Dummy title

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements Anonymous acknowledgements

1 Introduction

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

1.1 Processes and Sessions

▶ **Definition 1.1** (Expressions and Processes). *We define processes as follows:*

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

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

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

Processes can be composed in parallel into sessions.

▶ **Definition 1.2** (Multiparty Sessions). *Multiparty sessions are defined as follows.*

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

$\mathsf{p} \triangleleft \mathsf{P}$ denotes that participant p is running the process P, $|$ indicates parallel compositon. We write $\prod_{i\in I} \mathsf{p}_i \triangleleft \mathsf{P}_i$ to denote the session formed by p_i running P_i in parallel for all $i \in I$. $\mathcal{O}$ is an empty session with no participants, that is, the unit of parallel composition.

▶ Remark 1.3. Note that $\mathcal{O}$ is different than $\mathsf{p} \triangleleft \mathbf{0}$ as p is a participant in the latter but not the former. This differs from previous work, e.g. in [5] the unit of parallel composition is $\mathsf{p} \triangleleft \mathbf{0}$ while in [4] there is no unit. The unitless appproach of [4] results in a lot of repetition in the code, for an example see their definition of `unfoldP` which contains two of every constructor: one for when the session is composed of exactly two processes, and one for when it's composed of three or more. Therefore we chose to add an unit element to parallel composition. However, we didn't make that unit $\mathsf{p} \triangleleft \mathbf{0}$ in order to reuse some of the lemmas from [4] that use the fact that structural congruence preserves participants.

1.2 Structural Congruence and Operational Semantics

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

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

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 [13] 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 approaches see Remark 5.4.

1.2.1 Semantics With Silent Transitions

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

In Table 2, [R-COMM] describes a synchronous communication from p to q via message label ℓ_j. [R-REC] unfolds recursion, [R-CONDT] and [R-CONDF] express how to evaluate conditionals, and [R-STRUCT] shows that the reduction respects the structural pre-congruence. We write $\mathcal{M} \to \mathcal{N}$ if $\mathcal{M} \xrightarrow{\lambda} \mathcal{N}$ for some transition label λ. We write $\to^*$ to denote the reflexive transitive closure of $\to$. We also write $\mathcal{M} \Rrightarrow \mathcal{N}$ when $\mathcal{M} \equiv \mathcal{N}$ or $\mathcal{M} \to^* \mathcal{N}$ where all the transitions involved in the multistep reduction are τ transitions.

1.3 Reactive Semantics

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

2 The Type System

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

[R-COMM]

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

[R-REC]

$$\mathsf{p} \triangleleft \mu\mathbf{X}.\mathsf{P} \mid \mathcal{N} \xrightarrow{\tau} \mathsf{p} \triangleleft \mathsf{P}[\mu\mathbf{X}.\mathsf{P}/\mathbf{X}] \ \mid\ \mathcal{N}$$

[R-CONDT]

$$\frac{e \downarrow \text{true}}{\mathsf{p} \triangleleft \ \text{if } e \text{ then } \mathsf{P} \text{ else } \mathsf{Q} \ \mid\ \mathcal{N} \xrightarrow{\tau} \mathsf{p} \triangleleft \mathsf{P} \ \mid\ \mathcal{N}}$$

[R-CONDF]

$$\frac{e \downarrow \text{false}}{\mathsf{p} \triangleleft \ \text{if } e \text{ then } \mathsf{P} \text{ else } \mathsf{Q} \ \mid\ \mathcal{N} \xrightarrow{\tau} \mathsf{p} \triangleleft \mathsf{Q} \ \mid\ \mathcal{N}}$$

[R-STRUCT]

$$\frac{\mathcal{N}_1' \equiv \mathcal{N}_1 \quad \mathcal{N}_1 \xrightarrow{\lambda} \mathcal{N}_2 \quad \mathcal{N}_2 \equiv \mathcal{N}_2'}{\mathcal{N}_1' \xrightarrow{\lambda} \mathcal{N}_2'}$$

Table 2 Operational Semantics of Sessions

2.1 Local Types and Type Trees

▶ **Definition 2.1** (Sorts). *We define sorts as follows:*

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

and the corresponding Coq

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

▶ **Definition 2.2.** *Local types are defined inductively with the following syntax:*

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

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

We employ an equirecursive approach based on the standard techniques from [9] where $\mu\mathbf{t}.\mathbb{T}$ is considered to be equivalent to its unfolding $\mathbb{T}[\mu\mathbf{t}.\mathbb{T}/\mathbf{t}]$. This enables us to identify a recursive type with the possibly infinite local type tree obtained by fully unfolding its recursive subterms.

▶ **Definition 2.3.** *Local type trees are defined coinductively with the following syntax:*

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

The corresponding Coq definition is given below.

[Unf-struct]
$$\frac{\mathcal{M} \equiv \mathcal{N}}{\mathcal{M} \Rrightarrow \mathcal{N}}$$

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

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

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

[Unf-trans]
$$\frac{\mathcal{M} \Rrightarrow \mathcal{M}' \quad \mathcal{M}' \Rrightarrow \mathcal{N}}{\mathcal{M} \Rrightarrow \mathcal{N}}$$

Table 3 Unfolding of Sessions

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

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

Table 4 Reactive Semantics of Sessions

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

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

1. `SList xs`: A function that is equal to `True` if `xs` represents a continuation that has at least one element that is not `None`, and `False` otherwise.
2. `onth k xs`: A function that returns `Some x` if the element at index `k` (using 0-indexing) of `xs` is `Some x`, and returns `None` otherwise. Note that the function returns `None` if `k` is out of bounds for `xs`.
3. `Forall, Forall2` and `Forall2R` : `Forall` and `Forall2` are predicates from the Coq Standard Library [12, List] that are used to quantify over elements of one list and pairwise elements

of two lists, respectively. `Forall2R` is a weaker version of `Forall2` that might hold even if one parameter is shorter than the other. We frequently use `Forall2R` to express subset relations on continuations.

▶ Remark 2.4. Note that Coq allows us to create types such as `ltt_send q []` which don't correspond to well-formed local types as the continuation is empty. In our implementation we define a predicate `wfltt : ltt → Prop` capturing that all the continuations in the local type tree are non-empty. Henceforth we assume that all local types we mention satisfy this property.

▶ **Example 2.5.** Let local type $\mathbb{T} = \mu \mathbf{t}.\mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathsf{end}, \ell_2(\mathtt{bool}).\mathbf{t}\}$. This is equivalent to the following infinite local type tree:

$$\begin{tikzcd}[column sep=small]
 & \mathsf{q}\oplus \arrow[dl, "\ell_0(\mathtt{int})"'] \arrow[dr, "\ell_2(\mathtt{bool})"] & & \\
\mathsf{end} & & \mathsf{q}\oplus \arrow[dl, "\ell_0(\mathtt{int})"'] \arrow[dr, "\ell_2(\mathtt{bool})"] & \\
 & \mathsf{end} & & \dots
\end{tikzcd}$$

and the following Coq code

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

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

▶ Remark 2.6. We will occasionally be talking about equality (=) between coinductively defined trees in Coq. Coq's Leibniz equality is not strong enought to treat as equal the types that we will deem to be the same. To do that, we define a coinductive predicate `lttIsoC` that captures isomorphism between coinductive trees and take as an axiom that `lttIsoC T1 T2 → T1=T2`. Technical details can be found in [4].

2.2 Subtyping

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

▶ **Definition 2.7** (Subsorting and Subtyping). *Subsorting $\leq$ is the least reflexive binary relation that satisfies* $\mathtt{nat} \leq \mathtt{int}$. *Subtyping $\leqslant$ is the largest relation between local type trees coinductively defined by the following rules:*

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

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

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

In Coq we express coinductive relations such as subtyping using the Paco library [7]. The idea behind Paco is to formulate the coinductive predicate as the greatest fixpoint of an inductive relation parameterised by another relation `R` representing the "accumulated 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 l1 l2 ≜ paco2 subtype bot2 l1 l2.
```

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.

2.3 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 2.8** (Global type). *We define global types inductively as follows:*

$$\mathbb{G} ::= \quad \mathsf{end} \;\mid\; \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathbb{G}_i\}_{i\in I} \;\mid\; \boldsymbol{t} \;\mid\; \mu \boldsymbol{t}.\mathbb{G}$$

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

$$\mathtt{pt}(\mathsf{end}) = \mathtt{pt}(\boldsymbol{t}) = \emptyset$$

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

$$\mathtt{pt}(\mu \boldsymbol{T}.\mathbb{G}) = \mathtt{pt}(\mathbb{G})$$

end denotes a protocol that has ended, $\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathbb{G}_i\}_{i\in I}$ denotes a protocol where for any $i \in I$, participant p may send a value of sort S_i to another participant q via message label ℓ_i, after which the protocol continues as $\mathbb{G}_i$.

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

▶ **Definition 2.9** (Global type trees). *We define global type trees coinductively as follows:*

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

with the corresponding Coq code

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

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

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

2.4 Projection

We give definitions of projections with plain merging.

▶ **Definition 2.10** (Projection). *The projection of a global type tree onto a participant* r *is the largest relation* $\restriction_{\mathsf{r}}$ *between global type trees and local type trees such that, whenever* $\mathsf{G} \restriction_{\mathsf{r}} \mathsf{T}$*:*

- $\mathsf{r} \notin \mathtt{pt}\{\mathsf{G}\}$ *implies* $\mathsf{T} = \mathsf{end}$*;* [Proj-End]
- $\mathsf{G} = \mathsf{p} \to \mathsf{r} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ *implies* $\mathsf{T} = \mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ *and* $\forall i \in I, \mathsf{G} \restriction_{\mathsf{r}} \mathsf{T}_i$ [Proj-In]
- $\mathsf{G} = \mathsf{r} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ *implies* $\mathsf{T} = \mathsf{q}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I}$ *and* $\forall i \in I, \mathsf{G} \restriction_{\mathsf{r}} \mathsf{T}_i$ [Proj-Out]
- $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$*and* $\mathsf{r} \notin \{\mathsf{p},\mathsf{q}\}$ *implies that there are* $\mathsf{T}_i, i \in I$ *such that* $\mathsf{T} = \sqcap_{i\in I}\mathsf{T}_i$ *and* $\forall i \in I, \mathsf{G} \restriction_{\mathsf{r}} \mathsf{T}_i$ [Proj-Cont]

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

$$\mathsf{T}_1 \sqcap \mathsf{T}_2 = \begin{cases} \mathsf{T}_1 & \textit{if } \mathsf{T}_1 = \mathsf{T}_2 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

▶ Remark 2.11. In the MPST literature there exists a more powerful merge operator named full merging, defined as

$$\mathsf{T}_1 \sqcap \mathsf{T}_2 = \begin{cases} \mathsf{T}_1 & \text{if } \mathsf{T}_1 = \mathsf{T}_2 \\ \mathsf{T}_3 & \text{if } \exists I, J : \begin{cases} \mathsf{T}_1 = \mathsf{p}\&\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} & \text{and} \\ \mathsf{T}_2 = \mathsf{p}\&\{\ell_j(S_J).\mathsf{T}_j\}_{j\in J} & \textit{and} \\ \mathsf{T}_3 = \mathsf{p}\&\{\ell_k(S_k).\mathsf{T}_k\}_{k\in I\cup J} \end{cases} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Indeed, one of the papers we base this work on [14] uses full merging. However we used plain merging in our formalisation and consequently in this work as it was already implemented in [4]. Generally speaking, the results we proved can be adapted to a full merge setting, see the proofs in [14].

Informally, the projection of a global type tree G onto a participant r extracts a specification for participant r from the protocol whose bird's-eye view is given by G. [Proj-End] expresses that if r is not a participant of G then r does nothing in the protocol. [Proj-In] and [Proj-Out] handle the cases where r is involved in a communication in the root of G. [Proj-Cont] says that, if r is not involved in the root communication of G, then the only way it knows its role in the protocol is if there is a role for it that works no matter what choices p and q make in their communication. This "works no matter the choices of the other participants" property is captured by the merge operations.

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

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

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

```
Variant projection (R: gtt → part → ltt → Prop): gtt → part → ltt → Prop ≜
| proj_end : ∀ g r,
              (isgPartsC r g → False) →
              projection R g r (ltt_end)
| proj_in  : ∀ p r xs ys,
              p ≠ r →
              (isgPartsC r (gtt_send p r xs)) →
              List.Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨ (∃ s g t, u = Some(s, g) ∧ v = Some(s, t) ∧ R g r t)) xs ys →
              projection R (gtt_send p r xs) r (ltt_recv p ys)
| proj_out : ...
| proj_cont: ∀ p q r xs ys t,
              p ≠ q →
              q ≠ r →
              p ≠ r →
              (isgPartsC r (gtt_send p q xs)) →
              List.Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨
              (∃ s g t, u = Some(s, g) ∧ v = Some t ∧ R g r t)) xs ys →
              isMerge t ys →
              projection R (gtt_send p q xs) r t.
Definition projectionC g r t ≜ paco3 projection bot3 g r t.
```

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 Coq 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 2.12.** *If* `projectionC G p T` *and* `projectionC G p T'` *then* `T = T'`*.*

We write $\mathsf{G} \restriction \mathsf{r} = \mathsf{T}$ when $\mathsf{G} \restriction_{\mathsf{r}} \mathsf{T}$. Furthermore we will be frequently be making assertions about subtypes of projections of a global type e.g. $\mathsf{T} \leqslant \mathsf{G} \restriction \mathsf{r}$. In our Coq 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.
```

2.5 Balancedness, Global Tree Contexts and Grafting

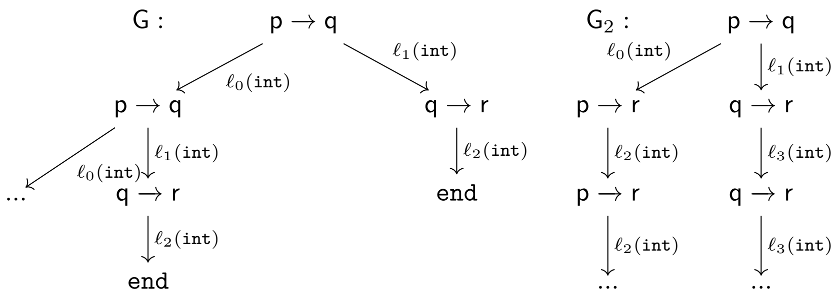
We introduce an important constraint on the types of global type trees we will consider, balancedness.

▶ **Definition 2.13** (Balanced Global Type Trees). *A global tree* G *is balanced if for any subtree* G' *of* G*, there exists* k *such that for all* $\mathsf{p} \in \mathtt{pt}(\mathsf{G}')$*,* p *occurs on every path from the root of* G' *of length at least* k*.*

In Coq balancedness is expressed with the predicate `balancedG (G : gtt)`

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

▶ **Example 2.14.** 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.



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 2.14 describes a defective protocol as it possible for p and q to constantly communicate through ℓ_0 and leave r waiting to receive from q a communication that will never come. We will be exploring these liveness properties from Section 3 onwards.

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

▶ **Definition 2.15** (Global Type Tree Context). *Global type tree contexts are defined inductively with the following syntax:*

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

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

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

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

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

▶ **Definition 2.16** (Grafting). *Given a global type tree context $\mathcal{G}$ whose holes are in the indexing set I and a set of global types $\{\mathsf{G}_i\}_{i \in I}$, the grafting $\mathcal{G}[\mathsf{G}_i]_{i \in I}$ denotes the global type tree obtained by substituting $[\ ]_i$ with G_i in Gcx.*

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

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

`typ_gtth gs gcx gt` *means that the grafting of the set of global type trees* `gs` *onto the context* `gcx` *results in the tree* `gt`.

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

▶ **Lemma 2.17** (Proper Grafting Lemma, [4]). *If* `G` *is a balanced global type tree and* `isgPartsC p G`*, then there is a global type tree context* `Gctx` *and an option list of global type trees* `gs` *such that* `typ_gtth gs Gctx G`*,* `~ ishParts p Gctx` *and every* `Some` *element of* `gs` *is of shape* `gtt_end`*,* `gtt_send p q` *or* `gtt_send q p`.

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

▶ Remark 2.18. From now on, all the global type trees we will be referring to are assumed to be balanced. When talking about the Coq implementation, any `G : gtt` we mention is assumed to satisfy the predicate `wfgC G`, expressing that `G` corresponds to some global type and that `G` is balanced.

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

3 LTS Semantics

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

3.1 Typing Contexts

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

▶ **Definition 3.1** (Typing Contexts).

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

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

In the Coq implementation we implement local typing contexts as finite maps of participants, which are represented as natural numbers, and local type trees.

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

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

- `M.add p t g`: Adds value `t` with the key `p` to the finite map `g`.
- `M.find p g`: If the key `p` is in the finite map `g` and is associated with the value `t`, returns `Some t`, else returns `None`.
- `M.In p g`: A `Prop` that holds iff `p` is in `g`.
- `M.mem p g`: A `bool` that is equal to `true` if `p` is in `g`, and `false` otherwise.
- `M.Equal g1 g2`: Unfolds to `∀ 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 MMaps library with the assertion that `M.Equal` forms a setoid, and theorems asserting that most functions on maps respect `M.Equal` by showing that they form `Proper` morphisms [11, Generalized Rewriting].

- `M.merge f g1 g2` where `f: key → option value → option value → 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 (M.find p g1) (M.find p g2)`.
- `MF.Disjoint g1 g2`: A `Prop` that holds iff the keys of `g1` and `g2` are disjoint.
- `M.Eqdom g1 g2`: A `Prop` that holds iff `g1` and `g2` have the same domains.

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

```
Definition both (z: nat) (o:option ltt) (o':option ltt) ≜
 match o,o' with
   | Some _, None   ⇒ o
   | None, Some _   ⇒ o'
   | _,_            ⇒ None
 end.

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

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

▶ **Definition 3.2** (Transition labels). *A transition label α has the following form:*

$$
\begin{array}{lll}
\alpha ::= \mathsf{p}:\mathsf{q}\&\ell(S) & & (\mathsf{p} \textit{ receives } \ell(S) \textit{ from } \mathsf{q}) \\
\quad\;\; \mid \; \mathsf{p}:\mathsf{q}\oplus\ell(S) & & (\mathsf{p} \textit{ sends } \ell(S) \textit{ to } \mathsf{q}) \\
\quad\;\; \mid \; (\mathsf{p},\mathsf{q})\ell & & (\ell \textit{ is transmitted from } \mathsf{p} \textit{ to } \mathsf{q})
\end{array}
$$

and in Coq

```
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.
```

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

In Coq we represent $\mathrm{subject}(\alpha)$ *with the predicate* `ispSubjl p alpha` *that holds iff* $\mathsf{p} \in \mathrm{subject}(\alpha)$.

```
Definition ispSubjl r l ≜
  match l with
    | lsend p q _ _ ⇒ p=r
    | lrecv p q _ _⇒ p=r
    | lcomm p q _ ⇒ p=r ∨ q=r
  end.
```

▶ Remark 3.3. From now on, we assume the all the types in the local type contexts always have non-empty continuations. In Coq terms, if `T` is in context `gamma` then `wfltt T` holds. This is expressed by the predicate `wfltt: tctx → Prop`.

3.2 Local Type Context Reductions

Next we define labelled transitions for local type contexts.

▶ **Definition 3.4** (Typing context reductions). *The typing context transition $\xrightarrow{\alpha}$ is defined inductively by the following rules:*

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

$$\frac{k \in I}{\mathsf{p} : \mathsf{q}\oplus\{\ell_i(S_i).\mathsf{T}_i\}_{i\in I} \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_k(S_k)} \mathsf{p} : \mathsf{T}_k}\ [\,\Gamma\text{ - }\oplus] \qquad \frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, \mathsf{p} : \mathsf{T} \xrightarrow{\alpha} \Gamma', \mathsf{p} : \mathsf{T}}\ [\Gamma\text{ -},]$$

$$\frac{\Gamma_1 \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell(S)} \Gamma_1' \qquad \Gamma_2 \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell(S')} \Gamma_2' \qquad S \leq S'}{\Gamma_1, \Gamma_2 \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \Gamma_1', \Gamma_2'}\ [\Gamma\text{ - }\oplus\&]$$

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

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

In Coq 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 (ltt_send q xs) M.empty) (lsend p q (Some s) n) (M.add p T M.empty)
  | Rrecv: ...
  | Rcomm: ∀ p q g1 g1' g2 g2' s s' n (H1: MF.Disjoint g1 g2) (H2: MF.Disjoint g1' g2'),
           p ≠ q →
           tctxR g1 (lsend p q (Some s) n) g1'  →
           tctxR g2 (lrecv q p (Some s') n) g2'  →
           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')
  | Rstruct: ∀ g1 g1' g2 g2' l, tctxR g1' l g2' →
    M.Equal g1 g1' →
    M.Equal g2 g2' →
    tctxR g1 l g2.
```

`Rsend`, `Rrecv` and `RvarI` are straightforward translations of $[\,\Gamma\text{ - }\&]$, $[\,\Gamma\text{ - }\oplus]$ and $[\Gamma\text{ -},]$. `Rcomm` captures $[\Gamma - \oplus\&]$ using the `disj_merge` function we defined for the compositions, and requires a proof that the contexts given are disjoint to be applied. `RStruct` captures the indistinguishability of local contexts under `M.Equal`.

We give an example to illustrate typing context reductions.

▶ **Example 3.5.** Let

$$\mathsf{T_p} = \mathsf{q}\oplus\{\ell_0(\mathtt{int}).\mathsf{T_p}\ ,\ \ell_1(\mathtt{int}).\mathsf{end}\}$$
$$\mathsf{T_q} = \mathsf{p}\&\{\ell_0(\mathtt{int}).\mathsf{T_q}\ ,\ \ell_1(\mathtt{int}).\mathsf{r}\oplus\{\ell_3(\mathtt{int}).\mathsf{end}\}\}$$
$$\mathsf{T_r} = \mathsf{q}\&\{\ell_2(\mathtt{int}).\mathsf{end}\}$$

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

$$\Gamma \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_0(\mathtt{int})} \Gamma \tag{1}$$

$$\Gamma \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell_0(\mathtt{int})} \Gamma \tag{2}$$

$$\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \tag{3}$$

$$\Gamma \xrightarrow{\mathsf{r}:\mathsf{q}\&\ell_2(\mathtt{int})} \mathsf{p}:\mathsf{T}_\mathsf{p},\ \mathsf{q}:\mathsf{T}_\mathsf{q}, \mathsf{r}:\mathsf{end} \tag{4}$$

$$\Gamma \xrightarrow{\mathsf{p}:\mathsf{q}\oplus\ell_1(\mathtt{int})} \mathsf{p}:\mathsf{end},\ \mathsf{q}:\mathsf{T}_\mathsf{q}, \mathsf{r}:\mathsf{T}_\mathsf{r} \tag{5}$$

$$\Gamma \xrightarrow{\mathsf{q}:\mathsf{p}\&\ell_1(\mathtt{int})} \mathsf{p}:\mathsf{T}_\mathsf{p},\ \mathsf{q}:\mathsf{r}\oplus\{\ell_3(\mathtt{int}).\mathsf{end}\}, \mathsf{r}:\mathsf{T}_\mathsf{r} \tag{6}$$

$$\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1} \mathsf{p}:\mathsf{end},\ \mathsf{q}:\mathsf{r}\oplus\{\ell_3(\mathtt{int}).\mathsf{end}\}, \mathsf{r}:\mathsf{T}_\mathsf{r} \tag{7}$$

and by (3) and (7) we have the synchronized reductions $\Gamma \to \Gamma$ and $\Gamma \to \Gamma' = \mathsf{p}:\mathsf{end},\ \mathsf{q}:\mathsf{r}\oplus\{\ell_2(\mathtt{int}).\mathsf{end}\}, \mathsf{r}:\mathsf{T}_\mathsf{r}$. Further reducing Γ' we get

$$\Gamma' \xrightarrow{\mathsf{q}:\mathsf{r}\oplus\ell_2(\mathtt{int})} \mathsf{p}:\mathsf{end},\ \mathsf{q}:\mathsf{end}, \mathsf{r}:\mathsf{T}_\mathsf{r} \tag{8}$$

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

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

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

In Coq, Γ is defined the following way:

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

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

```
Lemma red_1 : tctxR gamma (lsend prt_p prt_q (Some sint) 0) gamma.
```

3.3 Global Type Reductions

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

▶ **Definition 3.6** (Global type reductions). *The global type transition $\xrightarrow{\alpha}$ is defined coinductively as follows.*

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

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

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

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

3.4 Association Between Local Type Contexts and Global Types

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

▶ **Definition 3.7** (Association). *A local typing context* Γ *is associated with a global type tree* G, *written* $\Gamma \sqsubseteq \mathsf{G}$, *if the following hold:*

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

In Coq this is defined with the following:

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

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

▶ **Example 3.8.** In Example 3.5, we have that $\Gamma \sqsubseteq \mathsf{G}$ where

$$\mathsf{G} := \mathsf{p} \to \mathsf{q} : \{\ell_0(\mathtt{int}).\mathsf{G}, \ell_1(\mathtt{int}).\mathsf{q} \to \mathsf{r} : \{\ell_2(\mathtt{int}).\mathtt{end}\}\}$$

Note that G is the global type that was shown to be unbalanced in Example 2.14. In fact, we have $\Gamma(\mathsf{s}) = \mathsf{G} \upharpoonright \mathsf{s}$ for $\mathsf{s} \in \{\mathsf{p}, \mathsf{q}, \mathsf{r}\}$. Similarly, we have $\Gamma' \sqsubseteq \mathsf{G}'$ where

$$\mathsf{G}' := \mathsf{q} \to \mathsf{r} : \{\ell_2(\mathtt{int}).\mathtt{end}\}$$

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

▶ **Theorem 3.9** (Soundness of Association). *If* `assoc gamma G` *and* `gttstepC G G' p q ell`, *then there is a local type context* `gamma'`, *a global type tree* `G''` *and a message label* `ell'` *such that* `gttStepC G G'' p q ell'`, `assoc gamma' G''` *and* `tctxR gamma (lcomm p q ell') gamma'`.

Proof. ◀

▶ **Theorem 3.10** (Completeness of Association). *If* `assoc gamma G` *and* `tctxR gamma (lcomm p q ell) gamma'`, *then there exists a global type tree* `G'` *such that* `assoc gamma' G'` *and* `gttstepC G G' p q ell`.

Proof. ◀

▶ Remark 3.11. Note that in the statement of soundness we allow the message label for the local type context reduction to be different to the message label for the global type reduction. This is because our use of subtyping in association causes the entries in the local type context to be less expressive than the types obtained by projecting the global type. For example consider

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

and

$$\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_0(\mathtt{int}).\mathtt{end}, \ell_1(\mathtt{int}).\mathtt{end}\}$$

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

4 Properties of Local Type Contexts

We now use the LTS semantics to define some desirable properties on type contexts and their reduction sequences. Namely, we formulate safety, liveness and fairness properties based on the definitions in [14].

4.1 Safety

We start by defining safety:

▶ **Definition 4.1** (Safe Type Contexts). *We define* safe *coinductively as the largest set of type contexts such that whenever we have* $\Gamma \in \mathsf{safe}$*:*

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

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

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

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

▶ **Example 4.2.** Let $\Gamma_A = \mathsf{p} : \mathtt{end}$, then Γ_A is safe: the set of reducts is $\{\Gamma_A\}$ and this set respects [S- ⊕&] as its elements can't reduce, and it respects [S-→] as it's closed with respect to $\to$.

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

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

Consider Γ from Example 3.5. All the reducts satisfy [S-&⊕] , hence Γ is safe.

Being a coinductive property, safe can be expressed in Coq using Paco:

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

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

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

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

▶ **Theorem 4.3** (Safety by Association). *If* `assoc gamma g` *then* `safeC gamma`*.*

Proof. todo ◀

4.2 Linear Time Properties

We now focus our attention to fairness and liveness. In this paper we have defined LTS semantics on three types of constructs: sessions, local type contexts and global types. We will appropriately define liveness properties on all three of these systems, so it will be convenient to define a general notion of valid reduction paths (also known as *runs* or *executions* [1, 2.1.1]) along with a general statement of some Linear Temporal Logic [10] constructs.

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

▶ **Definition 4.4** (Reduction Paths). *A finite reduction path is an alternating sequence of states and labels* $S_0\lambda_0S_1\lambda_1...S_n$ *such that* $S_i \xrightarrow{\lambda_i} S_{i+1}$ *for all* $0 \leq i < n$*. An infinite reduction path is an alternating sequence of states and labels* $S_0\lambda_0S_1\lambda_1...S_n$ *such that* $S_i \xrightarrow{\lambda_i} S_{i+1}$ *for all* $0 \leq i$*.*

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

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

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

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

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

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

We can now define fairness and liveness on paths. We first restate the definition of fairness and liveness for local type context paths from [14], and use that to motivate our use of more general LTL constructs.

▶ **Definition 4.5** (Fair, Live Paths). *We say that a local type context path $\Gamma_0 \xrightarrow{\lambda_0} \Gamma_1 \xrightarrow{\lambda_2} ..$ is fair if, for all $n \in N : \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell}$ implies $\exists k, \ell'$ such that $N \ni k \geq n$ and $\lambda_k = (\mathsf{p},\mathsf{q})\ell'$, and therefore $\Gamma_k \xrightarrow{(\mathsf{p},\mathsf{q})\ell'} \Gamma_{k+1}$. We say that a path $(\Gamma_n)_{n \in N}$ is live iff, $\forall n \in N$:*

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

▶ **Definition 4.6** (Live Local Type Context). *A local type context Γ is live if whenever $\Gamma \to^* \Gamma'$, every fair path starting from Γ' is also live.*

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

▶ **Example 4.7.** Consider the contexts Γ, Γ' and Γ_{end} from Example 3.5. One possible reduction path is $\Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \ldots$. Denote this path as $(\Gamma_n)_{n\in\mathbb{N}}$, where $\Gamma_n = \Gamma$ for all $n \in \mathbb{N}$. By reductions (3) and (7), we have $\forall n, \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0}$ and $\Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1}$ as the only possible synchronised reductions from Γ_n. Accordingly, we also have $\forall n, \Gamma_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell_0} \Gamma_{n+1}$ in the path so this path is fair. However, this path is not live as we have by reduction (4) that $\Gamma_1 \xrightarrow{\mathsf{r}:\mathsf{q}\&\ell_2(\mathsf{int})}$ but there is no n, ℓ' with $\Gamma_n \xrightarrow{(\mathsf{q},\mathsf{r})\ell'} \Gamma_{n+1}$ in the path. Consequently, Γ is not a live type context.

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

Definition 4.5 , while intuitive, is not really convenient for a Coq formalisation due to the existential statements contained in them. It would be ideal if these properties could be expressed as a least or greatest fixed point, which could then be formalised via Coq's inductive or coinductive (via Paco) types. To do that, we turn to Linear Temporal Logic (LTL) [10].

▶ **Definition 4.8** (Linear Temporal Logic). *The syntax of LTL formulas ψ are defined inductively with boolean connectives $\wedge, \vee, \neg$, atomic propositions $P, Q, ..$, and temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ next and $\mathcal{U}$. Atomic propositions are evaluated over pairs of states and transitions (S, i, λ_i) (for the final state S_n in a finite reduction path we take that there is a null transition from S_n, corresponding to a* `None` *transition in Rocq) while LTL formulas are evaluated over reduction paths [1]. The satisfaction relation $\rho \models \psi$ (where $\rho = S_0 \xrightarrow{\lambda_0} S_1 ..$ is a reduction path, and ρ_i is the suffix of ρ starting from index i) is given by the following:*

- $\rho \models P \iff (S_0, \lambda_0) \models P$.
- $\rho \models \psi_1 \wedge \psi_2 \iff \rho \models \psi_1$ *and* $\rho \models \psi_2$
- $\rho \models \neg\psi_1 \iff$ *not* $\rho \models \psi_1$
- $\rho \models \bigcirc\psi_1 \iff \rho_1 \models \psi_1$
- $\rho \models \Diamond\psi_1 \iff \exists k \geq 0, \rho_k \models \psi_1$
- $\rho \models \Box\psi_1 \iff \forall k \geq 0, \rho_k \models \psi_1$
- $\rho \models \psi_1 \mathcal{U} \psi_2 \iff \exists k \geq 0, \rho_k \models \psi_2$ *and* $\forall j < k, \rho_j \models \psi_1$

Fairness and liveness for local type context paths Definition 4.5 can be defined in Linear Temporal Logic (LTL). Specifically, define atomic propositions $\mathtt{enabledComm}_{\mathsf{p},\mathsf{q},\ell}$ such that $(\Gamma, \lambda) \models \mathtt{enabledComm}_{\mathsf{p},\mathsf{q},\ell} \iff \Gamma \xrightarrow{(\mathsf{p},\mathsf{q})\ell}$, and $\mathtt{headComm}_{\mathsf{p},\mathsf{q}}$ that holds iff $\lambda = (\mathsf{p}, \mathsf{q})\ell$ for some ℓ. Then

- Fairness can be expressed in LTL with: for all p, q,

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

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

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

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

- $\Diamond P$ is the least solution to $\Diamond P \equiv P \vee \bigcirc(\Diamond P)$
- $\Box P$ is the greatest solution to $\Box P \equiv P \wedge \bigcirc(\Box P)$
- $P\mathcal{U}Q$ is the least solution to $P\mathcal{U}Q \equiv Q \vee (P \wedge \bigcirc(P\mathcal{U}Q))$

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

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

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

```
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.
```

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

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 (lsend p q (Some s) n)) False pt → eventually (headComm p q) pt) ∧
(to_path_prop (tctxRE (lrecv p q (Some s) n)) False pt → eventually (headComm q p) pt).
Definition live_path ≜ alwaysCG live_path_inner.
```

For instance, the fairness of the first reduction path for Γ given in Example 4.7 can be 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.
```

4.3 Rocq Proof of Liveness by Association

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

▶ Remark 4.9. We once again emphasise that all global types mentioned are assumed to be balanced (Definition 2.13). Indeed association with non-balanced global types doesn't guarantee liveness. As an example, consider Γ from Example 3.5, which is associated with G from Example 3.8. Yet we have shown in Example 4.7 that Γ is not a live type context. This is not surprising as Example 2.14 shows that G is not balanced.

Our proof proceeds in the following way:

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

First we define fairness and liveness for global types, analogous to Definition 4.5.

▶ **Definition 4.10** (Fairness and Liveness for Global Types). *We say that the label λ is enabled at G if the context $\{\mathsf{p}_i : \mathsf{G}\upharpoonright_{\mathsf{p}_i} \mid \mathsf{p}_i \in \mathtt{pt}\{\mathsf{G}\}\}$ can transition via λ. More explicitly, and in Rocq terms,*

```
Definition global_label_enabled l g≜ match l 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.
```

With this definition of enabling, fairness and liveness are defined exactly as in Definition 4.5. A global type reduction path is fair if the following holds:

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

and liveness is expressed with the following:

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

where `enabledSend`*,* `enabledRecv` *and* `enabledComm` *correspond to the match arms in the definition of* `global_label_enabled` *(Note that the names* `enabledSend` *and* `enabledRecv` *are chosen for consistency with Definition 4.5, there aren't actually any transitions with label* $\mathsf{p} : \mathsf{q} \oplus \ell(S)$ *in the transition system for global types). A global type* G *is live if whenever* $\mathsf{G} \to^* \mathsf{G}'$*, any fair path starting from* G' *is also live.*

Now our goal is to prove that all (well-formed, balanced, projectable) G are live under this definition. This is where the notion of grafting (Definition 2.13) becomes important, as the proof essentially proceeds by well-founded induction on the height of the tree obtained by grafting.

We first introduce some definitions on global type tree contexts (Definition 2.15).

▶ **Definition 4.11** (Global Type Context Equality, Proper Prefixes and Height)**.** *We consider two global type tree contexts to be equal if they are the same up to the relabelling the indices of their leaves. More precisely,*

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

this section is wrong, fix it

Informally, we say that the global type context $\mathbb{G}'$ *is a proper prefix of* $\mathbb{G}$ *if any path to a leaf in* $\mathbb{G}'$ *is a proper prefix of a path in* $\mathbb{G}$*. Alternatively, we can characterise it as akin to* `gtth_eq` *except where the context holes in* $\mathbb{G}'$ *are assumed to be "jokers" that can be matched with any global type context that's not just a context hole. In Rocq:*

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

give examples

We also define a function `gtth_height : gtth → Nat` *that computes the height [3] of a global type tree context.*

```
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.
```

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

▶ **Lemma 4.12.** *If* `gtth_eq gx gx'` *then* `gtth_height gx = gtth_height gx'`*.*

▶ **Lemma 4.13.** *If* `is_tree_proper_prefix gx gx'` *then* `gtth_height gx < gtth_height gx'`*.*

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

▶ **Lemma 4.14** (Multigrafting). *Let* `projectionC g p (ltt_send q xsp)` *or* `projectionC g p (ltt_recv q xsp)`*,* `projectionC g q Tq`*,* `g` *is* `p`*-grafted by* `ctx_p` *and* `gs_p`*, and* `g` *is* `q`*-grafted by* `ctx_q` *and* `gs_q`*. Then either* `is_tree_proper_prefix ctx_q ctx_p` *or* `gtth_eq ctx_p ctx_q`*. Furthermore, if* `gtth_eq ctx_p ctx_q` *then* `projectionC g q (ltt_send p xsq)` *or* `projectionC g q (ltt_recv p xsq)` *for some* `xsq`*.*

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

example

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

▶ **Lemma 4.15.** *Suppose* `g : gtt` *is* `p`*-grafted by* `gx : gtth` *and gs : list (option gtt),* `gttstepC g g' s t ell` *where* `p` $\neq$ `s` *and* `p` $\neq$ `t`*, and* `g'` *is* `p`*-grafted by* `gx'` *and* `gs'`*. Then*

(i) *If* `ishParts s gx` *or* `ishParts t gx` *, then* `gtth_height gx' < gtth_height gx`

(ii) *In general,* `gtth_height gx'` $\leq$ `gtth_height gx`

Proof. We define a inductive predicate `gttstepH : gtth` $\to$ `part` $\to$ `part` $\to$ `part` $\to$ `gtth` $\to$ `Prop` with the property that if `gttstepC g g' p q ell` for some `r` $\neq$ `p`, `q`, and tree contexts `gx` and `gx'` `r`-graft `g` and `g'` respectively, then `gttstepH gx p q ell gx'` (`gttstepH_consistent`). The results then follow by induction on the relation `gttstepH gx s t ell gx'`. ◀

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

▶ **Lemma 4.16.** *Let* `xs` *be a fair global type reduction path starting with* `g`*.*

(i) *If* `projectionC g p (ltt_send q xsp)` *for some* `xsp`*, then a* `lcomm p q ell` *transition takes place in* `xs` *for some message label* `ell`*.*

(ii) *If* `projectionC g p (ltt_recv q xsp)` *for some* `xsp`*, then a* `lcomm q p ell` *transition takes place in* `xs` *for some message label* `ell`*.*

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

Rephrasing slightly, we prove the following: forall `n : nat` and global type reduction path `xs`, if the head `g` of `xs` is `p`-grafted by `ctx_p` and `gtth_height ctx_p = n`, the lemma holds. We proceed by strong induction on `n`, that is, the tree context height of `ctx_p`.

Let `(ctx_q, gs_q)` be the `q`-grafting of `g`. By Lemma 4.14 we have that either `gtth_eq ctx_q ctx_p` (a) or `is_tree_proper_prefix ctx_q ctx_p` (b). In case (a), we have that `projectionC g q (ltt_recv p xsq)`, hence by (cite simul subproj or something here) and fairness of `xs`, we have that a `lcomm p q ell` transition eventually occurs in `xs`, as required.

In case (b), by Lemma 4.13 we have `gtth_height ctx_q < gtth_height ctx_p`, so by the induction hypothesis a transition involving `q` eventually happens in `xs`. Assume wlog that this transition has label `lcomm q r ell`, or, in the pen-and-paper notation, $(\mathsf{q},\mathsf{r})\ell$. Now consider the prefix of `xs` where the transition happens: `g` $\xrightarrow{\lambda}$ `g_1` $\to$ .. `g'` $\xrightarrow{(\mathsf{q},\mathsf{r})\ell}$ `g''`. Let `g'` be `p`-grafted by the global tree context `ctx'_p`, and `g''` by `ctx''_p`. By Lemma 4.15, `gtth_height ctx''_p < gtth_height ctx'_p` $\leq$ `gtth_height ctx_p`. Then, by the induction hypothesis, the suffix of `xs` starting with `g''` must eventually have a transition `lcomm p q ell'` for some `ell'`, therefore `xs` eventually has the desired transition too. ◀

Lemma 4.16 proves that any fair global type reduction path is also a live path, from which the liveness of global types immediately follows.

▶ **Corollary 4.17.** *All global types are live.*

We can now leverage the simulation established by Theorem 3.10 to prove the liveness (Definition 4.5) of local typing context reduction paths.

We start by lifting association (Definition 3.7) to reduction paths.

▶ **Definition 4.18** (Path Association). *Path association is defined coinductively by the following rules:*

(i) *The empty path is associated with the empty path.*

(ii) *If* $\Gamma \xrightarrow{\lambda_0} \rho$ *is path-associated with* $\mathsf{G} \xrightarrow{\lambda_1} \rho'$ *where (*ρ *and* ρ' *are local and global reduction 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.
```

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

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

▶ **Lemma 4.19.** *If* `assoc gamma g`*, then any local type context reduction path starting with* `gamma` *is associated with a global type reduction path starting with* `g`*.*

maybe just give the definition as a cofixpoint?

Proof. Let the local reduction path be `gamma` $\xrightarrow{\lambda}$ `gamma_1` $\xrightarrow{\lambda_1}$ .... We construct a path-associated global reduction path. By Theorem 3.10 there is a `g_1 : gtt` such that `g` $\xrightarrow{\lambda}$ `g_1` and `assoc gamma_1 g_1`, hence the path-associated global type reduction path starts with `g` $\xrightarrow{\lambda}$ `g_1`. We can repeat this procedure to the remaining path starting with `gamma_1` $\xrightarrow{\lambda_1}$ ... to get `g_2 : gtt` such that `assoc gamma_2 g_2` and `g_1` $\xrightarrow{\lambda_1}$ `g_2`. Repeating this, we get `g` $\xrightarrow{\lambda}$ `g_1` $\xrightarrow{\lambda_1}$ .. as the desired path associated with `gamma` $\xrightarrow{\lambda}$ `gamma_1` $\xrightarrow{\lambda_1}$ .... ◀

▶ Remark 4.20. In the Rocq implementation the construction above is implemented as a `CoFixpoint` returning a `coseq`. Theorem 3.10 is implemented as an ∃ statement that lives in `Prop`, hence we need to use the `constructive_indefinite_description` axiom to obtain the witness to be used in the construction.

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

▶ **Lemma 4.21.** *For a local reduction path* `xs` *and global reduction path* `ys`*, if* `path_assocC xs ys` *then*

(i) *If* `xs` *is fair then so is* `ys`

(ii) *If* `ys` *is live then so is* `xs`

As a corollary of Lemma 4.21, Lemma 4.19 and Lemma 4.16 we have the following:

▶ **Corollary 4.22.** *If* `assoc gamma g`*, then any fair local reduction path starting from* `gamma` *is live.*

Proof. Let `xs` be the local reduction path starting with `gamma`. By Lemma 4.19 there is a global path `ys` associated with it. By Lemma 4.21 (i) `ys` is fair, and by Lemma 4.16 `ys` is live, so by Lemma 4.21 (ii) xs is also live. ◀

Liveness of contexts follows directly from Corollary 4.22.

▶ **Theorem 4.23** (Liveness by Association). *If* `assoc gamma g` *then* `gamma` *is live.*

Proof. Suppose `gamma` $\to^*$ `gamma'`, then by Theorem 3.10 `assoc gamma' g'` for some `g'`, and hence by Corollary 4.22 any fair path starting from `gamma'` is live, as needed. ◀

5 Properties of Sessions

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

5.1 Typing rules

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

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

1. A local type context Γ associates participants with local type trees, as defined in cdef-type-ctx. Local type contexts are used to type sessions (Definition 1.2) i.e. a set of pairs of participants and single processes composed in parallel. We express such judgements as $\Gamma \vdash_{\mathcal{M}} \mathcal{M}$, or as `typ_sess M gamma` in Rocq.
2. A process variable context Θ_{T} associates process variables with local type trees, and an expression variable context Θ_{e} assigns sorts to expresion variables. Variable contexts are used to type single processes and expressions (Definition 1.1). Such judgements are expressed as $\Theta_{\mathsf{T}}, \Theta_{\mathsf{e}} \vdash_{\mathsf{P}} \mathsf{P} : \mathsf{T}$, or in as `typ_proc theta_T theta_e P T`.

$$\Theta \vdash_{\mathsf{P}} n\colon \texttt{nat} \qquad \Theta \vdash_{\mathsf{P}} i\colon \texttt{int} \qquad \Theta \vdash_{\mathsf{P}} \texttt{true}\colon \texttt{bool} \qquad \Theta \vdash_{\mathsf{P}} \texttt{false}\colon \texttt{bool} \qquad \Theta, x\colon \mathsf{S} \vdash_{\mathsf{P}} x\colon \mathsf{S}$$

$$\frac{\Theta \vdash_{\mathsf{P}} e\colon \texttt{nat}}{\Theta \vdash_{\mathsf{P}} \texttt{succ}\ e\colon \texttt{nat}} \qquad \frac{\Theta \vdash_{\mathsf{P}} e\colon \texttt{int}}{\Theta \vdash_{\mathsf{P}} \texttt{neg}\ e\colon \texttt{int}} \qquad \frac{\Theta \vdash_{\mathsf{P}} e\colon \texttt{bool}}{\Theta \vdash_{\mathsf{P}} \neg\ e\colon \texttt{bool}}$$

$$\frac{\Theta \vdash_{\mathsf{P}} e_1\colon \mathsf{S} \quad \Theta \vdash_{\mathsf{P}} e_2\colon \mathsf{S}}{\Theta \vdash_{\mathsf{P}} e_1 \oplus e_2\colon \mathsf{S}} \qquad \frac{\Theta \vdash_{\mathsf{P}} e_1\colon \texttt{int} \quad \Theta \vdash_{\mathsf{P}} e_2\colon \texttt{int}}{\Theta \vdash_{\mathsf{P}} e_1 > e_2\colon \texttt{bool}} \qquad \frac{\Theta \vdash_{\mathsf{P}} e\colon \mathsf{S} \quad \mathsf{S} \leq \mathsf{S}'}{\Theta \vdash_{\mathsf{P}} e\colon \mathsf{S}'}$$

Table 5 Typing expressions

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

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

Table 6 Typing processes

Table 5 and Table 6 state the standard typing rules for expressions and processes. We have a single rule for typing sessions:

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

5.2 Subject Reduction, Progress and Session Fidelity

give theorem no

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

▶ **Lemma 5.1.** *If* `typ_sess M gamma` *and* `unfoldP M M'` *then* `typ_sess M' gamma`.

Proof. By induction on `unfoldP M M'`. ◀

▶ **Theorem 5.2** (Subject Reduction). *If* `typ_sess M gamma` *and* `betaP_lbl M (lcomm p q ell) M'`*, then there exists a typing context* `gamma'` *such that* `tctxR gamma (lcomm p q ell) gamma'` *and* `typ_sess M' gamma'`*.*

▶ **Theorem 5.3** (Progress). *If* `typ_sess M gamma`*, one of the following hold :*

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

▶ Remark 5.4. Note that in Theorem 5.2 one transition between sessions corresponds to exactly one transition between local type contexts with the same label. That is, every session transition is observed by the corresponding type. This is the main reason for our choice of reactive semantics (**??**) as τ transitions are not observed by the type in ordinary semantics. In other words, with τ-semantics the typing relation is a *weak simulation* [**?**], while it turns into a strong simulation with reactive semantics. For our Rocq implementation working with the strong simulation turns out be more convenient.

We can also prove the following correspondence result in the reverse direction to Theorem 5.2, analogus to Theorem 3.9.

▶ **Theorem 5.5** (Session Fidelity). *If* `typ_sess M gamma` *and* `tctxR gamma (lcomm p q ell) gamma'`*, there exists a message label* `ell'` *and a session* `M'` *such that* `betaP_lbl M (lcomm p q ell') M'` *and* `typ_sess M' gamma'`*.*

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

▶ Remark 5.6. Again we note that by Theorem 5.5 a single-step context reduction induces a single-step session reduction on the type. With the τ-semantics the session reduction induced by the context reduction would be multistep.

5.3 Session Liveness

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

▶ **Definition 5.7** (Session Liveness). *Session $\mathcal{M}$ is live iff*

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

In Rocq we express this with the following:

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

Session liveness, analogous to liveness for typing contexts (Definition 4.5), says that when $\mathcal{M}$ is live, if $\mathcal{M}$ reduces to a session $\mathcal{M}'$ containing a participant that's attempting to send or receive, then $\mathcal{M}'$ reduces to a session where that communication has happened. It's also called *lock-freedom* in related work ([13, **?**]).

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

1. Formulate a "fairness" property for typable sessions, with the property that any finite session reduction path can be extended to a fair session reduction path.
2. Lift the typing relation to reduction paths, and show that fair session reduction paths are typed by fair local type context reduction paths.
3. Prove that a certain transition eventually happens in the local context reduction path, and that this means the desired transition is enabled in the session reduction path.

We first state a "fairness" (the reason for the quotes is explained in Remark 5.9) property for session reduction paths, analogous to fairness for local type context reduction paths (Definition 4.5).

▶ **Definition 5.8** ("Fairness" of Sessions). *We say that a* $(\mathsf{p},\mathsf{q})\ell$ *transition is enabled at* $\mathcal{M}$ *if* $\mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \mathcal{M}'$ *for some* $\mathcal{M}'$. *A session reduction path is fair if the following LTL property holds:*

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

▶ Remark 5.9. Definition 5.8 is not actually a sensible fairness property for our reactive semantics, mainly because it doesn't satisfy the *feasibility* [6] property stating that any finite execution can be extended to a fair execution. Consider the following session:

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

We have that $\mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1} \mathcal{M}'$ where $\mathcal{M}' = \mathsf{p} \triangleleft \mathbf{0} \mid \mathsf{q} \triangleleft \mathbf{0} \mid \mathsf{r} \triangleleft \mathsf{p}?\ell_2(\mathbf{x}).\mathbf{0}$, and also $\mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{r})\ell_2} \mathcal{M}''$ for another $\mathcal{M}''$. Now consider the reduction path $\rho = \mathcal{M} \xrightarrow{(\mathsf{p},\mathsf{q})\ell_1} \mathcal{M}'$. $(\mathsf{p},\mathsf{r})\ell_2$ is enabled at $\mathcal{M}$ so in a fair path it should eventually be executed, however no extension of ρ can contain such a transition as $\mathcal{M}'$ has no remaining transitions. Nevertheless, it turns out that there is a fair reduction path starting from every typable session can (Lemma 5.13), and this will be enough to prove our desired liveness property.

We can now lift the typing relation to reduction paths, just like we did in Definition 4.18.

▶ **Definition 5.10** (Path Typing). *[Path Typing] Path typing is a relation between session reduction paths and local type context reduction paths, defined coinductively by the following rules:*

(i) *The empty path is typed with the empty path.*

(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 reduction paths, respectively), then* $\lambda_0 = \lambda_1$ *and* ρ *is typed by* ρ'.

Similar to Lemma 4.19, we can show that if the head of the path is typable then so is the whole path.

▶ **Lemma 5.11.** *If* `typ_sess M gamma`*, then any session reduction path* `xs` *starting with* `M` *is typed by a local context reduction path* `ys` *starting with* `gamma`*.*

Proof. We can construct a local context reduction path that types the session path. The construction exactly like Lemma 4.19 but elements of the output stream are generated by Theorem 5.2 instead of Theorem 3.10. ◀

We also have that typing path preserves fairness.

▶ **Lemma 5.12.** *If session path* `xs` *is typed by the local context path* `ys`*, and* `xs` *is fair, then so is* `ys`*.*

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

▶ **Lemma 5.13** (Fair Path Existence)**.** *If* `typ_sess M gamma`*, then there is a fair session reduction path* `xs` *starting from* `M`*.*

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

▶ Remark 5.14. The Rocq implementation of Lemma 5.13 computes a `CoFixpoint` corresponding to the fair path constructed above. As in Lemma 4.19, we use `constructive_indefinite_description` to turn existence statements in `Prop` to dependent pairs. We also assume the informative law of excluded middle (`excluded_middle_informative`) in order to carry out the "check if there is a transition" step in the algorithm above. When proving that the constructed path is fair, we sometimes rely on the LTL constructs we outlined in **??** reminiscent of the techniques employed in [2].

We can now prove that typed sessions are live.

▶ **Theorem 5.15** (Liveness by Typing)**.** *For a session* `Mp`*, if* ∃ `gamma, typ_sess Mp gamma` *then* `live_sess Mp`*.*

Proof. We detail the proof for the send case of Definition 5.7, the case for the receive is similar. Suppose that `betaRtc Mp M` and `unfoldP M ( (p <-- p_send q ell e P') ||| M')`. Our goal is to show that there exists a `M''` such that `betaRtc M ((p <-- P')|||M'')`. First, observe that it suffices to show that `betaRtc ((p <-- p_send q ell e P') ||| M') M''` for some `M''`. Also note that `typ_sess M gamma` for some `gamma` by Theorem 5.2, therefore `typ_sess ((p <-- p_send q ell e P') ||| M') gamma` by **??**. Now let `xs` be the fair reduction path starting from `((p <-- p_send q ell e P') ||| M')`, which exists by Lemma 5.13. Let `ys` be the local context reduction path starting with `gamma` that types `xs`, which exists by Lemma 5.11. Now `ys` is fair by Lemma 5.12. Therefore by Theorem 4.23 `ys` is live, so a `lcomm p q ell'` transition eventually occurs in `ys` for some `ell'`. Therefore `ys` $=$ `gamma` $\to^*$ `gamma_0` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell'}$ `gamma_1` $\to$ .. for some `gamma_0`, `gamma_1`. Now consider the session `M_0` typed by `gamma_0` in `xs`. We have `betaRtc ((p <-- p_send q ell e P') ||| M'') M_0` by `M_0` being on a reduction path starting from `M`. We also have that `M_0` $\xrightarrow{(\mathsf{p},\mathsf{q})\ell''}$ `M_1` for some ℓ'', `M_1` by Theorem 5.5. Now observe that `M_0`≡ `((p <-- p_send q ell e P') ||| M'')` for some `M''` as no transitions involving `p` have happened on the reduction path to `M_0`. Therefore $\ell = \ell''$, so `M_1` ≡ `((p <-- P') ||| M'')` for some `M''`, as needed. ◀

6 Related and Future Work

References

1 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

2 Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 102–115, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

4 Burak Ekici, Tadayoshi Kamegai, and Nobuko Yoshida. Formalising subject reduction and progress for multiparty session processes. Technical report, University of Oxford, 2025.

5 Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 104:127–173, 2019. URL: https://www.sciencedirect.com/science/article/pii/S2352220817302237, doi:10.1016/j.jlamp.2018.12.002.

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

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

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

9 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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

11 The Rocq Development Team. *The Rocq Reference Manual*. Inria, 2025. https://rocq-prover.org/doc/V9.0.0/refman.

12 The Rocq Development Team. *The Rocq Standard Library*. Inria, 2025. https://rocq-prover.org/doc/V9.0.0/stdlib.

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

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