduction path is typed with the empty context reduction path.*
- (ii) *If  $\mathcal{M} \xrightarrow{\lambda_0} \rho$  is typed by  $\Gamma \xrightarrow{\lambda_1} \rho'$  where  $(\rho$  and  $\rho'$  are session and local type context reduction paths, respectively), then  $\lambda_0 = \lambda_1$  and  $\rho$  is typed by  $\rho'$ .*

Similar to Lemma 5.20, we can show that if the head of the path is typable then so is the whole path.

► **Lemma 6.11.** *If  $\text{typ\_sess } M \text{ gamma}$ , then any session reduction path  $xs$  starting with  $M$  is typed by a local context reduction path  $ys$  starting with  $\text{gamma}$ .*

**Proof.** We can construct a local context reduction path that types the session path. The construction exactly like Lemma 5.20 but elements of the output stream are generated by Theorem 6.2 instead of Theorem 4.10. ◀

We also have that typing path preserves fairness.

► **Lemma 6.12.** *If session path  $xs$  is typed by the local context path  $ys$ , and  $xs$  is fair, then so is  $ys$ .*

The final lemma we need in order to prove liveness is that there exists a fair reduction path from every typable session.

► **Lemma 6.13** (Fair Path Existence). *If  $\text{typ\_sess } M \text{ gamma}$ , then there is a fair session reduction path  $xs$  starting from  $M$ .*

**Proof.** We can construct a fair path starting from  $M$  by repeatedly cycling through all participants, checking if there is a transition involving that participant, and executing that transition if there is. ◀

► **Remark 6.14.** The Rocq implementation of Lemma 6.13 computes a **CoFixpoint** corresponding to the fair path constructed above. As in Lemma 5.20, we use **constructive\_indefinite\_description** to turn existence statements in **Prop** to dependent pairs. We also assume the informative law of excluded middle (**excluded\_middle\_informative**) in order to carry out the "check if there is a transition" step in the algorithm above. When proving that the constructed path is fair, we sometimes rely on the LTL constructs we outlined in Section 5.2 reminiscent of the techniques employed in [2].

We can now prove that typed sessions are live.

► **Theorem 6.15** (Liveness by Typing). *For a session  $Mp$ , if  $\exists \text{ gamma } \text{gamma} \vdash_{\mathcal{M}} Mp$  then  $\text{live\_sess } Mp$ .*

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

Now let  $xs$  be a fair reduction path starting from  $((p \leftarrow p\_send \ q \ \text{ell} \ e \ P') \ ||| \ M')$ , which exists by Lemma 6.13. Let  $ys$  be the local context reduction path starting with  $\text{gamma}$  that types  $xs$ , which exists by Lemma 6.11. Now  $ys$  is fair by Lemma 6.12. Therefore by Theorem 5.24  $ys$  is live, so a  $\text{lcomm } p \ q \ \text{ell}'$  transition eventually occurs in  $ys$  for some  $\text{ell}'$ . 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  $\text{gamma}_0$  in  $xs$ . We have  $((p \leftarrow p\_send \ q \ \text{ell} \ e \ P') \ ||| \ M') \rightarrow^* M_0$  by  $M_0$  being on  $xs$ . We also have that  $M_0 \xrightarrow{(p,q)\ell''} M_1$  for some  $\ell'', M_1$  by Theorem 6.5. Now observe that  $M_0 \equiv ((p \leftarrow p\_send \ q \ \text{ell} \ e \ P') \ ||| \ M'')$  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 Related and Future Work

### References

- 1 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- 2 Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 102–115, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 4 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.
- 5 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.
  - 6 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.
  - 7 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.
  - 8 Pierre Letouzey and Andrew W. Appel. Modular Finite Maps over Ordered Types. URL: <https://github.com/rocq-community/mmaps>.
  - 9 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.
  - 10 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.
  - 11 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
  - 12 Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. iee, 1977.
  - 13 The Rocq Development Team. *The Rocq Reference Manual*. Inria, 2025. <https://rocq-prover.org/doc/V9.0.0/refman>.
  - 14 The Rocq Development Team. *The Rocq Standard Library*. Inria, 2025. <https://rocq-prover.org/doc/V9.0.0/stdlib>.
  - 15 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.
  - 16 Nobuko Yoshida and Ping Hou. Less is more revisited, 2024. URL: <https://arxiv.org/abs/2402.16741>, arXiv:2402.16741.