

¹ Formally Verified Liveness with Synchronous ² Multiparty Session Types in Rocq

³ **Anonymous author**

⁴ Anonymous affiliation

⁵ **Anonymous author**

⁶ Anonymous affiliation

⁷ — Abstract —

⁸ We mechanise a synchronous multiparty session type framework that guarantees liveness for typed
⁹ processes. We type sessions using a context of local types, and use "association" with global types to
¹⁰ denote a set of well-behaved local type contexts. We give LTS semantics to local contexts and global
¹¹ types and prove operational correspondences between the LTSs local context and their associated
¹² global types. We then prove that sessions typed by a local context that's associated with a global
¹³ type are live.

¹⁴ **2012 ACM Subject Classification** Replace ccsdesc macro with valid one

¹⁵ **Keywords and phrases** Dummy keyword

¹⁶ **Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

¹⁷ **Acknowledgements** Anonymous acknowledgements

¹⁸ 1 Introduction

²¹ In this work we present the Rocq formalisation of a session type system for a simple
²² session calculus, and prove that sessions typable in this system are *safe*, *deadlock-free*, and
²³ *live*. The approach we take in our type system is very similar to the one followed by Hou
²⁴ and Yoshida in [16]. Namely, we proceed by defining local and global type trees, and relate
²⁵ them using projections. We then extend this projection relation to an *association* relation
²⁶ between local type contexts i.e. collections of local types paired with participants, and global
²⁷ type trees. Next we give LTS semantics to local type contexts and global type trees, and
²⁸ prove an operational correspondence between them. We then proceed to formulate safety
²⁹ and liveness properties for local type contexts, and show that local type contexts associated
³⁰ with global type trees enjoy these properties. We relate associated local type contexts to
³¹ sessions via typing rules, and demonstrate an operational correspondence between contexts
³² and sessions via *subject reduction*, *progress* and *session fidelity* theorems. Finally we show,
³³ using the liveness properties we defined on local type contexts, that typable sessions are live.

³⁴ Our Rocq implementation builds upon the recent formalisation of subject reduction for
³⁵ MPST by Ekici et. al. [4], which itself is based on [5]. The methodology in [4] takes an

Session types introduction
Liveness introduction

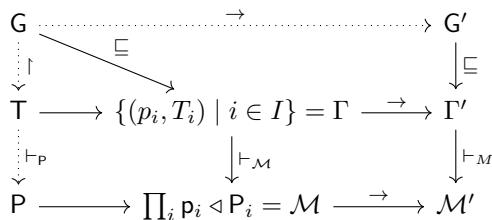


Figure 1 Design overview. The dotted lines correspond to relations inherited from [4] while the solid lines denote relations that are new, or substantially rewritten, in this paper.



© Anonymous author(s); licensed under Creative Commons License CC-BY 4.0

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

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:28



Leibniz International Proceedings in Informatics

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

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 2 The Session Calculus

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

50 2.1 Processes and Sessions

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

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

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

55 $p!\ell(e).P$ is a process that sends the value of expression e with label ℓ to participant p , and
56 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
57 any label ℓ_i where $i \in I$, binding the result to x_i and continuing with P_i , depending on
58 which ℓ_i the value was received from. X is a recursion variable, $\mu X.P$ is a recursive process,
59 if e then P else P is a conditional and 0 is a terminated process.

60 Processes can be composed in parallel into sessions.

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

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

62 $p \triangleleft P$ denotes that participant p is running the process P , \mid indicates parallel composition. We
63 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
64 an empty session with no participants, that is, the unit of parallel composition.

65 ► **Remark 2.3.** Note that \mathcal{O} is different than $p \triangleleft 0$ as p is a participant in the latter but not
66 the former. This differs from previous work, e.g. in [5] the unit of parallel composition is
67 $p \triangleleft 0$ while in [4] there is no unit. The unitless approach of [4] results in a lot of repetition
68 in the code, for an example see their definition of `unfoldP` which contains two of every
69 constructor: one for when the session is composed of exactly two processes, and one for
70 when it's composed of three or more. Therefore we chose to add an unit element to parallel
71 composition. However, we didn't make that unit $p \triangleleft 0$ in order to reuse some of the lemmas
72 from [4] that use the fact that structural congruence preserves participants.

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

```

Inductive process : Type  $\triangleq$ 
| p_send : part  $\rightarrow$  label  $\rightarrow$  expr  $\rightarrow$  process  $\rightarrow$  process
| p_rect : part  $\rightarrow$  list(option process)  $\rightarrow$  process
| p_ite : expr  $\rightarrow$  process  $\rightarrow$  process  $\rightarrow$  process
| p_rec : process  $\rightarrow$  process
| p_var : nat  $\rightarrow$  process
| p_inact : process.

Inductive session : Type  $\triangleq$ 
| s_ind : part  $\rightarrow$  process  $\rightarrow$  session
| s_par : session  $\rightarrow$  session  $\rightarrow$  session
| s_zero : session.

Notation "p"  $\leftarrow\!\!\!-\>$  "P"  $\triangleq$  (s_indep P) (at level 50, no associativity).
Notation "s1"  $\parallel\!\!\!||\>$  "s2"  $\triangleq$  (s_par s1 s2) (at level 50, no associativity).

```

77

78 2.2 Structural Congruence and Operational Semantics

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

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

86 2.2.1 Semantics With Silent Transitions

⁸⁷ We have two kinds of transitions, *silent* (τ) and *observable* (β). Correspondingly, we have
⁸⁸ two kinds of *transition labels*, τ and $(p, q)\ell$ where p, q are participants and ℓ 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 .

$\boxed{[R\text{-COMM}]}$	$j \in I \quad e \downarrow v$	
$p \triangleleft \sum_{i \in I} q? \ell_i(x_i).P_i \mid q \triangleleft p! \ell_j(e).Q \mid N$	$\xrightarrow{(p,q)\ell_j} p \triangleleft P_j[v/x_j] \mid q \triangleleft Q \mid N$	
$\boxed{[R\text{-REC}]}$	$\boxed{[R\text{-COND T}]}$	
$p \triangleleft \mu X.P \mid N \xrightarrow{\tau} p \triangleleft P[\mu X.P/X] \mid N$	$\frac{e \downarrow \text{true}}{p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid N} \xrightarrow{\tau} p \triangleleft P \mid N$	
$\boxed{[R\text{-COND F}]}$	$e \downarrow \text{false}$	$\boxed{[R\text{-STRUCT}]}$
$p \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid N \xrightarrow{\tau} p \triangleleft Q \mid N$	$\frac{N'_1 \equiv N_1 \quad N_1 \xrightarrow{\lambda} N_2 \quad N_2 \equiv N'_2}{N'_1 \xrightarrow{\lambda} N'_2}$	

■ **Table 2** Operational Semantics of Sessions

23:4 Dummy short title

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

96 2.3 Reactive Semantics

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

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

■ Table 3 Unfolding of Sessions

98 $\mathcal{M} \Rightarrow \mathcal{N}$ means that \mathcal{M} can transition to \mathcal{N} through some internal actions, or τ transitions
 99 in the semantics of Section 2.2.1. We say that \mathcal{M} *unfolds* to \mathcal{N} . In Rocq it's captured by
 100 the predicate `unfoldP : session → session → Prop`.

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

■ Table 4 Reactive Semantics of Sessions

101 [R-COMM] captures communications between processes, and [R-UNFOLD] lets us consider
 102 reductions up to unfoldings. In Rocq, `betaP_lbl M lambda M'` denotes $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$. We write
 103 $\mathcal{M} \rightarrow \mathcal{M}'$ if $\mathcal{M} \xrightarrow{\lambda} \mathcal{M}'$ for some λ , which is written `betaP M M'` in Rocq. We write \rightarrow^* to
 104 denote the reflexive transitive closure of \rightarrow , which is called `betaRtc` in Rocq.

106 3 The Type System

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

110 **3.1 Local Types and Type Trees**

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

112 $S ::= \text{int} \mid \text{bool} \mid \text{nat}$

113 and the corresponding Rocq

```
Inductive sort: Type  $\triangleq$ 
| sbool: sort
| sint : sort
| snat : sort.
```

114

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

116 $\mathbb{T} ::= \text{end} \mid p\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid p\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid t \mid \mu t.\mathbb{T}$

117 Informally, in the above definition, `end` represents a role that has finished communicating.
 118 $p\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$ denotes a role that may, from any $i \in I$, receive a value of sort S_i with
 119 message label ℓ_i and continue with \mathbb{T}_i . Similarly, $p\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$ represents a role that may
 120 choose to send a value of sort S_i with message label ℓ_i and continue with \mathbb{T}_i for any $i \in I$.
 121 $\mu t.\mathbb{T}$ represents a recursive type where t is a type variable. We assume that the indexing
 122 sets I are always non-empty. We also assume that recursion is always guarded.

123 We employ an equirecursive approach based on the standard techniques from [11] where
 124 $\mu t.\mathbb{T}$ is considered to be equivalent to its unfolding $\mathbb{T}[\mu t.\mathbb{T}/t]$. This enables us to identify
 125 a recursive type with the possibly infinite local type tree obtained by fully unfolding its
 126 recursive subterms.

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

128 $\mathbb{T} ::= \text{end} \mid p\&\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I} \mid p\oplus\{\ell_i(S_i).\mathbb{T}_i\}_{i \in I}$

129 The corresponding Rocq definition is given below.

```
CoInductive ltt: Type  $\triangleq$ 
| ltt_end : ltt
| ltt_recv: part  $\rightarrow$  list (option(sort*ltt))  $\rightarrow$  ltt
| ltt_send: part  $\rightarrow$  list (option(sort*ltt))  $\rightarrow$  ltt.
```

130

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

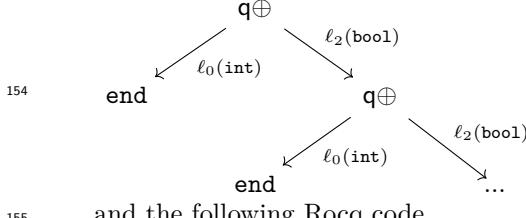
these may go

- 137 1. `SList xs`: A function that is equal to `True` if `xs` represents a continuation that has at
 138 least one element that is not `None`, and `False` otherwise.
- 139 2. `onth k xs`: A function that returns `Some x` if the element at index k (using 0-indexing) of
 140 `xs` is `Some x`, and returns `None` otherwise. Note that the function returns `None` if k is out
 141 of bounds for `xs`.
- 142 3. `Forall`, `Forall12` and `Forall12R`: `Forall` and `Forall12` are predicates from the Rocq Standard
 143 Library [14, List] that are used to quantify over elements of one list and pairwise
 144 elements of two lists, respectively. `Forall12R` is a weaker version of `Forall12` that might
 145 hold even if one parameter is shorter than the other. We frequently use `Forall12R` to
 146 express subset relations on continuations.

23:6 Dummy short title

147 ► Remark 3.4. Note that Rocq allows us to create types such as `ltt_send q []` which don't
 148 correspond to well-formed local types as the continuation is empty. In our implementation
 149 we define a predicate `wfltt : ltt → Prop` capturing that all the continuations in the local
 150 type tree are non-empty. Henceforth we assume that all local types we mention satisfy this
 151 property.

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



155 and the following Rocq code

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

156

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

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

165 3.2 Subtyping

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

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

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

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

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

knowledge" obtained during the course of the proof. Hence our subtyping relation looks like 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 (ltt_recv p xs) (ltt_recv p ys)
| sub_out : ∀ p xs ys,
  wfsend subsort R xs ys →
  subtype R (ltt_send p xs) (ltt_send p ys).

Definition subtypeC 11 12 ≡ paco2 subtype bot2 11 12.
```

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

3.3 Global Types and Type Trees

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

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

`G ::= end | p → q : {ℓi(Si).Gi}i ∈ I | t | μt.G`

We further inductively define the function `pt(G)` that denotes the participants of type `G`:

`pt(end) = pt(t) = ∅`

`pt(p → q : {ℓi(Si).Gi}i ∈ I) = {p, q} ∪ ⋃i ∈ I pt(Gi)`

`pt(μT.G) = pt(G)`

`end` denotes a protocol that has ended, `p → q : {ℓi(Si).Gi}i ∈ I` denotes a protocol where for any $i \in I$, participant `p` may send a value of sort S_i to another participant `q` via message label ℓ_i , after which the protocol continues as `Gi`.

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

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

`G ::= end | p → q : {ℓi(Si).Gi}i ∈ I`

with the corresponding Rocq code

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

We extend the function `pt` onto trees by defining `pt(G) = pt(G)` where the global type `G` corresponds to the global type tree `G`. Technical details of this definition such as well-definedness can be found in [4, 5].

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

211 3.4 Projection

212 We give definitions of projections with plain merging.

213 ▶ **Definition 3.10** (Projection). *The projection of a global type tree onto a participant r is the largest relation \upharpoonright_r between global type trees and local type trees such that, whenever $G \upharpoonright_r T$:*215 ■ $r \notin \text{pt}\{G\}$ implies $T = \text{end}$; [PROJ-END]216 ■ $G = p \rightarrow r : \{\ell_i(S_i).G_i\}_{i \in I}$ implies $T = p \& \{\ell_i(S_i).T_i\}_{i \in I}$ and $\forall i \in I, G \upharpoonright_r T_i$ [PROJ-IN]217 ■ $G = r \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$ implies $T = q \oplus \{\ell_i(S_i).T_i\}_{i \in I}$ and $\forall i \in I, G \upharpoonright_r T_i$ [PROJ-OUT]218 ■ $G = p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$ and $r \notin \{p, q\}$ implies that there are $T_i, i \in I$ such that
219 $T = \sqcap_{i \in I} T_i$ and $\forall i \in I, G \upharpoonright_r T_i$ [PROJ-CONT]220 where \sqcap is the merging operator. We also define plain merge \sqcap as

221
$$T_1 \sqcap T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

224
$$T_1 \sqcap T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = p \& \{\ell_i(S_i).T_i\}_{i \in I} & \text{and} \\ T_2 = p \& \{\ell_j(S_J).T_j\}_{j \in J} & \text{and} \\ T_3 = p \& \{\ell_k(S_k).T_k\}_{k \in I \cup J} \end{cases} \\ \text{undefined} & \text{otherwise} \end{cases}$$

225 Indeed, one of the papers we base this work on [16] uses full merging. However we used plain
226 merging in our formalisation and consequently in this work as it was already implemented in
227 [4]. Generally speaking, the results we proved can be adapted to a full merge setting, see the
228 proofs in [16].229 Informally, the projection of a global type tree G onto a participant r extracts a specification
230 for participant r from the protocol whose bird's-eye view is given by G . [PROJ-END]
231 expresses that if r is not a participant of G then r does nothing in the protocol. [PROJ-IN]
232 and [PROJ-OUT] handle the cases where r is involved in a communication in the root of G .
233 [PROJ-CONT] says that, if r is not involved in the root communication of G , then the only
234 way it knows its role in the protocol is if there is a role for it that works no matter what
235 choices p and q make in their communication. This "works no matter the choices of the other
236 participants" property is captured by the merge operations.237 In Rocq these constructions are expressed with the inductive `isMerge` and the coinductive
238 `projectionC`.

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

239

240 `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,
  (isPartsC r g → False) →
  projection R g r (litt_end)
| proj_in : ∀ p r xs ys,
  p ≠ r →
  (isPartsC r (ggt_send p r xs)) →
  List.Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨ (exists s g t, u = Some(s, g) ∧ v = Some(s, t) ∧ R g r t)) xs ys →
```

241

```

projection R (ggt_send p r xs) r (litt_recv p ys)
| proj_out : ...
| proj_cont:  $\forall$  p q r xs ys t,
  p  $\neq$  q  $\rightarrow$ 
  q  $\neq$  r  $\rightarrow$ 
  p  $\neq$  r  $\rightarrow$ 
    (isgPartsC r (ggt_send p q xs))  $\rightarrow$ 
    List.ForAll2 (fun u v  $\Rightarrow$  (u = None  $\wedge$  v = None)  $\vee$ 
      ( $\exists$  s g t, u = Some(s, g)  $\wedge$  v = Some t  $\wedge$  R g r t)) xs ys  $\rightarrow$ 
    isMerge t ys  $\rightarrow$ 
    projection R (ggt_send p q xs) r t.
Definition projectionC g r t  $\triangleq$  paco3 projection bot3 g r t.
```

242

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

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

► **Lemma 3.12.** If $\text{projection}_C G \models p \wedge T$ and $\text{projection}_C G \models p \wedge T'$ then $T = T'$.

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

```
Definition issubProj (t:ltt) (g:gtt) (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.

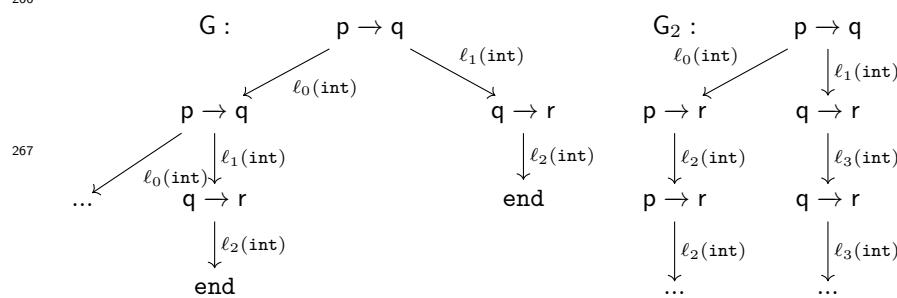
► **Definition 3.13** (Balanced Global Type Trees). A global tree G is balanced if for any subtree G' of G , there exists k such that for all $p \in \text{pt}(G')$, p occurs on every path from the root of G' of length at least k .

260 *In Rocq balancedness is expressed with the predicate balancedG (G : gtt)*

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

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

266



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

23:10 Dummy short title

270 a defective protocol as it is possible for p and q to constantly communicate through ℓ_0 and
 271 leave r waiting to receive from q a communication that will never come. We will be exploring
 272 these liveness properties from Section 4 onwards.

273 One other reason for formulating balancedness is that it allows us to use the "grafting"
 274 technique, turning proofs by coinduction on infinite trees to proofs by induction on finite
 275 global type tree contexts.

276 ▶ **Definition 3.15** (Global Type Tree Context). *Global type tree contexts are defined inductively
 277 with the following syntax:*

278
$$\mathcal{G} ::= p \rightarrow q : \{\ell_i(S_i).\mathcal{G}_i\}_{i \in I} \mid []_i$$

279 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 * gtth) → gtth.
```

280

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

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

285 ▶ **Definition 3.16** (Grafting). *Given a global type tree context \mathcal{G} whose holes are in the
 286 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
 287 tree obtained by substituting $[]_i$ with G_i in Gx .*

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

```
Inductive typ_gtth : list (option gtt) → gtth → gtt → Prop.
```

290

291 `typ_gtth gs gtx gt` means that the grafting of the set of global type trees `gs` onto the context
 292 `gtx` results in the tree `gt`.

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

295 ▶ **Lemma 3.17** (Proper Grafting Lemma, [4]). *If G is a balanced global type tree and `isgPartsC`
 296 $p G$, then there is a global type tree context $Gctx$ and an option list of global type trees gs
 297 such that $typ_gtth gs Gctx G, \sim ishParts p Gctx$ and every `Some` element of gs is of shape
 298 `gtt_end`, `gtt_send p q` or `gtt_send q p`.*

299 3.17 enables us to represent a coinductive global type tree featuring participant p as the
 300 grafting of a context that doesn't contain p with a list of trees that are all of a certain
 301 structure. If $typ_gtth gs Gctx G, \sim ishParts p Gctx$ and every `Some` element of gs is of shape
 302 `gtt_end`, `gtt_send p q` or `gtt_send q p`, then we call the pair gs and $Gctx$ as the p -grafting
 303 of G , expressed in Rocq as `typ_p_gtth gs Gctx p G`. When we don't care about the contents
 304 of gs we may just say that G is p -grafted by $Gctx$.

305 ▶ **Remark 3.18.** From now on, all the global type trees we will be referring to are assumed
 306 to be balanced. When talking about the Rocq implementation, any $G : gtt$ we mention is
 307 assumed to satisfy the predicate `wfgC G`, expressing that G corresponds to some global type
 308 and that G is balanced.

309 Furthermore, we will often require that a global type is projectable onto all its participants.
 310 This is captured by the predicate `projectableA G = $\forall p, \exists T, projectionC G p T$` . As with
 311 `wfgC`, we will be assuming that all types we mention are projectable.

312 4 Semantics of Types

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

315 4.1 Typing Contexts

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

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

317 $\Gamma ::= \emptyset \mid \Gamma, p : T$

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

321 In the Rocq implementation we implement local typing contexts as finite maps of
 322 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 lttr.
```

323

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

- 327 ■ `M.add p t g`: Adds value t with the key p to the finite map g .
- 328 ■ `M.find p g`: If the key p is in the finite map g and is associated with the value t , returns
 $\text{Some } t$, else returns `None`.
- 329 ■ `M.In p g`: A `Prop` that holds iff p is in g .
- 330 ■ `M.mem p g`: A `bool` that is equal to `true` if p is in g , and `false` otherwise.
- 331 ■ `M.Equal g1 g2`: Unfolds to $\forall p, M.find p g1 = M.find p g2$. For our purposes, if
 $M.Equal g1 g2$ then $g1$ and $g2$ are indistinguishable. This is made formal in the MMMaps
 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].
- 332 ■ `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 $f p (\text{M.find } p g1) (\text{M.find } p g2)$.
- 333 ■ `MF.Disjoint g1 g2`: A `Prop` that holds iff the keys of $g1$ and $g2$ are disjoint.
- 334 ■ `M.Eqdom g1 g2`: A `Prop` that holds iff $g1$ and $g2$ have the same domains.
- 335 One important function that we define is `disj_merge`, which merges disjoint maps and is
 used to represent the composition of typing contexts.

this section
might go

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

344

23:12 Dummy short title

```

345 Definition disj_merge (g1 g2:tctx) (H:MF.Disjoint g1 g2) : tctx ≡
346   M.merge both g1 g2.
347
348   We give LTS semantics to typing contexts, for which we first define the transition labels.
349
350   ▶ Definition 4.2 (Transition labels). A transition label  $\alpha$  has the following form:
351
352     
$$\begin{array}{ll} \alpha ::= p : q\&\ell(S) & (p \text{ receives } \ell(S) \text{ from } q) \\ \mid p : q\oplus\ell(S) & (p \text{ sends } \ell(S) \text{ to } q) \\ \mid (p,q)\ell & (\ell \text{ is transmitted from } p \text{ to } q) \end{array}$$

353

```

and in Rocq

```

353 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.
354
355   We also define the function subject( $\alpha$ ) as  $\text{subject}(p : q\&\ell(S)) = \text{subject}(p : q\oplus\ell(S)) = \{p\}$ 
356   and  $\text{subject}((p,q)\ell) = \{p,q\}$ .
357   In Rocq we represent subject( $\alpha$ ) with the predicate ispSubjl p alpha that holds iff  $p \in$ 
358   subject( $\alpha$ ).

```

```

358 Definition ispSubjl r 1 ≡
  match 1 with
  | lsend p q _ _ ⇒ p=r
  | lrecv p q _ _ ⇒ p=r
  | lcomm p q _ ⇒ p=r ∨ q=r
  end.
359
360   ▶ Remark 4.3. From now on, we assume the all the types in the local type contexts always
361   have non-empty continuations. In Rocq terms, if  $T$  is in context gamma then wfltt T holds.
362   This is expressed by the predicate wfltt: tctx → Prop.

```

4.2 Local Type Context Reductions

Next we define labelled transitions for local type contexts.

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

$$\begin{array}{c}
 \frac{k \in I}{p : q\&\{\ell_i(S_i).T_i\}_{i \in I} \xrightarrow{p:q\&\ell_k(S_k)} p : T_k} [\Gamma - \&] \\
 \frac{k \in I}{p : q\oplus\{\ell_i(S_i).T_i\}_{i \in I} \xrightarrow{p:q\oplus\ell_k(S_k)} p : T_k} [\Gamma - \oplus] \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, p : T \xrightarrow{\alpha} \Gamma', p : T} [\Gamma -,]
 \\[1em]
 \frac{\Gamma_1 \xrightarrow{p:q\oplus\ell(S)} \Gamma'_1 \quad \Gamma_2 \xrightarrow{q:p\&\ell(S')} \Gamma'_2 \quad S \leq S'}{\Gamma_1, \Gamma_2 \xrightarrow{(p,q)\ell} \Gamma'_1, \Gamma'_2} [\Gamma - \oplus\&]
 \end{array}$$

367 We write $\Gamma \xrightarrow{\alpha}$ if there exists Γ' 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, ℓ . We write $\Gamma \rightarrow$ iff $\Gamma \rightarrow \Gamma'$ for some Γ' . 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 (litt_send q xs) M.empty) (lsend p q (Some s) n) (M.add p T M.empty)
| Rrecv: ...
| Rcomm: ∀ p q g1' g2' s s' n (H1: MF.Disjoint g1 g2) (H2: MF.Disjoint g1' g2'),
  p ≠ q →
  tctxR g1 (lsend p q (Some s) n) g1' →
  tctxR g2 (lrecv q p (Some s') n) g2' →
  subsort 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') →
| Restruct: ∀ 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

```
Tp = q⊕{ℓ0(int).Tp, ℓ1(int).end}
Tq = p&{ℓ0(int).Tq, ℓ1(int).r⊕{ℓ2(int).end}}
Tr = q&{ℓ2(int).end}
```

384

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

$$\begin{array}{lll} 386 \quad \Gamma & \xrightarrow{p:q \oplus \ell_0(\text{int})} & \Gamma & (1) \\ 387 \quad \Gamma & \xrightarrow{q:p \& \ell_0(\text{int})} & \Gamma & (2) \\ 388 \quad \Gamma & \xrightarrow{(p,q)\ell_0} & \Gamma & (3) \\ 389 \quad \Gamma & \xrightarrow{r:q \& \ell_2(\text{int})} & p : T_p, q : T_q, r : \text{end} & (4) \\ 390 \quad \Gamma & \xrightarrow{p:q \oplus \ell_1(\text{int})} & p : \text{end}, q : T_q, r : T_r & (5) \\ 391 \quad \Gamma & \xrightarrow{q:p \& \ell_1(\text{int})} & p : T_p, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : T_r & (6) \\ 392 \quad \Gamma & \xrightarrow{(p,q)\ell_1} & p : \text{end}, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : T_r & (7) \end{array}$$

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 Γ' we get

23:14 Dummy short title

$$395 \quad \Gamma' \xrightarrow{q:r \oplus \ell_2(\text{int})} p : \text{end}, q : \text{end}, r : T_r \quad (8)$$

$$396 \quad \Gamma' \xrightarrow{r:q \& \ell_2(\text{int})} p : \text{end}, q : r \oplus \{\ell_3(\text{int}).\text{end}\}, r : \text{end} \quad (9)$$

$$397 \quad \Gamma' \xrightarrow{(q,r)\ell_2} p : \text{end}, q : \text{end}, r : \text{end} \quad (10)$$

398 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
399 context that can't be reduced any further.

400 In Rocq, Γ is defined the following way:

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

401

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

```
Lemma red_1 : tctxR gamma (lsend prt_p prt_q (Some sint) o) gamma.
```

403

404 4.3 Global Type Reductions

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

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

$$408 \quad \frac{k \in I}{p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I} \xrightarrow{(p,q)\ell_k} G_k} [\text{GR-}\oplus\&]$$

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

409 In Rocq $G \xrightarrow{(p,q)\ell_k} G'$ is expressed with the coinductively defined (via Paco) predicate gttstepC
410 $G \quad G' \quad p \quad q \quad k$.

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

416 4.4 Association Between Local Type Contexts and Global Types

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

- 421 ► **Definition 4.7** (Association). A local typing context Γ 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 $\Gamma(p) \leqslant G \upharpoonright p$.
 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:

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

426

427 Informally, $\Gamma \sqsubseteq G$ says that the local type trees in Γ 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 gamma' , a global type tree G'' , and a message label 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 $\text{tctxR } \text{gamma} (\text{lcomm } p \ q \ \text{ell}) \ \text{gamma}'$, then there exists a global type tree G' such that $\text{assoc } \text{gamma}' \ G'$ and $\text{gttstepC } G \ G' \ p \ q \ \text{ell}$.

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

446 $\Gamma = p : q \oplus \{\ell_0(\text{int}).\text{end}\}, q : p \& \{\ell_0(\text{int}).\text{end}, \ell_1(\text{int}).\text{end}\}$

447 and

448 $G = p \rightarrow q : \{\ell_0(\text{int}).\text{end}, \ell_1(\text{int}).\text{end}\}$

449 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
 450 soundness still requires that $\Gamma \xrightarrow{(p,q)\ell_x}$ for some x , which is satisfied in this case by the valid
 451 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].

23:16 Dummy short title

458 5.1 Safety

459 We start by defining safety:

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

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

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

465 Informally, safety says that if p and q communicate with each other and p requests to send a value using message label ℓ , then q should be able to receive that message label. Furthermore, 466 this property should be preserved under any typing context reductions. Being a coinductive 467 property, to show that $\text{safe}(\Gamma)$ it suffices to give a set φ such that $\Gamma \in \varphi$ and φ satisfies 468 $[\text{S-}\&\oplus]$ and $[\text{S-}\rightarrow]$. This amounts to showing that every element of Γ' of the set of reducts 469 of Γ , defined $\varphi := \{\Gamma' \mid \Gamma \rightarrow^* \Gamma'\}$, satisfies $[\text{S-}\&\oplus]$. We illustrate this with some examples:

471 ▶ **Example 5.2.** Let $\Gamma_A = p : \text{end}$, then Γ_A is safe: the set of reducts is $\{\Gamma_A\}$ and this set 472 respects $[\text{S-}\oplus\&]$ as its elements can't reduce, and it respects $[\text{S-}\rightarrow]$ as it's closed with 473 respect to \rightarrow .

474 Let $\Gamma_B = p : q \oplus \{\ell_0(\text{int}).\text{end}\}, q : p \& \{\ell_0(\text{nat}).\text{end}\}$. Γ_B is not safe as we have 475 $\Gamma_B \xrightarrow{p:q \oplus \ell_0}$ and $\Gamma_B \xrightarrow{q:p \& \ell_0}$ but we don't have $\Gamma_B \xrightarrow{(p,q)\ell_0}$ as $\text{int} \not\leq \text{nat}$.

476 Let $\Gamma_C = p : q \oplus \{\ell_1(\text{int}).q \oplus \{\ell_0(\text{int}).\text{end}\}\}, q : p \& \{\ell_1(\text{int}).p \& \{\ell_0(\text{nat}).\text{end}\}\}$. Γ_C is not 477 safe as we have $\Gamma_C \xrightarrow{(p,q)\ell_1} \Gamma_B$ and Γ_B is not safe.

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

479 Being a coinductive property, `safe` can be expressed in Rocq using Paco:

```
Definition weak_safety (c: tctx) ≡
  ∀ p q s s' k k', tctxRE (Isend p q (Some s) k) c → tctxRE (Irecv q p (Some s') k') c →
    tctxRE (lcomm p q k) c.

Inductive safe (R: tctx → Prop): tctx → Prop ≡
  | safety_red : ∀ c, weak_safety c → (∀ p q c' k,
    tctxR c (lcomm p q k) c' → R c')
    → safe R c.

Definition safeC c ≡ paco1 safe bot1 c.
```

481 `weak_safety` corresponds $[\text{S-}\&\oplus]$ where `tctxRE 1 c` is shorthand for $\exists c', \text{tctxR } c \ 1 \ c'$. In 482 the inductive `safe`, the constructor `safety_red` corresponds to $[\text{S-}\rightarrow]$. Then `safeC` is defined 483 as the greatest fixed point of `safe`.

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

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

486 **Proof.** $[\text{S-}\&\oplus]$ follows by inverting the projection and the subtyping, and $[\text{S-}\rightarrow]$ holds by 487 Theorem 4.10. ◀

488 5.2 Linear Time Properties

489 We now focus our attention to fairness and liveness. In this paper we have defined LTS 490 semantics on three types of constructs: sessions, local type contexts and global types. We will 491 appropriately define liveness properties on all three of these systems, so it will be convenient

492 to define a general notion of valid reduction paths (also known as *runs* or *executions* [1,
 493 2.1.1]) along with a general statement of some Linear Temporal Logic [12] constructs.

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

496 ▶ **Definition 5.4** (Reduction Paths). *A finite reduction path is an alternating sequence of
 497 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
 498 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
 499 all $0 \leq i$.*

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

503 In Rocq, we define reduction paths using possibly infinite cosequences of pairs of states
 504 (which will be `tctx`, `ggt` 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 (ggt*option label)).
Notation session_path ≡ (coseq (session*option label)).
```

505

506 Note the use of `option label`, where we employ `None` to represent transitions into the
 507 end of the list, `conil`. For example, $S_0 \xrightarrow{\lambda_0} S_1 \xrightarrow{\lambda_1} S_2$ would be represented in
 508 Rocq as `cocons (s_0, Some lambda_0) (cocons (s_1, Some lambda_1) (cocons (s_2, None)
 509 conil))), and cocons (s_1, Some lambda) conil would not be considered a valid path.`

510 Note that this definition doesn't require the transitions in the `coseq` to actually be valid.
 511 We achieve that using the coinductive predicate `valid_path_GC A:Type (V: A → label →`
`A → Prop)`, where the parameter `V` is a *transition validity predicate*, capturing if a one-step
 512 transition is valid. For all `V`, `valid_path_GC V conil` and `forall x, valid_path_GC V (cocons (x,
 513 None) conil)` hold, and `valid_path_GC V cocons (x, Some l) (cocons (y, l') xs)` holds if
 514 the transition validity predicate `V x l y` and `valid_path_GC V (cocons (y, l') xs)` hold. We
 515 use different `V` based on our application, for example in the context of local type context
 516 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).
```

518

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

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

524 ▶ **Definition 5.5** (Fair, Live Paths). *We say that a local type context path $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_1} \dots$ is
 525 fair if, for all $n \in N : \Gamma_n \xrightarrow{(p,q)\ell} \text{implies } \exists k, \ell' \text{ such that } N \ni k \geq n \text{ and } \lambda_k = (p, q)\ell'$, and
 526 therefore $\Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$. We say that a path $(\Gamma_n)_{n \in N}$ is live iff, $\forall n \in N :$
 527 1. $\forall n \in N : \Gamma_n \xrightarrow{p:q\oplus\ell(S)} \text{implies } \exists k, \ell' \text{ such that } N \ni k \geq n \text{ and } \Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$
 528 2. $\forall n \in N : \Gamma_n \xrightarrow{q:p\&\ell(S)} \text{implies } \exists k, \ell' \text{ such that } N \ni k \geq n \text{ and } \Gamma_k \xrightarrow{(p,q)\ell'} \Gamma_{k+1}$*

529 ► **Definition 5.6** (Live Local Type Context). A local type context Γ is live if whenever $\Gamma \rightarrow^* \Gamma'$,
 530 every fair path starting from Γ' is also live.

531 In general, fairness assumptions are used so that only the reduction sequences that are
 532 "well-behaved" in some sense are considered when formulating other properties [6]. For our
 533 purposes we define fairness such that, in a fair path, if at any point p attempts to send to q
 534 and q attempts to send to p then eventually a communication between p and q takes place.
 535 Then live paths are defined to be paths such that whenever p attempts to send to q or q
 536 attempts to send to p , eventually a p to q communication takes place. Informally, this means
 537 that every communication request is eventually answered. Then live typing contexts are
 538 defined to be the Γ where all fair paths that start from Γ are also live.

539 ► **Example 5.7.** Consider the contexts Γ, Γ' and Γ_{end} from Example 4.5. One possible
 540 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
 541 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
 542 possible synchronised reductions from Γ_n . Accordingly, we also have $\forall n, \Gamma_n \xrightarrow{(p,q)\ell_0} \Gamma_{n+1}$ in
 543 the path so this path is fair. However, this path is not live as we have by reduction (4) that
 544 $\Gamma_1 \xrightarrow{r:q \& \ell_2(\text{int})}$ but there is no n, ℓ' with $\Gamma_n \xrightarrow{(q,r)\ell'} \Gamma_{n+1}$ in the path. Consequently, Γ is not
 545 a live type context.

546 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
 547 $(\Gamma'_n)_{n \in \{1..4\}}$. This path is fair with respect to reductions from Γ'_1 and Γ'_2 as shown above,
 548 and it's fair with respect to reductions from Γ'_3 as reduction (10) is the only one available
 549 from Γ'_3 and we have $\Gamma'_3 \xrightarrow{(q,r)\ell_2} \Gamma'_4$ as needed. Furthermore, this path is live: the reduction
 550 $\Gamma_1 \xrightarrow{r:q \& \ell_2(\text{int})}$ that causes (Γ_n) to fail liveness is handled by the reduction $\Gamma'_3 \xrightarrow{(q,r)\ell_2} \Gamma'_4$ in
 551 this case.

552 Definition 5.5 , while intuitive, is not really convenient for a Rocq formalisation due to
 553 the existential statements contained in them. It would be ideal if these properties could
 554 be expressed as a least or greatest fixed point, which could then be formalised via Rocq's
 555 inductive or coinductive (via Paco) types. To do that, we turn to Linear Temporal Logic
 556 (LTL) [12].

557 ► **Definition 5.8** (Linear Temporal Logic). The syntax of LTL formulas ψ are defined inductively with boolean connectives \wedge, \vee, \neg , atomic propositions P, Q, \dots , and temporal operators
 558 \square (always), \diamond (eventually), \circ next and \mathcal{U} . Atomic propositions are evaluated over pairs
 559 of states and transitions (S, i, λ_i) (for the final state S_n in a finite reduction path we take
 560 that there is a null transition from S_n , corresponding to a `None` transition in Rocq) while
 561 LTL formulas are evaluated over reduction paths¹. The satisfaction relation $\rho \models \psi$ (where
 562 $\rho = S_0 \xrightarrow{\lambda_0} S_1 \dots$ is a reduction path, and ρ_i is the suffix of ρ starting from index i) is given
 563 by the following:

- 564 ■ $\rho \models P \iff (S_0, \lambda_0) \models P$.
- 565 ■ $\rho \models \psi_1 \wedge \psi_2 \iff \rho \models \psi_1 \text{ and } \rho \models \psi_2$
- 566 ■ $\rho \models \neg \psi_1 \iff \text{not } \rho \models \psi_1$
- 567 ■ $\rho \models \circ \psi_1 \iff \rho_1 \models \psi_1$
- 568 ■ $\rho \models \diamond \psi_1 \iff \exists k \geq 0, \rho_k \models \psi_1$

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

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

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 ℓ . Then fairness can be expressed in LTL with: for all p, q ,

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

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

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

581 The reason we defined the properties using LTL properties is that the operators \Diamond and \square
 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 $\Diamond P \equiv P \vee \Diamond(P)$
 585 ■ $\square P$ is the greatest solution to $\square P \equiv P \wedge \Diamond(\square P)$
 586 ■ $P \cup Q$ is the least solution to $P \cup Q \equiv Q \vee (P \wedge \Diamond(P \cup Q))$

587 Thus fairness and liveness correspond to greatest fixed points, which can be defined coinductively.

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

```
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.
```

591

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.

```
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 (lcomm p q (Some s) n)) False pt → eventually (headComm p q) pt) ∧
  (to_path_prop (tctxRE (irecv p q (Some s) n)) False pt → eventually (headComm q p) pt).
Definition live_path ≡ alwaysCG live_path_inner.
```

594

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

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

597

598

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

604 5.3 Rocq Proof of Liveness by Association

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

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

611 Our proof proceeds in the following way:

612 1. Formulate an analogue of fairness and liveness for global type reduction paths.

613 2. Prove that all global types are live for this notion of liveness.

614 3. Show that if $G : \text{gtt}$ is live and `assoc gamma G`, then `gamma` is also live.

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

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

```
619 Definition global_label_enabled 1 g ≡ match 1 with
| lsend p q (Some s) n ⇒ ∃ xs g',
  projectionC g p (ltt_send q xs) ∧ onth n xs=Some (s,g')
| lrecv p q (Some s) n ⇒ ∃ xs g',
  projectionC g p (ltt_recv q xs) ∧ onth n xs=Some (s,g')
| lcomm p q n ⇒ ∃ g', gttstepC g g' p q n
| _ ⇒ False end.
```

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

622 $\Box(\text{enabledComm}_{p,q,\ell} \implies \Diamond(\text{headComm}_{p,q}))$

623 and liveness is expressed with the following:

624 $\Box((\text{enabledSend}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{p,q})) \wedge$
 625 $(\text{enabledRecv}_{p,q,\ell,S} \implies \Diamond(\text{headComm}_{q,p})))$

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

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

635 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) ∨ (exists 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).
```

639

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)
    ∨ exists 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).
```

645

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.
```

649

650 651 `gtth_height`, `gtth_eq` and `is_tree_proper_prefix` interact in the expected way.

652 ► **Lemma 5.13.** If $\text{gtth_eq } \mathbf{g} \mathbf{g}'$ then $\text{gtth_height } \mathbf{g} = \text{gtth_height } \mathbf{g}'$.

653 ► **Lemma 5.14.** If $\text{is_tree_proper_prefix } \mathbf{g} \mathbf{g}'$ then $\text{gtth_height } \mathbf{g} < \text{gtth_height } \mathbf{g}'$.

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 (ltt_send q xs)` or `projectionC g p (ltt_recv q xs)`, `projectionC g q Tq`, \mathbf{g} is \mathbf{p} -grafted by $\mathbf{ctx_p}$ and $\mathbf{gs_p}$, and \mathbf{g} is \mathbf{q} -grafted by $\mathbf{ctx_q}$ and $\mathbf{gs_q}$. Then either `is_tree_proper_prefix ctx_q ctx_p` or `gtth_eq ctx_p ctx_q`. Furthermore, if $\text{gtth_eq } \mathbf{ctx_p} \mathbf{ctx_q}$ then `projectionC g q (ltt_send p xsq)` or `projectionC g q (ltt_recv p xsq)` for some \mathbf{xsq} .

657 658 659 660 661 **Proof.** By induction on the global type context `ctx_p`.

example

662 663 We also have that global type reductions that don't involve participant \mathbf{p} can't increase
 664 the height of the \mathbf{p} -grafting, established by the following lemma:

665 ► **Lemma 5.16.** Suppose $\mathbf{g} : \mathbf{gtt}$ is \mathbf{p} -grafted by $\mathbf{g}' : \mathbf{gtt}$ and $\mathbf{gs} : \mathbf{list}(\mathbf{option} \mathbf{gtt})$, `gttstepC`
 666 $\mathbf{g} \mathbf{g}' \mathbf{s} \mathbf{t} \mathbf{ell}$ where $\mathbf{p} \neq \mathbf{s}$ and $\mathbf{p} \neq \mathbf{t}$, and \mathbf{g}' is \mathbf{p} -grafted by \mathbf{g}' and \mathbf{gs}' . Then

23:22 Dummy short title

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

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 $\text{gttstepC } g \text{ g' p q ell}$ for some $r \neq p, q$, and
 671 tree contexts gx and gx' r -graft g and g' respectively, then $\text{gttstepH } \text{gx } p \text{ q ell } \text{gx}'$
 672 ($\text{gttstepH_consistent}$). The results then follow by induction on the relation gttstepH
 673 $\text{gx } s \text{ t ell } \text{gx}'$. ◀

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 $\text{projectionC } g \text{ p (ltt_send q xs)}$ for some xs , then a lcomm p q ell transition
 678 takes place in xs for some message label ell .
 679 (ii) If $\text{projectionC } g \text{ p (ltt_recv q xs)}$ for some xs , then a lcomm q p ell transition
 680 takes place in xs for some message label 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 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 ctx_p .

685 Let $(\text{ctx_q}, \text{gs_q})$ be the q -grafting of g . By Lemma 5.15 we have that either gtth_eq
 686 ctx_q ctx_p (a) or $\text{is_tree_proper_prefix } \text{ctx_q ctx_p}$ (b). In case (a), we have that
 687 $\text{projectionC } g \text{ q (ltt_recv p xs)}$, hence by (cite simul subproj or something here) and
 688 fairness of xs , we have that a lcomm p q 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 lcomm q r 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 ctx'_p , and g'' by 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 $\text{lcomm p q ell}'$
 696 for some ell' , therefore xs eventually has the desired transition too. ◀

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 (ρ and ρ' are local and global reduction
 707 paths, respectively), then $\lambda_0 = \lambda_1$ and ρ is path-associated with ρ' .

```
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 $\text{assoc } \gamma g$, then any local type context reduction path starting with
714 γ is associated with a global type reduction path starting with g .*

715 **Proof.** Let the local reduction path be $\gamma \xrightarrow{\lambda} \gamma_1 \xrightarrow{\lambda_1} \dots$. 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 $g \xrightarrow{\lambda} g_1$.
718 We can repeat this procedure to the remaining path starting with $\gamma_1 \xrightarrow{\lambda_1} \dots$ 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 $g \xrightarrow{\lambda} g_1 \xrightarrow{\lambda_1} \dots$ as the desired path associated with $\gamma \xrightarrow{\lambda} \gamma_1 \xrightarrow{\lambda_1} \dots$. ◀

maybe just
give the defi-
nition 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 $\text{path_assocC } xs ys$ 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 $\text{assoc } \gamma g$, then any fair local reduction path starting from γ is
733 live.*

734 **Proof.** Let xs be the fair local reduction path starting with γ . 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 $\text{assoc } \gamma g$ then γ is live.*

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

746 6.1 Typing rules

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

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

- 749 1. A local type context Γ 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 $\Gamma \vdash_M M$, or as `typ_sess M gamma` or `gamma ⊢ M` in Rocq.
- 750 2. A process variable context Θ_T associates process variables with local type trees, and an expression variable context Θ_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`.

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

757 **Table 5** Typing expressions

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

758 **Table 6** Typing processes

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

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

761 [T-SESS] says that a session made of the parallel composition of processes $\prod_i p_i \triangleleft P_i$ can
762 be typed by an associated local context Γ if the local type of participant p_i in Γ types the
763 process

764 6.2 Subject Reduction, Progress and Session Fidelity

give theorem 765 The subject reduction, progress and non-stuck theorems from [4] also hold in this setting,
no 766 with minor changes in their statements and proofs. We won't discuss these proofs in detail.

767 ▶ **Lemma 6.1.** If $\gamma \vdash_M M$ and $M \Rightarrow M'$, then $\text{typ_sess } M' \gamma$.

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

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

771 ▶ **Theorem 6.3** (Progress). If $\gamma \vdash_M M$, one of the following hold :

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

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

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

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

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 τ -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 →
  (forall p q ell e P' M', p ≠ q → unfoldP M ((p ← p_send q ell e P') \ \ \ \ M') → ∃ M'',
  betaRtc M ((p ← P') \ \ \ \ M''))
  ∧
  (forall p q l1p M', p ≠ q → unfoldP M ((p ← p_recv q l1p) \ \ \ \ M') →
  ∃ M'', P' e k, onth k l1p = Some P' ∧ betaRtc M ((p ← subst_expr_proc P' e 0 0) \ \ \ \ M'')).
```

798

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:

23:26 Dummy short title

804 1. Formulate a "fairness" property for typable sessions, with the property that any finite
 805 session reduction path can be extended to a fair session reduction path.

806 2. Lift the typing relation to reduction paths, and show that fair session reduction paths
 807 are typed by fair local type context reduction paths.

808 3. Prove that a certain transition eventually happens in the local context reduction path,
 809 and that this means the desired transition is enabled in the session reduction path.

810 We first state a "fairness" (the reason for the quotes is explained in Remark 6.9) property
 811 for session reduction paths, analogous to fairness for local type context reduction paths
 812 (Definition 5.5).

813 ► **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
 815 holds:*

$$816 \quad \square(\text{enabledComm}_{p,q,\ell} \implies \Diamond(\text{headComm}_{p,q}))$$

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

$$820 \quad \mathcal{M} = p \triangleleft \text{if}(\text{true} \oplus \text{false}) \text{ then } q! \ell_1(\text{true}) \text{ else } r! \ell_2(\text{true}).\mathbf{0} \mid q \triangleleft p? \ell_1(\mathbf{x}).\mathbf{0} \mid r \triangleleft p? \ell_2(\mathbf{x}).\mathbf{0}$$

821 We have that $\mathcal{M} \xrightarrow{(p,q)\ell_1} \mathcal{M}'$ where $\mathcal{M}' = p \triangleleft \mathbf{0} \mid q \triangleleft \mathbf{0} \mid r \triangleleft p? \ell_2(\mathbf{x}).\mathbf{0}$, and also $\mathcal{M} \xrightarrow{(p,r)\ell_2} \mathcal{M}''$
 822 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
 823 \mathcal{M} so in a fair path it should eventually be executed, however no extension of ρ can contain
 824 such a transition as \mathcal{M}' has no remaining transitions. Nevertheless, it turns out that there
 825 is a fair reduction path starting from every typable session (Lemma 6.13), and this will be
 826 enough to prove our desired liveness property.

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

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

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

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

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

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

840 We also have that typing path preserves fairness.

841 ► **Lemma 6.12.** *If session path \mathbf{xs} is typed by the local context path \mathbf{ys} , and \mathbf{xs} is fair, then
 842 so is \mathbf{ys} .*

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

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

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

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

857 We can now prove that typed sessions are live.

858 ► **Theorem 6.15** (Liveness by Typing). *For a session M_p , if $\exists \gamma \text{ gamma } \gamma \vdash_M M_p$ then*
 859 *$\text{live_sess } M_p$.*

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

865 Now let xs be a fair reduction path starting from $((p \leftarrow p_{\text{send}} q \text{ ell } e P') ||| M')$,
 866 which exists by Lemma 6.13. Let ys be the local context reduction path starting with γ
 867 that types xs , which exists by Lemma 6.11. Now ys is fair by Lemma 6.12. Therefore by
 868 Theorem 5.24 ys is live, so a $\text{lcomm } p \text{ q ell}'$ transition eventually occurs in ys for some
 869 ell' . Therefore $ys = \gamma \rightarrow^* \gamma_0 \xrightarrow{(p,q)\ell'} \gamma_1 \rightarrow \dots$ for some γ_0, γ_1 . Now
 870 consider the session M_0 typed by γ_0 in xs . We have $((p \leftarrow p_{\text{send}} q \text{ ell } e P') |||$
 871 $M'') \rightarrow^* M_0$ by M_0 being on xs . We also have that $M_0 \xrightarrow{(p,q)\ell''} M_1$ for some ℓ'' , M_1 by
 872 Theorem 6.5. Now observe that $M_0 \equiv ((p \leftarrow p_{\text{send}} q \text{ ell } e P') ||| M'')$ for some M'' as
 873 no transitions involving p have happened on the reduction path to M_0 . Therefore $\ell = \ell''$, so
 874 $M_1 \equiv ((p \leftarrow P') ||| M'')$ for some M'' , as needed. ◀

876 7 Related and Future Work

877 References

- 878 1 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and*
 879 *Mind Series)*. The MIT Press, 2008.
- 880 2 Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Paweł
 881 Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 102–115, Berlin, Heidelberg,
 882 2005. Springer Berlin Heidelberg.
- 883 3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction*
 884 *to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 885 4 Burak Ekici, Tadayoshi Kamegai, and Nobuko Yoshida. Formalising Subject Reduction and
 886 Progress for Multiparty Session Processes. In Yannick Forster and Chantal Keller, editors, *16th*
 887 *International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz*
 888 *International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:23, Dagstuhl, Germany,

- 889 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.
- 890
- 891 5 Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida.
892 Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Meth-*
893 *ods in Programming*, 104:127–173, 2019. URL: <https://www.sciencedirect.com/science/article/pii/S2352220817302237>, doi:10.1016/j.jlamp.2018.12.002.
- 894
- 895 6 Rob Van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Computing*
896 *Surveys*, 52(4):1–38, August 2019. URL: <http://dx.doi.org/10.1145/3329125>, doi:10.1145/3329125.
- 897
- 898 7 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization
899 in coinductive proof. *SIGPLAN Not.*, 48(1):193–206, January 2013. doi:10.1145/2480359.
900 2429093.
- 901
- 902 8 Pierre Letouzey and Andrew W. Appel. Modular Finite Maps over Ordered Types. URL:
903 <https://github.com/rocq-community/mmaps>.
- 904
- 905 9 Robin MILNER. Chapter 19 - operational and algebraic semantics of concurrent pro-
906 cesses. In JAN VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook
907 of Theoretical Computer Science, pages 1201–1242. Elsevier, Amsterdam, 1990. URL:
908 <https://www.sciencedirect.com/science/article/pii/B978044488074150024X>, doi:10.
909 1016/B978-0-444-88074-1.50024-X.
- 910
- 911 10 Luca Padovani. Deadlock and lock freedom in the linear pi-calculus. In *Proceedings of the*
912 *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic*
913 (*CSL*) and the *Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science*
914 (*LICS*), CSL-LICS ’14, New York, NY, USA, 2014. Association for Computing Machinery.
915 doi:10.1145/2603088.2603116.
- 916
- 917 11 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- 918
- 919 12 Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of*
920 *computer science (sfcs 1977)*, pages 46–57. ieee, 1977.
- 921
- 922 13 The Rocq Development Team. *The Rocq Reference Manual*. Inria, 2025. <https://rocq-prover.org/doc/V9.0.0/refman>.
- 923
- 924 14 The Rocq Development Team. *The Rocq Standard Library*. Inria, 2025. <https://rocq-prover.org/doc/V9.0.0/stdlib>.
- 925 15 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make
926 session types complete for lock-freedom. In *Proceedings of the 36th Annual ACM/IEEE*
927 *Symposium on Logic in Computer Science*, LICS ’21, New York, NY, USA, 2021. Association
928 for Computing Machinery. doi:10.1109/LICS52264.2021.9470531.
- 929
- 930 16 Nobuko Yoshida and Ping Hou. Less is more revisited, 2024. URL: <https://arxiv.org/abs/2402.16741>, arXiv:2402.16741.
- 931