Formalising Subject Reduction and Progress for Multiparty Session Processes

Burak Ekici
Department of Computer Science, University of Oxford, UK

Tadayoshi Kamegai
Department of Computer Science, University of Oxford, UK

Nobuko Yoshida
Department of Computer Science, University of Oxford, UK

Abstract

Multiparty session types (MPST) provide a robust typing discipline for specifying and verifying communication protocols in concurrent and distributed systems involving multiple participants. This work formalises the *non-stuck theorem* for synchronous MPST in the Coq proof assistant, ensuring that well-typed communications never get stuck. We present a fully mechanised proof of the theorem, where recursive type unfoldings are modelled as infinite trees, leveraging coinductive reasoning. This marks the first formal proof to incorporate *precise subtyping*, aiming to extend the typability of processes thus precision of the type system. The proof is grounded in fundamental properties such as subject reduction and progress.

During the mechanisation process, we discovered that the structural congruence rule for recursive processes, as presented in several prior works on MPST, violates subject reduction. We resolve this issue by revising and formalising the rule to ensure the preservation of type soundness.

Our approach to formal proofs about infinite type trees involves analysing their finite prefixes through inductive reasoning within outer-level coinductively stated goals. We employ the greatest fixed point of the parameterised least fixed point technique to define coinductive predicates and use parameterised coinduction to prove properties. The formalisation comprises approximately 16K lines of Coq code, accessible at: `https://github.com/Apiros3/smpst-sr-smer`.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Theory of computation → Type theory; Theory of computation → Logic and verification; Theory of computation → Proof theory

Keywords and phrases multiparty session types, type trees, subtyping, progress, subject reduction, non-stuck theorem, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.19

Supplementary Material *Software (Source Code)*: `https://github.com/Apiros3/smpst-sr-smer`

Funding This work is partially supported by EPSRC EP/T006544/2, EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, EP/Z0005801/1, EU Horizon (TARDIS) 101093006, Advanced Research and Invention Agency (ARIA) Safeguarded AI, and a grant from the Simons Foundation.

1 Introduction

Distributed and concurrent systems rely on message-passing for communication, guided by predefined protocols. Ensuring protocol conformance is crucial to prevent failures like deadlocks and mismatched communications. *Session types*, rooted in process calculi [25, 54], provide a type-theoretic framework for specifying communication structures. Initially designed for two-party interactions [24], they were extended to *multiparty session types* (MPST) to support multi-participant protocols [17, 66]. MPST have been implemented in various languages, including Java [39, 4, 28, 29], Scala [52, 2, 64, 9], OCaml [31, 32], F⋆ [69], F♯ [46], Python [48, 12], Erlang [47, 45], MPI-C [49, 43], Go [8, 7], TypeScript [15, 44], and Rust [11, 41, 42, 63, 33]. Session types have also been formalised in proof assistants, particularly Coq [22, 19, 20, 35, 21, 34, 6, 60, 13], Idris [5, 27], and Agda [57]. For a comprehensive discussion, see [65].

MPST describes communication protocols as *global types*, outlining interactions among participants, which are then *projected* into *local types* for individual processes. A session represents an instance of a *protocol*, structuring message-passing. MPST supports various synchronisation models. In synchronous MPST [16], communication requires real-time coordination between senders and receivers, ensuring protocol compliance and message order.

This work extends MPST and synchronous communication [16] with a *mechanised proof of the non-stuck theorem* using coinductive reasoning over type trees. These trees, derived from global and local types, represent recursive structures via infinite unfoldings. The proof exploits type tree properties to refine projection accuracy under subtyping. A key novelty is integrating **subtyping** into type checking, unlike prior mechanisation efforts [34, 6, 60] that prove progress for MPST. In Coq, infinite trees are defined using positive coinductive types, differing from function-based definitions in [16, Definition A.4]. To ensure that structural equivalence (isomorphism) of infinite trees is aligned with Coq's Leibniz equality, we introduce a coinductive extensionality axiom (Axiom 22); see Remark 23 for a justification of soundness. Our type system guarantees:

1. **subject reduction**: if a typed session $\mathcal{M}$ reduces to $\mathcal{M}'$, then its typing tree G transitions via consumption steps (Definition 14) to a new tree G' that types $\mathcal{M}'$;
2. **progress**: every session $\mathcal{M}$ either terminates or reduces to another session $\mathcal{M}'$.

The *non-stuck theorem*, which states that "*well-typed sessions are free of communication errors (e.g., label mismatch, polarity mismatch, etc.) and always either normally terminate or evolve into well-typed sessions*," follows as a corollary of these properties.

Defining structural congruence as a symmetric relation, as in some prior work [2, 16, 50, 17, 18], invalidates subject reduction. To address this, we redefine congruence for processes and sessions (Table 1), disabling symmetry by removing foldback identities. This issue was identified and addressed during the formal proof process. The fix, detailed in § 3.3 (see Rem.17 and Ex.18), highlights the importance of formalisation.

Terms are categorised into processes and sessions, with types divided into *channel implicit* global ($\mathbb{G}$) and local types ($\mathbb{T}$). Traditionally, global types validate sessions, while local types validate processes. Global types define multi-party protocols, while local types specify individual roles. Both use the recursion binder μ to model repetition. The projection relation maps global types to local ones for each participant. This is the *top-down* method. Figure 1 illustrates both the *subject reduction* proof structure and our design choices. We interpret global and local types onto coinductive type trees, avoiding the μ binder by leveraging circularity of coinduction. Our equi-recursive approach treats recursive types as equivalent to their unfoldings, mapping both to the same type tree. In our setting, global type trees (G) type multiparty sessions ($\mathcal{M}$), while local type trees (T) type processes (P). We ensure that a global type always exists that unfolds into the tree used for typing a given session. The process typing $\vdash_{\mathrm{p}}$ is enhanced by the subsumption rule, allowing a supertype T' to type any process of type T. When a well-typed session $\vdash_{\mathcal{M}} \mathcal{M} \colon \mathsf{G}$ evolves into $\mathcal{M}'$, a global type tree G' is obtained by consuming actions of G, ensuring $\vdash_{\mathcal{M}} \mathcal{M}' \colon \mathsf{G}'$(*subject reduction*). All concepts in Figure 1, along with the *non-stuckness* property, are implemented in Coq [56]. The formalisation is available at: `https://github.com/Apiros3/smpst-sr-smer`.

Key Insight for Mechanisation. Proving statements that involve multiple coinduct-

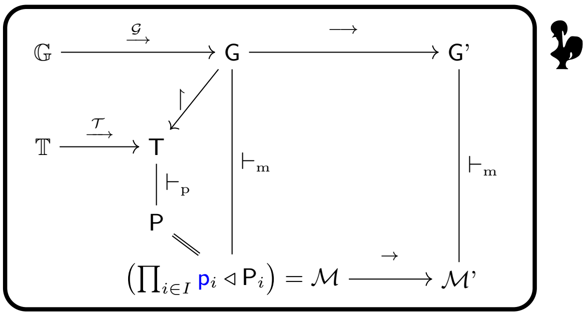


Figure 1 Design overview

ive declarations over infinite trees is challenging in Coq. We address this by (vertically) decomposing a tree into a finite prefix that excludes certain structures (e.g., participants in balanced trees—Lemma 26), then applying induction to reason about the finite portion within an outer coinductive goal.

Additionally, we use list structures to encode the finite width of a given tree, rather than function types or infinite structures such as colists. This choice simplifies proofs about trees, as it enables inductive reasoning on the width. However, it renders corecursive functions (e.g., translations from types to trees) ill-formed, since the inner finite structure prevents them from being productive. To address this, we axiomatise such functions as coinductive data types in Coq's `Prop`. While not required for the current development, we could leverage the axiom *constructive indefinite description* to inject computational content in and prove existential properties over trees. We believe these design choices are reasonable, as they scaled effectively and ultimately led to our *non-stuck* proof for MPST in Coq. We also employ the `Paco` library [30, 68], which facilitates coinductive proofs by bypassing Coq's syntactic guardedness checks.

Our mechanisation of a *core top-down MPST system* highlights key challenges, and designed for extensibility, it supports future adaptations, including *merging* [16, Definition 3.6] in *projection* and properties like *liveness* [67, Definition 12]. See § 5 for details. In the accompanying library, we employ classical reasoning to conduct case analysis primarily over coinductively defined predicates. The library comprises around 16K lines of Coq code, containing 341 proven lemmata and 117 definitions.

2 Synchronous Multiparty Session Calculus

In this section, we introduce the process calculus for sessions, employing a *semi equi-recursive* approach. This approach ensures that a recursive process and its unfolded form are represented identically, while preventing folded versions of an already unfolded process from being considered equivalent. This distinction plays a key role in establishing the proof of subject reduction theorem. Further details on this approach are covered in § 2.1.

▶ Note 1. Throughout the paper, we hyperlink Coq source code to the symbol 🐓, while `highlighted` text denotes excerpts from the Coq source.

We introduce some preliminaries. Processes interact by exchanging expressions (`expr` in Coq), denoted by e. An expression can be a value (`e_val`), such as an integer, natural number, or boolean constant, or it may be recursively formed using operators like `succ` (`e_succ`), `not` (`e_not`), $\neg$ (`e_neg`), $>$ (`e_gt`), and $+$ (`e_plus`). The language of processes is inductively defined by the following constructors 🐓.

$$\begin{array}{lcl} \mathsf{P} & ::= & \mathsf{p}!\ell(e).\mathsf{P} \quad | \quad \sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{P}_i \quad | \\ & & \text{if } \ \mathsf{e} \ \text{ then } \ \mathsf{P} \text{ else } \ \mathsf{P} \quad | \\ & & \mu\mathbf{X}.\mathsf{P} \quad | \quad \mathbf{X} \end{array}$$

```
Inductive process : Type ≜
  | p_send : part → label → expr → process → process
  | p_recv : part → list(option process) → process
  | p_ite : expr → process → process → process
  | p_rec : process → process
  | p_var : nat → process
  | p_inact : process.
```

The first constructor defines a process that sends an expression e, tagged with label ℓ, to participant p, and then proceeds as P. The second one defines a process that receives a list of messages from participant p, each tagged with labels ℓ_i. These messages are then bound to expression variables x_i within the corresponding continuations P_i. The constructor "if e then P else P′" is the conditional process representing the choice between processes P and P'. We represent inactive processes with $\mathbf{0}$ and process variables with $\mathbf{X}$. We employ de Bruijn indices to represent process variables in Coq. Processes can be recursive, thanks to the μ-binder. We assume *guarded* recursion, meaning (1) recursion always unfolds to a receive or send, and (2) all process terms are *closed*—e.g., $\mu\mathbf{X}.\mathbf{X}$ is invalid as it violates (1).

▶ **Definition 2** (option lists). *An option list of some type* `A` *is a list in which each element is either of type* `A` *or the "none" value, denoted by* $\perp$*.*

▶ Remark 3. In the accompanying Coq declaration `process`, the `p_recv` constructor uses an option list of processes. Non-existing labels are represented as `None`. Each label maps to an index in the option list. For example, if the third element in the list is `Some P`, it indicates that the label indexed by three has a valid continuation `P`; if it is `None`, no continuation is associated with that label. Using option lists eliminates the need to search for labels. We apply this approach throughout the paper when necessary. This method is sound in our setting, as no label is ever used to identify more than one continuation.

A *multiparty session* 🐓 is parallel composition "|" of participant-process pairs, denoted $\mathsf{p} \triangleleft \mathsf{P}$.

$$\mathcal{M} \quad ::= \quad \mathsf{p} \triangleleft \mathsf{P} \quad | \quad \mathcal{M} \mid \mathcal{M}$$

```
Inductive session: Type ≜
  | s_ind : part    → process → session
  | s_par : session → session → session.
```

We employ the notation `M ||| M'` to denote the parallel composition `s_par M M'` of sessions, and `p ← P` for the individual case `s_ind p P`.

2.1 Structural Pre-Congruence and Reduction Rules

The operational semantics for expressions is immaterial and therefore omitted. Instead, we present the reduction rules for sessions in Table 1 (below the dashed line). These rules rely on a *non-symmetric yet transitive preorder* relation, $\Rrightarrow$ (above the dashed line). A discussion of an issue found in previously published literature [2, 50, 17, 18], which violates *subject reduction* due to the use of *symmetric and transitive congruence*, is postponed to Remark 17 and Example 18, as it becomes more apparent under the typing rules listed in Table 2. The rule [PO-UNF] permits treating a recursive process, within a session, and its unfoldings as congruent, but not vice versa. The rule [PO-PERM] extends this idea, allowing the reordering of participant-process pairs in parallel compositions.

▶ **Notation 4.** *The notation* $\prod_{i\in I} \mathsf{p}_i \triangleleft \mathsf{P}_i$ *represents a session composed of parallel compositions* $\mathsf{p}_i \triangleleft \mathsf{P}_i$ *for all* $i \in I$*.*

The [R-COMM] rule in Table 1 governs the synchronous interaction between participants p and q such that q sends an expression payload e towards p with the label ℓ_j and continues as

$$\frac{}{\mathsf{p} \triangleleft \mu\mathbf{X}.\mathsf{P} \mid \mathcal{M} \;\Rrightarrow\; \mathsf{p} \triangleleft \mathsf{P}[\mu\mathbf{X}.\mathsf{P}/\mathbf{X}] \mid \mathcal{M}}\ [\text{PO-UNF}] \qquad \frac{J \text{ is a permutation of } I}{\prod_{i\in I} \mathsf{p}_i \triangleleft \mathsf{P}_i \;\Rrightarrow\; \prod_{j\in J} \mathsf{p}_j \triangleleft \mathsf{P}_j}\ [\text{PO-PERM}]$$

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

$$\frac{e \downarrow \mathsf{true}}{\mathsf{p} \triangleleft \text{ if } e \text{ then } \mathsf{P} \text{ else } \mathsf{Q} \;\mid\; \mathcal{M} \;\longrightarrow\; \mathsf{p} \triangleleft \mathsf{P} \;\mid\; \mathcal{M}}\ [\text{RT-ITE}] \qquad \frac{\mathcal{M}_1' \Rrightarrow \mathcal{M}_1 \quad \mathcal{M}_1 \longrightarrow \mathcal{M}_2 \quad \mathcal{M}_2 \Rrightarrow \mathcal{M}_2'}{\mathcal{M}_1' \longrightarrow \mathcal{M}_2'}\ [\text{R-STRUCT}]$$

Table 1 Session Structure Pre-Congruence (top) and Reduction Rules (bottom): we omit [RF-ITE]

the process Q. In the meantime, p awaits to receive the payload, performs the label match immediately after the reception, substitutes the value v (obtained by reducing the expression e, $e \downarrow v$) within the process P_j with the expression variable x_j, and resumes as is. If some participant p behaves as a conditional process if e then P else Q, it resumes as P in case the expression evaluates to *true*, governed by the [RT-ITE] rule, or as Q otherwise, [RF-ITE] rule. The rule [R-STRUCT] ensures that session reduction respects the pre-congruence $\Rrightarrow$ of sessions. We formalise these rules employing a `Prop` valued relation over sessions, `betaP`:

```
Inductive betaP : relation session ≜
  | ...
  | r_comm  : ∀ (p q : string) (xs : list (option process)) (y : process) (l : nat) (e : expr) (v : value) (Q : process) (M : session),
              onth l xs = Some y →   stepE e (e_val v) →
              betaP (((p ← p_recv q xs) ||| (q ← p_send p l e Q)) ||| M) (((p ← subst_expr_proc y (e_val v) 0 0) ||| (q ← Q)) ||| M)
  | r_struct: ∀ (M1 M1' M2 M2': session), unfoldP M1 M1' → unfoldP M2' M2 → betaP M1' M2' → betaP M1 M2.
```

As part of the `r_comm` constructor, the function `onth` computes the `l` th member `y` of the continuation option list of processes `xs`. The expression `e` is evaluated to the value `e_val v` by the `stepE` predicate, and the corresponding expression variable is substituted into `y` using the `subst_expr_proc` function. The `unfoldP` predicate within the `r_struct` constructor represents the pre-congruence relation $\Rrightarrow$.

3 Type System

This section covers fundamental concepts such as types, type trees, and key operations like projection, consumption, subtyping, and typing, which underpin the non-stuck theorem. In § 3.5, we introduce type tree contexts and the grafting operation, allowing traversal of finite prefixes in infinite trees—essential for reasoning about balanced infinite trees.

3.1 Types and Trees

Global types provide a high-level overview of the communication protocol, offering a comprehensive perspective on the interactions and roles of all participants involved.

▶ **Definition 5** (global types). *Global types are inductively generated by:*

$$\begin{array}{lcl} \mathsf{S} & ::= & \mathtt{nat} \mid \mathtt{int} \mid \mathtt{bool} \\ \mathbb{G} & ::= & \mathsf{end} \mid \boldsymbol{t} \mid \mu\boldsymbol{t}.\mathbb{G} \mid \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathbb{G}_i\}_{i\in I} \end{array}$$

```
Inductive global  : Type ≜
  | g_end : global
  | g_var : nat → global
  | g_send: part → part → list(option(sort*global)) → global
  | g_rec : global → global.
```

The constructor $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathbb{G}_i\}_{i\in I}$ denotes a communication from participant p to participant q with a set of messages, each identified by a label ℓ_i, payload sorts S_i, and continuations $\mathbb{G}_i$. The **end** signals the end of the protocol. Recursive types are enabled by the μ binder, and $\mathbf{t}$ represents type variables. We assume *guarded recursion*. That is, after a finite number of unfoldings, a μ-type either allows an arbitrary sequence of communication choices or reaches termination—$\mu\mathbf{t}.\mathbf{t}$ is not a valid type. Similar to the case of processes, we use de Bruijn indices to represent global type variables (also for local types; see Definition 10). We develop sorts as a variant in Coq with constructors `snat`, `sint` and `sbool`.

A tree structure can be derived from a global type, where recursive types are represented by their infinite unfoldings. Using the *equi-recursive* approach (rightmost rule in Def. 7), we represent $\mu\mathbf{t}.\mathbb{G}$ and $\mathbb{G}[\mu\mathbf{t}.\mathbb{G}/\mathbf{t}]$ with the same tree, as their intensional behaviours are identical.

▶ **Definition 6** (global type trees). *Global type trees are coinductively generated as follows.*

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

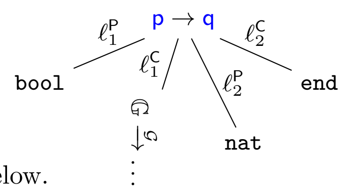
```
CoInductive gtt: Type ≜
  | gtt_end : gtt
  | gtt_send: part → part → list(option(sort*gtt)) → gtt.
```

▶ **Definition 7** (global types → global type trees). *Translating global types into global type trees is handled by the relation* $\xrightarrow{\mathcal{G}}: \mathbb{G} \to \mathsf{G} \to \mathrm{Prop}$, *with the following coinductive rules.*

$$\frac{\forall i \in I, \quad \mathbb{G}_i \xrightarrow{\mathcal{G}} \mathsf{G}_i}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathbb{G}_i\}_{i\in I} \xrightarrow{\mathcal{G}} \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}} \qquad \frac{}{\mathtt{end} \xrightarrow{\mathcal{G}} \mathtt{end}} \qquad \frac{\mathbb{G}[\mu\mathbf{t}.\mathbb{G}/\mathbf{t}] \xrightarrow{\mathcal{G}} \mathsf{G}}{\mu\mathbf{t}.\mathbb{G} \xrightarrow{\mathcal{G}} \mathsf{G}}$$

▶ **Example 8** (translation). We present a global type $\mathbb{G}$ and its corresponding type tree, where internal nodes denote communications ($\mathsf{p} \to \mathsf{q}$), and leaf nodes represent either payload types or **end**. Edges link internal nodes to a payload (ℓ^{P}) or a continuation (ℓ^{C}).

$$\mathbb{G} = \mu\mathbf{t}.\mathsf{p} \to \mathsf{q} \begin{cases} \ell_1(\mathtt{bool}).\mathbf{t} \\ \ell_2(\mathtt{nat}).\mathtt{end} \end{cases} \qquad \overset{\mathcal{G}}{\longmapsto}$$



We encode the relation $\xrightarrow{\mathcal{G}}$ in Coq as shown below.

```
Inductive gttT (R : global → gtt → Prop) : global → gtt → Prop ≜
  | ...
  | gttT_rec: ∀ G Q G', subst_global 0 0 (g_rec G) G Q → R Q G' → gttT R (g_rec G) G'.
Definition gttTC G G' ≜ paco2 gttT bot2 G G'.
```

Both `g_rec G` and its unfolding `Q` map to the tree `G'`. The `subst_global` relation handles unfolding, using `0`s for sort and global type variables as de Bruijn indices.

▶ Remark 9. Formalising translation in Coq follows the *greatest fixed point of the least fixed point* technique using the Paco library [30, 68]. We define an inductive `Prop` predicate `gttT`, acting as a generating function. It is *parametrised* by a relation `R` with the same signature, accumulating knowledge during coinductive foldings of `gttTC`. The greatest fixed point is derived using `paco2` (as long as the generating function is *monotone*— `gttT` meets this condition as it is monotone), initialised with the empty relation `bot2`. The suffix 2 indicates that the generating function has arity 2: `global` and `gtt`.

▶ **Definition 10** (local types). *Local types are inductively generated as follows.*

$$\mathbb{T} \quad ::= \quad \mathsf{end} \quad | \quad \boldsymbol{t} \quad | \quad \mu\boldsymbol{t}.\mathbb{T} \quad | \quad \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathbb{T}_i \quad | \quad \&_{i\in I} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i$$

```
Inductive local  : Type ≜
  | l_end : local
  | l_var : nat → local
  | l_rec : local → local
  | l_send : part → list(option(sort*local)) → local
  | l_recv : part → list(option(sort*local)) → local.
```

The constructor $\&_{i\in I} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i$ denotes external choice (branching) interactions with a set of messages towards participant p with labels ℓ_i, payload sorts S_i and continuations $\mathbb{T}_i$ while $\bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathbb{T}_i$ stands for internal choice (selection) and specifies a set of messages from p with labels ℓ_i, payload sorts S_i and continuations $\mathbb{T}_i$.

We derive tree structures from local types, similar to the global types (Definition 7), except that internal nodes represent branching ($\&$) or selection ($\bigoplus$).

▶ **Definition 11** (local type trees). *Local type trees are coinductively generated as follows.*

$$\mathsf{T} \quad ::= \quad \mathsf{end} \quad | \quad \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i \quad | \quad \&_{i\in I} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i$$

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

3.2 Projection and Consumption

Projection extracts local type trees for a participant from global type trees, while consumption evolves global type trees by consuming communication actions.

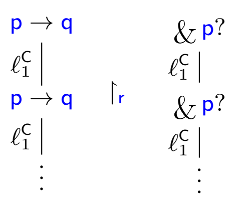
▶ **Notation 12.** *We write* `p ∈g pt(G)` *to indicate that* `p` *appears in the global type tree* `G`.

▶ **Definition 13** (projection). *Projection onto a participant* r *is the largest relation* $\upharpoonright_{\mathsf{r}}: \mathsf{G} \to \mathsf{T} \to \mathrm{Prop}$ *coinductively defined by the following rules.*

$$\dfrac{\forall i \in I, \quad \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}_i}{\mathsf{r} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I} \upharpoonright_{\mathsf{r}} \bigoplus_{i\in I} \mathsf{q}!\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{PS}] \qquad \dfrac{\forall i \in I, \quad \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}_i}{\mathsf{p} \to \mathsf{r} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I} \upharpoonright_{\mathsf{r}} \&_{i\in I} \mathsf{q}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{PR}]$$

$$\dfrac{\forall i \in I, \quad \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \quad \boxed{\forall j \in I, \mathsf{r} \in \mathtt{pt}(\mathsf{G}_j)} \quad \mathsf{G}_i \upharpoonright_{\mathsf{r}} \mathsf{T}}{\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I} \upharpoonright_{\mathsf{r}} \mathsf{T}}\ [\text{PC}] \qquad \dfrac{\mathsf{r} \notin \mathtt{pt}(\mathsf{G})}{\mathsf{G} \upharpoonright_{\mathsf{r}} \mathsf{end}}\ [\text{PE}]$$

Projection defines a participant's role within a given protocol—here with a tree representation. Clearly, participants that do not occur have no specific role in the protocol, which is what rule [PE] states. Projecting onto the sending (resp. receiving) participant at the root of a given global type tree results in a local type tree featuring an internal (resp. external) choice where the root is the receiving (resp. sending) participant and branches are local type trees obtained by coinductively applying projection to the branches of the initial global type tree as established by the rule [PS] (resp. [PR]). The rule [PC] states that if a given global type tree begins with a communication from p to q, it can be projected onto r, with $\mathsf{r} \notin \mathsf{p}, \mathsf{q}$, resulting in some local type tree T if, for all continuations, r is involved (highlighted) and their projection onto r is defined to be T—known as *plain merging*.



The highlighted condition is crucial as it prevents undesirable scenarios, such as the one depicted in the figure on the right. We develop projection in Coq as follows.

```
Variant projection (R: gtt → part → ltt → Prop): gtt → part → ltt → Prop ≜
  | ...
  | proj_cont: ∀ p q r xs ys t, p ≠ q → q ≠ r → p ≠ r → isgPartsC r (gtt_send p q xs) →
    Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨ (∃ s g l, u = Some(s, g) ∧ v = Some l ∧ R g r l)) 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.
```

For the global type tree `gtt_send p q xs`, the list `ys` contains the projections of every external choice found in the list `xs`, as ensured by the `Forall2` condition. Additionally, `isPartsC` checks whether a participant occurs in a global type tree by verifying if it is a member of the type from which the tree is extracted. This condition is the highlighted case in Definition 13. The `isMerge` predicate indicates that the projections of the entire continuation onto the participant `r` (distinct from `p` and `q`) are identical and equal to some local type tree `t`. Therefore, the projection of the global tree onto `r` results in `t`. In the rest, we use the notation `G` $\upharpoonright_{\mathtt{p}}$ `T` to represent the proposition `projectionC G p T`.

Global types trees, evolve by consuming communication actions. This allows sessions to remain well-typed even after taking several β steps. See Theorem 35.

▶ **Definition 14** (global type tree consumption). *The step (consumption) relation* $\backslash \mathsf{p} \xrightarrow{\ell} \mathsf{q} \colon \mathsf{G} \to \mathsf{G} \to \mathrm{Prop}$ *over global type trees, is defined using the following coinductive rules.*

$$\dfrac{\forall i \in I, \quad \exists k \in I, \ell = \ell_k}{(\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}) \backslash \mathsf{p} \xrightarrow{\ell} \mathsf{q}\, \mathsf{G}_k}\ [\textsc{se}] \qquad \dfrac{\forall i \in I, \quad \{\mathsf{r},\mathsf{s}\} \cap \{\mathsf{p},\mathsf{q}\} = \emptyset \quad \forall j \in I,\ \{\mathsf{p},\mathsf{q}\} \subseteq \mathtt{pt}(\mathsf{G}_j)}{(\mathsf{r} \to \mathsf{s} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}) \backslash \mathsf{p} \xrightarrow{\ell} \mathsf{q}\, (\mathsf{r} \to \mathsf{s} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i \backslash \mathsf{p} \xrightarrow{\ell} \mathsf{q}\}_{i\in I})}\ [\textsc{sn}]$$

A tree that begins with a communication from p towards q, $\mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}$, can consume the communication $\mathsf{p} \xrightarrow{\ell_k} \mathsf{q}$ according to the input label ℓ_k, provided that it represents a valid branch. Once this communication is consumed, the tree transitions into the subtree G_k, as specified by the [se] rule. The [sn] rule ensures that the communication $\mathsf{p} \xrightarrow{\ell} \mathsf{q}$ is consumed coinductively across all continuation branches of the tree $\mathsf{r} \to \mathsf{s} : \{\ell_i(\mathsf{S}_i).\mathsf{G}_i\}_{i\in I}$, as long as all participants are distinct and both p and q are explicitly present in every branch (highlighted). The relation is undefined in any other case, and developed in Coq as follows.

```
Variant gttstep (R: gtt → gtt → part → part → nat → Prop): gtt → gtt → part → part → nat → Prop ≜
  | ...
  | stneq: ∀ p q r s xs ys n, p ≠ q → r ≠ s → r ≠ p → r ≠ q → s ≠ p → s ≠ q →
     Forall (fun u ⇒ u = None ∨ (∃ s g, u = Some(s, g) ∧ isgPartsC p g ∧ isgPartsC q g)) xs →
     Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨ (∃ s g g', u = Some(s, g) ∧ v = Some(s, g') ∧ R g g' p q n)) xs ys →
     gttstep R (gtt_send r s xs) (gtt_send r s ys) p q n.
Definition gttstepC g1 g2 p q n ≜ paco5 gttstep bot5 g1 g2 p q n.
```

The condition with `Forall2` ensures that the relation is coinductively applied over the list of continuations in `xs`, producing `ys`, where each branch takes the intended step. Meanwhile, the condition with `Forall` ensures that participants `p` and `q` appear in every branch, validated by `isgPartsC`. We define the predicate `multiC` to handle the reflexive transitive closure of the `gttstepC` relation. We use the notation `G \p` $\xrightarrow{\mathtt{n}}$ `q G'` to represent the proposition `gttstepC G G' p q n`.

3.3 Subtyping

Subtyping refers to a relation between types that allows one type (the subtype) to be used in place of another type (the super-type) in any context without causing type errors. This increases the flexibility of the type system.

▶ **Definition 15** (subtyping). *The subtyping relation* $\leqslant \colon \mathsf{T} \to \mathsf{T} \to \mathrm{Prop}$ *over local type trees is coinductively defined by the following rules:*

$$\frac{}{\mathsf{end} \leqslant \mathsf{end}} \qquad \frac{\forall i \in I, \quad \mathsf{S}_i \preceq \mathsf{S}'_i \quad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i \leqslant \bigoplus_{i\in I\cup J} \mathsf{p}!\ell_i(\mathsf{S}'_i).\mathsf{T}'_i} \qquad \frac{\forall i \in I, \quad \mathsf{S}'_i \preceq \mathsf{S}_i \quad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\&_{i\in I\cup J} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i \leqslant \&_{i\in I} \mathsf{p}?\ell_i(\mathsf{S}'_i).\mathsf{T}'_i}$$

Intuitively, a subtype permits fewer internal choices and requires more external ones. The symbol $\preceq$ denotes subsorting, the least reflexive relation over payload sorts (e.g., **`nat`** $\preceq$ **`int`**).

```
Variant subtype (R: ltt → ltt → Prop): ltt → ltt → Prop ≜
  | ...
  | 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
```

The `subsort` construct encodes the subsorting $\preceq$ relation while `wfsend` ensures that types (resp. sorts) in `xs` are subtypes (resp. subsort) of those in `ys` structurally, and allows `ys` to contain trailing `sort`-`type` pairs. We use the infix symbol $\leqslant$ to denote the `subtypeC` relation and the symbol $\preceq$ for the `subsort` relation in the rest of the paper.

3.4 Typing Rules

We introduce type systems that govern processes, and sessions. Typing rules for expressions are folklore `typ_expr` thus skipped. Table 2 presents rules for processes and sessions.

▶ Remark 16. Processes and sessions are typed with local and global type trees rather than types themselves, allowing greater flexibility by abstracting away challenges of recursion. A session $\mathcal{M}$ is then *well-typed*, $\vdash \mathcal{M} \colon \mathsf{G}$, if G is the tree representation of some global type $\mathbb{G}$, namely $\mathbb{G} \xrightarrow{\mathcal{G}} \mathsf{G}$. Apart from that types do not play a critical role in the system we formalise.

$$\frac{}{\Gamma \vdash_{\mathrm{p}} \mathbf{0} \colon \mathsf{end}}\,[\text{TEND}] \qquad \frac{}{\Gamma, \mathbf{X} \colon \mathsf{T} \vdash_{\mathrm{p}} \mathbf{X} \colon \mathsf{T}}\,[\text{TVAR}] \qquad \frac{\Gamma, \mathbf{X} \colon \mathsf{T} \vdash_{\mathrm{p}} \mathsf{P} \colon \mathsf{T}}{\Gamma \vdash_{\mathrm{p}} \mu\mathbf{X}.\mathsf{P} \colon \mathsf{T}}\,[\text{TREC}] \qquad \frac{\Gamma \vdash_{\mathrm{p}} \mathsf{P} \colon \mathsf{T} \quad \mathsf{T} \leqslant \mathsf{T}'}{\Gamma \vdash_{\mathrm{p}} \mathsf{P} \colon \mathsf{T}'}\,[\text{TSUB}]$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e \colon \mathsf{bool} \quad \Gamma \vdash_{\mathrm{p}} \mathsf{P}_1 \colon \mathsf{T} \quad \Gamma \vdash_{\mathrm{p}} \mathsf{P}_2 \colon \mathsf{T}}{\Gamma \vdash_{\mathrm{p}} \text{if } e \text{ then } \mathsf{P}_1 \text{ else } \mathsf{P}_2 \colon \mathsf{T}}\,[\text{TITE}] \qquad \frac{\forall i \in I, \quad \Gamma, x_i \colon \mathsf{S}_i \vdash_{\mathrm{p}} \mathsf{P}_i \colon \mathsf{T}_i}{\Gamma \vdash_{\mathrm{p}} \sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{P}_i \colon \&_{i\in I} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\,[\text{TIN}]$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e \colon \mathsf{S} \quad \Gamma \vdash_{\mathrm{p}} \mathsf{P} \colon \mathsf{T}}{\Gamma \vdash_{\mathrm{p}} \mathsf{p}!\ell(e).\mathsf{P} \colon \bigoplus \mathsf{p}!\ell(\mathsf{S}).\mathsf{T}}\,[\text{TOUT}] \qquad \frac{\forall i \in I, \quad \mathsf{G}\!\upharpoonright_{\mathsf{p}_i} \mathsf{T}_i \quad \vdash_{\mathrm{p}} \mathsf{P}_i \colon \mathsf{T}_i \quad \mathsf{pt}(\mathsf{G}) \subseteq \{\mathsf{p}_i \mid i \in I\}}{\vdash_{\mathrm{m}} \Pi_{i\in I} \mathsf{p}_i \triangleleft \mathsf{P}_i \colon \mathsf{G}}\,[\text{TSESS}]$$

Table 2 Typing processes and sessions

▶ Remark 17. We now discuss the issue with *structural congruence*, which arises in several previous works on MPST [2, 50, 17, 18]. These studies adopt a congruence relation, $\equiv$, based on the axiom $\mu\mathbf{X}.\mathsf{P} \equiv \mathsf{P}[\mu\mathbf{X}.\mathsf{P}/\mathbf{X}]$ which lets a recursive process and its unfolding to be congruent in both directions. This violates the *subject reduction*, as the following statement does not hold:

Assume $\Gamma \vdash_{\mathrm{p}} \mathsf{P} : \mathsf{T}$ and $\mathsf{P} \equiv \mathsf{Q}$. Then we have $\Gamma \vdash_{\mathrm{p}} \mathsf{Q} : \mathsf{T}$.

▶ **Example 18** (Counterexample). Let P be $\mathsf{p}?\ell(x).\mathsf{p}!\ell'(x).\mathbf{X}$. Then we have: $\vdash_{\mathrm{p}} \mathsf{P}[\mu\mathbf{X}.\mathsf{P}/\mathbf{X}] : \mathsf{T}$, where $\mathsf{T} = \mathsf{p}?\ell(\mathtt{bool}).\mathsf{p}!\ell'(\mathtt{bool}).\mathsf{p}?\ell(\mathtt{nat}).\mathsf{p}!\ell'(\mathtt{nat}).\mathsf{T}$. However, $\nvdash_{\mathrm{p}} \mu\mathbf{X}.\mathsf{P} : \mathsf{T}$. By inverting the typing rules defined in Table 2, it can be established that if $\Gamma \vdash \mu\mathbf{X}.\mathsf{P} : \mathsf{T}''$ for some T'', then T'' must be a supertype of some T' where $\mathsf{T}' = \mathsf{p}?\ell(\mathsf{S}).\mathsf{p}!\ell'(\mathsf{S}).\mathsf{T}'$. Notably, for any sort S, T is not a supertype of T'. Therefore, types are not preserved under folding.

Our solution is to replace the structural congruence $\equiv$ with a *pre-congruence* $\Rrightarrow$ where the foldback identities are disabled by the rules in Table 1. This is solution minimal in formalisation and already imported by some recently published work [61, 3].

Formalising process typing rules `typ_proc` 🐓, we maintain two contexts: `ctxS` for expression-sort pairs and `ctxT` for process-type pairs.

```
Inductive typ_proc: ctxS → ctxT → process → ltt → Prop ≜
  | tc_sub: ∀ cs ct p t t', typ_proc cs ct p t → t ⩽ t' → wfC t' → typ_proc cs ct p t'
  | tc_rec: ∀ cs ct p t, typ_proc cs (Some t :: ct) p t → typ_proc cs ct (p_rec p) t ...
```

The predicate `wfC` 🐓 within the `tc_sub` constructor ensures that the local type tree `t'` is extracted from a local type `lt` such that `lt` is guarded, and its continuations are neither all `None` nor empty—well-foundedness property. We employ the notation `Gs Gt ⊢ P: T` and `Gs ⊢ e: S` to denote the propositions `typ_proc Gs Gt P T` and `typ_expr Gs e S`. The typing rule for sessions `typ_sess` 🐓 is implemented as follows.

```
Inductive typ_sess : session → gtt → Prop ≜
| tsess: ∀ M G, wfgC G → (∀ pt, isgPartsC pt G → InT pt M) → NoDup (flattenT M) →
             ForallT (fun p P ⇒ ∃ T, G ↾p T ∧ nil nil ⊢ P: T) M → typ_sess M G.
```

The predicate `ForallT` applies a property over participants and processes to every parallel composition within a session. The function `flattenT` extracts all participants from a session in a list, while the `inT` function checks if a specific participant is present in the session. A session `M` is well typed by a global type tree `G` if for every composition `p◁ P` in `M`, the type `G` is projectable onto `p` to yield a local type tree `T`, and the process `P` conforms to `T`. The session `M` must not contain any duplicate participants (`NoDup (flattenT M)`). If a participant appears in the global type tree `G`, it must also be present in the session `M`.

▶ Note 19. The weakening `wfgC G` 🐓 in `tsess` guarantees the existence of a global type, from which the tree `G`—typing session `M`—is derived using the translation in Definition 7. The purpose of using inductive syntax alongside coinductive semantics is to lift syntactic identity among types to a semantic notion of equivalence through translation employing equi-recursion, thereby simplifying property proofs. A similar outcome could, of course, be achieved by defining types directly using coinductive syntax.

We prove translation "*well-behaved*" by showing that *a global type and its unfolding translate to the same tree* 🐓. To illustrate a translation, we verify Example 8 🐓. Also, in the rest, parameters in the theorem statements are universally quantified unless otherwise stated.

Lemma 20 inverts process typing rules for two cases. See `inversion.v` 🐓 for all cases.

▶ **Lemma 20.** 🐓 *Given* `Gs Gt ⊢ P: T`*,*

(a) If `P` *is of the form* `p_recv p xs`*, then ∃ option list* `ys` *of sort-local type tree pairs such that* `(ltt_recv p ys) ⩽ T` *and for all processes* `Q` *in* `xs` *and sort-local type tree pairs* `(s, t)` *in* `ys`*, we can reason that* `(Some s :: Gs) Gt ⊢ Q: t`*.*

(b) If `P` *is of the form* `p_send p l e Q`*, then ∃ sort* `S` *and local type tree* `T'` *such that* `Gs ⊢ e: S`*,* `Gs Gt ⊢ Q: T'`*, and* `(ltt_send p (+[l] (Some (S,T')))) ⩽ T`*.*

The function `+[n]` (called `extendLis` 🐓 in the code) takes an instance `a: A` and returns an option list of type `A`, where the first `n` elements are `None`, and the `n`th element is `a`.

3.5 Grafting, Balancedness and Well formedness

We introduce *global type tree contexts* Γ_G 🐓, representing finite prefixes of a global type tree G by truncating the infinite continuation at specific nodes, leaving holes at those points.

$$\Gamma_{\mathsf{G}} \quad ::= \quad \mathsf{p} \to \mathsf{q} : \{\ell_i(\mathsf{S}_i).\Gamma_{\mathsf{G}_i}\}_{i \in I} \quad | \quad [\ ]_i$$

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

▶ **Definition 21** (Grafting). *The* grafting *operation constructs a global type tree* G *by* filling all holes *in an input context* Γ_{G} *with non-*$\bot$ *elements of a specified option list of global type trees* $[\mathsf{G}_0, \ldots, \mathsf{G}_m]$*, denoted* $\Gamma_{\mathsf{G}}[\mathsf{G}_0, \ldots, \mathsf{G}_m] = \mathsf{G}$*. See Figure 2 for an example.*

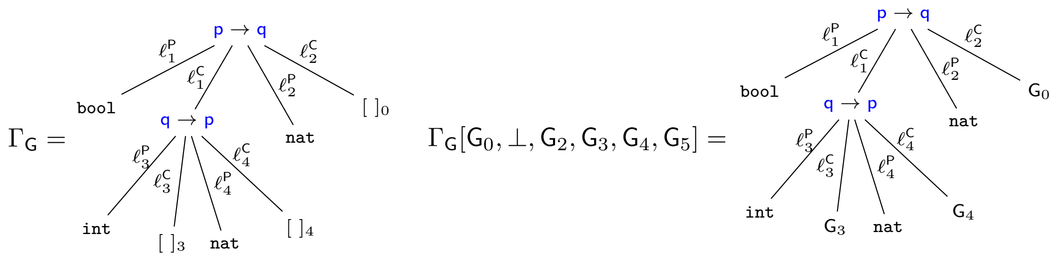


Figure 2 Grafting Example

The *grafting* approach is used to inductively track finite prefixes of global type trees through contexts, offering a way to gain insights into infinite trees. The procedure for associating holes with global type trees for grafting purposes relies on how the holes are identified. In the `gtth` declaration, we make use of naturals to identify the holes. We then accordingly clarify a method for this association in the Coq declaration `typ_gtth` of grafting.

```
Inductive typ_gtth : list (option gtt) → gtth → gtt → Prop ≜
  | gt_hol : ∀ n l gc, onth n l = Some gc → typ_gtth l (gtth_hol n) gc
  | gt_send: ∀ l p q xs ys, SList xs →
      Forall2 (fun u v ⇒ (u = None ∧ v = None) ∨ (∃ s g g', u = Some(s, g) ∧ v = Some(s, g') ∧ typ_gtth l g g')) xs ys →
      typ_gtth l (gtth_send p q xs) (gtt_send p q ys).
```

The `gt_hol` constructor indicates which element from the option list `l` is used to fill each hole: the `n` th element of `l` fills `gtth_hol n`, provided it is not `None`. In the `gt_send` constructor, the condition `SList xs` ensures that the list `xs` contains `Some` continuation context, rather than being entirely composed of `None` values. Furthermore, the condition making use of `Forall2` guarantees that all holes (`gtth_hol`) in the continuation list `xs` are filled with `gtt` s from the list `l` , resulting in a list of global type tree continuations `ys` .

The `gtth` declaration allows a single natural number to reference multiple holes within a type tree context. In this case, holes are grafted with the same `gtt` . This design poses no issues as `gtth` is used only for grafting within `typ_gtth` . If the list of `gtt` s lacks enough information to fill even one hole, the grafting operation is undefined. Unused elements in the list play no crucial role either. Theorems in the paper consider only those used in grafting.

The grafting aids proofs with infinite trees. One such example is *the partiality of the projection*: if projecting a *well-formed* (Definition 24) tree G onto a participant p results in trees T_1 and T_2, then $\mathsf{T}_1 = \mathsf{T}_2$, where "=" is Coq's Leibniz equality. We omit the proof here but emphasise that to establish this in Coq, we use the *coinductive extensionality* principle (Axiom 22) to treat an isomorphism between local type trees "∼" as Leibniz equality.

▶ **Axiom 22** (coinductive extensionality). $\forall \mathsf{T}_1$ *and* T_2*, we assume* $\mathsf{T}_1 \sim \mathsf{T}_2 \implies \mathsf{T}_1 = \mathsf{T}_2$.

▶ Remark 23. In Coq, local type trees can be characterised by the type `lttmapA`, representing partial functions that map paths—lists of natural numbers `list nat` and Booleans `bool`—to nodes `node` [16, Definition A.4]. These nodes include actions like send `lnode_send`, receive `lnode_recv`, end `lnode_end`, and payload sorts `lnode_s`, with the Boolean flag indicating whether to consider payload sorts or continuations in the tree.

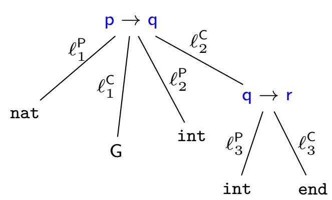
```
Inductive lnode : Type ≜
 | lnode_end : lnode
 | lnode_send: part ⟶ lnode
 | lnode_recv: part ⟶ lnode
 | lnode_s   : sort ⟶ lnode
Inductive lttmapA: list nat ⟶ bool ⟶ lnode ⟶ Prop ≜
  | lend  : lttmapA nil false lnode_end
  | lcons : ∀ p w gn l L, lttmapA w false (lnode_send p) ⟶ In l L ⟶ lttmapA (w ++ [l]) false gn
  | lcsend: ∀ p w gk l, lttmapA (w ++ [l]) false (lnode_send p) ⟶ lttmapA w false gk
  | lcsort: ∀ w s gk l, lttmapA (w ++ [l]) true (lnode_s s) ⟶ lttmapA w false gk ...
```

We justify that Axiom 22 does not introduce unsoundness in Coq by leveraging isomorphisms between coinductive and function types [1]. Specifically, `ltt` with the coinductive extensionality is isomorphic to `lttmapA` with functional extensionality. Thus, characterising local type trees using (1) partial functions with functional extensionality and (2) positive coinductive types with coinductive extensionality are equivalent. Thus, Axiom 22 is sound.

▶ **Definition 24** (Balancedness). G *is balanced, if* $\forall$ *subtree* G' *of* G, *whenever* p *is in participants of* G, $\mathsf{p} \in_{\mathsf{g}} \mathtt{pt}(\mathsf{G}')$, *then* $\exists k \in \mathbb{N}$ *such that*

1. $\forall$ *paths* γ, *of length* k, *from the root of* G', p *is involved in a node along* γ
2. $\forall$ *paths* γ *leading to an* `end`, *from the root of* G', p *is involved in a node along* γ.

G =



Balancedness is best exemplified via its negation. Figure on the left depicts an example of an unbalanced tree G. Observe that the path with labels ℓ_1^{C} has no r. **Well-formedness :** Global type tree G is *well-formed* (`wfgC`) if $\exists$ global type $\mathbb{G}$, where recursion is guarded and all continuations are both non-empty and non-$\bot$, such that $\mathbb{G} \xrightarrow{\mathcal{G}} \mathsf{G}$ and G is balanced.

▶ Note 25. In all of the following statements, global type trees are assumed to be *well-formed.* Additionally, we write `p ∈h pt(Gl)` when `p` appears in the global type tree context `Gl`.

Also, balancedness is a regularity condition that ensures *liveness*, meaning that all sends and receives in the protocol prescribed by a given type tree are eventually executed. For unbalanced trees, the grafting technique described above cannot be applied; specifically, Lemma 26 cannot be established.

▶ **Lemma 26.** *If* `p ∈g pt(G)`, *then* $\exists$ *an option list* `L` *of global types and a context* `Gl` *such that* `typ_gtth L Gl G` *with* `p ∉h pt(Gl)`. *Each element filling a hole in* `Gl` *from* `L` *is of* `gtt_send p q lsg`, `gtt_send q p lsg` *or* `gtt_end` *shape, for some participant* `q` *and option lists* `lsg` *of sort-global type tree pairs.*

The statement asserts that a global type tree can be formed by grafting a tree context, excluding a specific participant, by a list of global type trees with particular structure.

Proof follows by induction on the length k of the paths (`gttmap`) within balanced global type trees.

4 Proof of Non-stuck Theorem in Coq

This section presents a Coq formalisation of the *non-stuck theorem* for synchronous multiparty session types, proven through *subject reduction* and *progress*. Figure 3 illustrates the interrelations among the lemma/theorem statements discussed in § 3 and § 4.

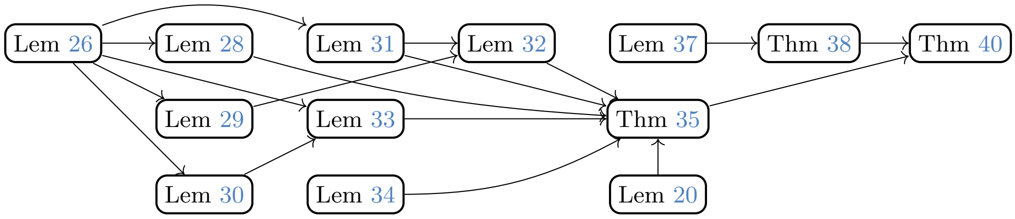


Figure 3 Dependency Graph

▶ **Notation 27.** *We write* $\mathtt{l}_\mathtt{i}$ *to refer to* `onth i l`*, where* `i` *is some index and* `l` *is a list.*

▶ **Lemma 28.** *If we have* $\mathtt{G} \upharpoonright_\mathtt{p} \mathtt{(ltt\_send\ q\ l_1)}$, $\mathtt{G} \upharpoonright_\mathtt{q} \mathtt{(ltt\_recv\ p\ l_2)}$, $\mathtt{(snd\ l_1)_n = T}$, $\mathtt{(snd\ l_2)_n = T'}$ *and* $\mathtt{G \backslash p \xrightarrow{n} q\ G'}$ *then* $\mathtt{G'} \upharpoonright_\mathtt{p} \mathtt{T}$ *and* $\mathtt{G'} \upharpoonright_\mathtt{q} \mathtt{T'}$ *hold.*

This statement preserves projections of global type trees under the consumption relation. Given a well-formed tree `G` with projections onto `p` and `q`, where `p` sends to `q` with continuations $\mathtt{l_1}$, `q` receives from `p` with continuations $\mathtt{l_2}$, and $\mathtt{n}^{\text{th}}$ elements of these lists are `T` and `T'`. If the communication step "`p` to `q`" in `G` is consumed with the $\mathtt{n}^{\text{th}}$ continuation, the resulting projections onto `p` and `q` yield `T` and `T'`.

▶ **Lemma 29.** *Given* $\mathtt{G} \upharpoonright_\mathtt{p} \mathtt{(ltt\_send\ q\ l_1)}$, $\mathtt{G} \upharpoonright_\mathtt{q} \mathtt{(ltt\_recv\ p\ l_2)}$, $\mathtt{(snd\ l_1)_n = T}$, $\mathtt{(snd\ l_2)_n = T'}$, $\mathtt{G \backslash p \xrightarrow{n} q\ G'}$ *and* $\mathtt{G} \upharpoonright_\mathtt{r} \mathtt{T''}$, $\exists$ `L` *such that* $\mathtt{G'} \upharpoonright_\mathtt{r} \mathtt{L}$ *and* `L = T''`.

The statement is a variation of Lemma 28 in that the final projection is not restricted to the participants involved in the consumed communication step.

▶ **Lemma 30.** *Given* $\mathtt{G} \upharpoonright_\mathtt{p} \mathtt{(ltt\_send\ q\ l_1)}$, $\mathtt{G} \upharpoonright_\mathtt{q} \mathtt{(ltt\_recv\ p\ l_2)}$ *and* $\mathtt{(l_1)_n = (s,\ T)}$, $\exists$ *a sort* `s'` *and a local type tree* `T'` *such that* $\mathtt{(l_2)_n = (s',\ T')}$.

This property ensures the "well-definedness condition" of projections: continuations do not result in `None`. Specifically, for a well-formed tree `G` with projections onto `p` and `q`, where `p` sends to `q` with continuations $\mathtt{l_1}$ and `q` receives from `p` with continuations $\mathtt{l_2}$, if the $\mathtt{n}^{\text{th}}$ continuation in $\mathtt{l_1}$ is well-defined, then the $\mathtt{n}^{\text{th}}$ continuation in $\mathtt{l_2}$ is also well-defined.

▶ **Lemma 31.** *Given* $\mathtt{G} \upharpoonright_\mathtt{p} \mathtt{(ltt\_send\ q\ l_1)}$, $\mathtt{G} \upharpoonright_\mathtt{q} \mathtt{(ltt\_recv\ p\ l_2)}$ *and* $\mathtt{G \backslash p \xrightarrow{n} q\ G'}$, $\exists$ *sorts* `s`, `s'` *and local type trees* `T`, `T'` *such that* $\mathtt{(l_1)_n = (s,\ T)}$ *and* $\mathtt{(l_2)_n = (s',\ T')}$.

The statement establishes "well-definedness" of projections with respect to the consumption.

To complete proofs of Lemmas 28, 29, 30, and 31, we apply Lemma 26 (w.r.t. participant `p`) and obtain the global type tree context, then proceed by induction on this context.

▶ **Lemma 32.** *Given* `G` $\upharpoonright_{\texttt{p}}$ `(ltt_send q l`$_1$`)` , `G` $\upharpoonright_{\texttt{q}}$ `(ltt_recv p l`$_2$`)` , `G \p` $\xrightarrow{\texttt{n}}$ `q G'` *and for every participant-process pair* `s_ind u P` *in* `M` *,* $\exists$ *local type tree* `T` *such that* `G` $\upharpoonright_{\texttt{u}}$ `T` *and* `nil nil` $\vdash$ `P: T` *, we obtain that for every participant-process pair* `s_ind u P` *in* `M` *,* $\exists$ *local type tree* `T` *such that* `G'` $\upharpoonright_{\texttt{u}}$ `T` *and* `nil nil` $\vdash$ `P: T` *.*

> This statement connects projectability, consumption, and typability of global type trees. Given a well-formed tree `G` , where `p` sends to `q` and `q` receives from `p` , and `G` types the session `M` which does not contain `p` and `q` , if `G` transitions to a well-formed `G'` by consuming the action “ `p` to `q` (with some arbitrary label `n`)”, then `G'` also types participant-process pairs in `M` .

The proof proceeds by induction on the structure of `M` , obtaining the base case thanks to Lemma 31 and Lemma 29, while the step case follows from the induction hypothesis.

▶ **Lemma 33.** *If* `G` $\upharpoonright_{\texttt{p}}$ `(ltt_send q l`$_1$`)` , `G` $\upharpoonright_{\texttt{q}}$ `(ltt_recv p l`$_2$`)` , `(xs)`$_{\texttt{n}}$ `= (s',T')` , `ltt_recv p xs` $\leqslant$ `ltt_recv p l`$_2$ *and* `ltt_send q (+[n] (s, T))` $\leqslant$ `ltt_send q l`$_1$ *, then* $\exists$ *global type tree* `G'` *such that* `G \p` $\xrightarrow{\texttt{n}}$ `q G'` *.*

> This property derives consumption information from projectability and subtyping predicates. For a well-formed global type tree `G` with projections onto `p` and `q` , where `p` sends to `q` with continuations `l`$_1$ and `q` receives from `p` with continuations `l`$_2$, if the `n` $^{\text{th}}$ elements of `l`$_1$ and `l`$_2$ are supertypes of some types `T` , `T'` (rather than being `T` , `T'` —relaxed), then consuming the communication “from `p` to `q` ” using the `n` $^{\text{th}}$ continuation is well-defined.

Inverting the second subtyping predicate and Lemma 30 reveals a sort and a local type tree. Lemma 26 provides a global type tree context, on which the proof proceeds by induction.

▶ **Lemma 34.** *If we have* `(Some S :: Gs) Gt` $\vdash$ `P: T` *and* `Gs` $\vdash$ `e: S` *then* `Gs Gt` $\vdash$ `(subst_expr_proc P e 0 0): T` *holds.*

> The statement says that substituting a typed expression in a typed process is type preserving.

The proof follows from induction on the process `P` .

▶ **Theorem 35** (subject reduction)**.** *If we have* `typ_sess M G` *and* `betaP M M'` *then* $\exists$ `G'` *such that* `typ_sess M' G'` *and* `multiC G G'` *hold.*

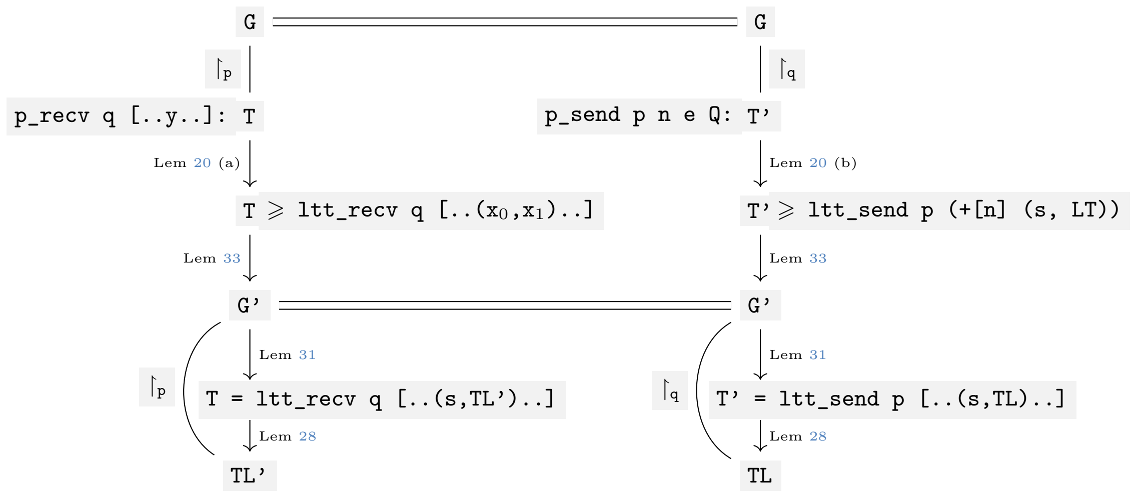
> The statement also known as *session fidelity* [26, Corollary 5.23] or *protocol conformance*.

Proof. We start with structural induction on the predicate `betaP M M'` and handle the case for `r_comm` here, skipping the remaining cases due to lack of space. In this case, we are given

$$\begin{cases} (H) & \texttt{typ\_sess (((p} \leftarrow \texttt{p\_recv q xs) ||| (q} \leftarrow \texttt{p\_send p n e Q)) ||| M) G}, \\ (Hn) & \texttt{xs}_{\texttt{n}} \texttt{ = Some y}. \end{cases}$$

with `e` reduces into the value `e_val v` , and the goal looks like

$$\begin{cases} (G_1) & \exists\ \texttt{G',typ\_sess (p} \leftarrow \texttt{subst\_expr\_proc y (e\_val v) 0 0 ||| q} \leftarrow \texttt{Q ||| M) G'} \\ (G_2) & \texttt{multiC G G'}. \end{cases}$$



In the above diagram, we illustrate a sequence of preprocessing steps to build a goal context. These steps involve inversion and lemma application to derive new hypotheses. Arrows indicate applications, straight lines show projections, and double lines share endpoints.

By inverting H and Hn, we obtain the judgments `p_recv q [..y..]: T`, `p_send p n e Q: T'`, `nil ⊢ e_val v: s`, $\texttt{G} \upharpoonright_{\texttt{p}} \texttt{T}$, and $\texttt{G} \upharpoonright_{\texttt{q}} \texttt{T'}$, for some sort `s`, and local type trees `T`, `T'`, where `[..n..]` denotes the $\texttt{n}^{\texttt{th}}$ member of a list. Lemma 20 describes the structure of `T` and `T'` with respect to subtyping: $\texttt{ltt\_recv q [..(x}_0\texttt{,x}_1\texttt{..)]} \leqslant \texttt{T}$ and $\texttt{ltt\_send p (+[n] (s, LT))} \leqslant \texttt{T'}$, for some option list $\texttt{[..(x}_0\texttt{,x}_1\texttt{..)]}$ of sort-local type tree pairs and `LT` of local type tree, such that $\texttt{(Some x}_0\texttt{ :: nil) nil} \vdash \texttt{y: x}_1$. These relations show that `T` is of the form `ltt_recv q`, and `T'` is of the form `ltt_send p`. Given this structure, Lemma 33 further establishes the existence of `G'` such that $\texttt{G \textbackslash q} \xrightarrow{\texttt{n}} \texttt{p G'}$.

From the step into `G'` and the projections of `G`, Lemma 31 implies that the $\texttt{n}^{\texttt{th}}$ continuations of `T` and `T'` are `(s, TL')` and `(s, TL)`, for some local type trees `TL` and `TL'`. Given this, Lemma 28 further provides that $\texttt{G'} \upharpoonright_{\texttt{p}} \texttt{TL'}$ and $\texttt{G'} \upharpoonright_{\texttt{q}} \texttt{TL}$.

We apply `tsess` and `tc_sub` to G_1 after substituting `G'` as the existential argument. This reduces the proof to `nil nil ⊢ (subst_expr_proc y (e_val v) 0 0): x`$_1$ and `nil nil ⊢ Q: TL`. Lemma 34 reduces the first statement to $\texttt{(Some x}_0\texttt{ :: nil) nil} \vdash \texttt{y: x}_1$ and $\texttt{nil} \vdash \texttt{e\_val v: x}_0$. The former was established earlier, and the latter follows by inverting $\texttt{ltt\_recv q [..(s,TL')..]} \geqslant \texttt{ltt\_recv q [..(x}_0\texttt{,x}_1\texttt{)..]}$ and applying `sc_sub` 🐓. The second statement follows from Lemma 32, using `G`'s projection and transition to `G'`. Finally, G_2, `multiC G G'`, follows from $\texttt{G \textbackslash p} \xrightarrow{\texttt{n}} \texttt{q G'}$. ◀

▶ **Example 36.** We show an application of Theorem 35 to [16, Ex. 3.17]. The code is here 🐓.

▶ **Lemma 37** (canonical forms for processes and sessions). (🐓, 🐓)

- *Given* `typ_sess M (gtt_send p q xs)`, ∃ *session* `M'` *such that* `unfoldP M M'` *and* `M'` *is of* `p ← P ||| q ← Q ||| M''` *or* `p ← P ||| q ← Q` *form.*
- *Given* `typ_sess M gtt_end` ∃ *session* `M'` *such that* `unfoldP M M'` *and every process in* `M'` *is either* `p_inact` *or* `p_ite e Q Q'` *and* `nil nil ⊢ (p_ite e Q Q'): ltt_end`.

The statement derives canonical forms of sessions up to unfolding.

Proof. By induction on `M`, and unfolding recursion an appropriate number of times. ◀

▶ **Theorem 38** (progress). *If* `typ_sess M G`*, then* $\exists$ *session* `M'` *such that* `betaP M M'`*, or both* `unfoldP M M'` *and every process in* `M'` *is* `p_inact`.

Proof. By a case split on `G` and matching with Lemma 37. ◀

▶ **Definition 39** (stuck). *A multiparty session* `M` *is stuck if* $\nexists$ `M'` *such that* `betaP M M'`*, and* $\nexists$ `M''` *such that both* `unfoldP M M''` *holds and every process in* `M''` *is* `p_inact`*. A session* `M` *gets stuck (* `stuckM M` *) if it reduces to a stuck session.*

▶ **Theorem 40** (non-stuck). *If* `typ_sess M G`*, then* `stuckM M → False`.

Proof. Corollary of Theorems 35 and 38. ◀

5 Related Work and Conclusion

Castro-Perez et al. [6] introduced Zooid, a domain-specific language embedded in Coq for certified multiparty communication. Zooid ensures mechanised soundness and completeness through trace equivalences between the label transition systems of local and global types, preserving properties like deadlock freedom and protocol compliance.

Tirore et al. [60] introduced a novel computable projection function, mapping global types into local types. This function is formally verified in Coq to be sound and complete with respect to its coinductive tree semantics. Their work focuses exclusively on projections.

Ekici and Yoshida [13] formalised a framework for asynchronous MPST in Coq, proving that precise subtyping, as in [17, 18], is complete. The focus is on action reorderings thus protocol optimisations in asynchronous interactions. Neither [60] nor [13] includes a process or typing calculus, missing proofs of subject reduction, progress, and type safety.

Hinrichsen et al. [22, 19, 20] developed Actris, a tool integrating separation logics with asynchronous session types (with subtyping), built on the Coq Iris program logic [40, 38, 37, 36]. Jacobs et al. extended Actris into LinearActris [35], incorporating linear logic to ensure deadlock and leak freedom. Their work is limited to binary session types.

Hinrichsen et al. [21] introduced the Multris framework, combining separation logic for verifying functional correctness with multiparty message-passing and shared-memory concurrency. They formally proved protocol consistency within the Coq Iris environment, drawing inspiration from the bottom-up approach to MPST in [53], which focuses on local types. Therefore, inherent properties of global types are not proven for Multris.

Tassarotti et al. [55] developed a compiler for a functional language with binary session types, based on a simplified version of the GV system [14], and formally verified its correctness in Coq. Jacobs et al. [34] extended this work into MPGV which enhances linear lambda calculus with multiparty sessions, supporting participant redirecting and dynamic thread spawning. Their type system includes global and local types, with local types handling linear data. Deadlock freedom is ensured by representing cyclic communication as an acyclic graph, eliminating the need for central coordination. The proof [34, Theorem 5.7] uses separation logic and configuration invariants to ensure preservation and progress, showing that configurations satisfying the invariant cannot get stuck.

Tirore [59] in his PhD thesis formalises *subject reduction* in Coq for the multiparty session π-calculus in [26], incorporating session initialisation and delegation. The type system uses *channel-explicit* global and local types, with projections derived from [60]. Channel-explicit types further require linearity checks, ensuring global types to be projectable, therefore making the formalisation harder to extend or integrate with other systems, as most session type systems (including ours) use *channel-implicit* types. In a subsequent work, Tirore et al. [58] extend the results of his thesis by formalising the proofs of communication safety and safety preservation in Coq.

Brady [5] designed secure communication protocols for binary sessions in Idris, while Thiemann et al. [57] formalised progress and preservation for binary session types in Agda.

Hirsch and Garg introduced Pirouette [23], a choreographic language with formal guarantees verified in Coq. Cruz-Filipe et al. [10] formalised the theory of choreographic programming in Coq. Pohjola et al. [51] presented Kalas, a compiler for a choreographic language whose correctness has been verified within HOL4.

Comparison. Unlike [59], our subject reduction property ensures protocol conformance (session fidelity) [26, Corollary 5.23]. We formalise *progress* and *non-stuckness* too. In contrast to [34], our language extends a core multiparty session calculus with key MPST features. The type system, based on *channel-implicit* global and local types with coinductive projections, guarantees: (1) deadlock freedom via a *top-down approach*, (2) the *non-stuck theorem* through *subject reduction* and *progress* and (3) incorporates *subtyping*. Our formalisation is designed to be extensible, allowing for future enhancements such as incorporating projection with full merging, and properties like fairness and liveness (discussed below).

In Coq. Tirore [59], Castro-Perez et al. [6], and our formalisation use inductive syntax for types and coinductive syntax for (equi-)recursive type unfoldings. In these works, projection is defined using plain merging. While [59] and [6] model consumptions using LTS semantics, we implement a coinductive step relation. These formalisations use `paco` constructs to define coinductive relations. Jacobs et al. [34] use coinductive syntax for types and corecursion to capture repetitive behaviour, formalised in Coq with native coinduction.

Formalising μ types is challenging. One approach uses infinite unfoldings over a coinductive tree, while another defines types directly within a coinductive framework. Coinductive techniques aid proof mechanisation in Coq but complicate rewriting codata (Leibniz equality is undecidable). A common solution is defining a bisimulation over coinductive structures and assuming extensionality principles, aligning bisimilarity with Coq's Leibniz equality.

Future Work. Our future work includes extensions to the coinductive full merging [62, Definition 4.23] and the proof of liveness [67]. These extensions are plausible as in many parts of the codebase, proofs will remain unaffected by changes to the *merging* operator. Additionally, statements concerning projections can often be directly reused or require minor adaptations to support the proofs needed for *liveness*.

▷ **Full Merging.** The proof for the coinductive full merge is largely self-contained, requiring modifications to the proofs of Lemmas 28 and 29. Key statements as `typ_after_step_1` remain valid and follow by induction on the global type tree context. Adjustments are needed, particularly for `typ_after_step_3_helper`, where we establish subtyping instead of strict equality. This change has minimal impact on the rest of the codebase.

▷ **Liveness.** We will introduce typing contexts—distinct from those in grafting—as participant-local type tree pairs, linked to global type trees via projection. Their reductions, based on transition labels, yield (potentially) infinite traces. Using LTL constructs, we say a trace is *live* if every enabled reduction is *eventually* executed, and this *always* holds. We aim to prove in Coq that if a global type tree has an associated typing context, then the context is live. Existing proofs using projection can be adapted for reuse at the typing context level.

References

1 Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. Inductive reasoning for coinductive types. *CoRR*, abs/2301.09802, 2023. URL: https://doi.org/10.48550/arXiv.2301.09802, arXiv:2301.09802, doi:10.48550/ARXIV.2301.09802.

2 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Designing asynchronous multiparty protocols with crash-stop failures. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 1:1–1:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.1, doi:10.4230/LIPICS.ECOOP.2023.1.

3 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Crash-stop failures in asynchronous multiparty session types. *Logical Methods in Computer Science*, 2025. URL: https://arxiv.org/abs/2311.11851.

4 Jelle Bouma, Stijn de Gouw, and Sung-Shik Jongmans. Multiparty Session Typing in Java, Deductively. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 19–27. Springer, 2023. doi:10.1007/978-3-031-30820-8_3.

5 Edwin C. Brady. Type-driven development of concurrent communicating systems. *Comput. Sci.*, 18(3), 2017. URL: https://doi.org/10.7494/csci.2017.18.3.1413, doi:10.7494/CSCI.2017.18.3.1413.

6 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

7 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in Go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.

8 David Castro-Perez and Nobuko Yoshida. Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 6:1–6:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.6, doi:10.4230/LIPICS.ECOOP.2023.6.

9 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API generation for multiparty session types, revisited and revised using scala 3. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.27, doi:10.4230/LIPICS.ECOOP.2022.27.

10 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. URL: https://doi.org/10.1007/s10817-023-09665-3, doi:10.1007/S10817-023-09665-3.

11 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022. doi:10.1145/3503221.3508404.

12 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. URL: https://doi.org/10.1007/s10703-014-0218-8, doi:10.1007/S10703-014-0218-8.

13 Burak Ekici and Nobuko Yoshida. Completeness of asynchronous session tree subtyping in Coq. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia*, volume 309 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ITP.2024.13, doi:10.4230/LIPICS.ITP.2024.13.

14 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

15 Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-contract for flexible multiparty session protocols. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.8, doi:10.4230/LIPICS.ECOOP.2022.8.

16 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *JLAMP*, 104:127–173, 2019.

17 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5:16:1–16:28, jan 2021.

18 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Logic*, 24(2), Nov 2023. doi:10.1145/3568422.

19 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in Separation Logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. doi:10.1145/3371074.

20 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris 2.0: Asynchronous session-type based reasoning in Separation Logic. *Log. Methods Comput. Sci.*, 18(2), 2022. URL: https://doi.org/10.46298/lmcs-18(2:16)2022, doi:10.46298/LMCS-18(2:16)2022.

21 Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. Multris: Functional verification of multiparty message passing in Separation Logic. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1446–1474, 2024. doi:10.1145/3689762.

22 Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021. doi:10.1145/3437992.3439914.

23 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.

24 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

25 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, pages 122–138, 1998. URL: http://dx.doi.org/10.1007/BFb0053567, doi:10.1007/BFb0053567.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. URL: http://doi.acm.org/10.1145/2827695, doi:10.1145/2827695.

27 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010. doi:10.1007/978-3-642-14107-2_16.

28 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016. doi:10.1007/978-3-662-49665-7_24.

29 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017. doi:10.1007/978-3-662-54494-5_7.

30 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013. doi:10.1145/2429069.2429093.

31 Keigo Imai, Julien Lange, and Rumyana Neykova. Kmclib: Automated inference and verification of session types from ocaml programs. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 379–386. Springer, 2022. doi:10.1007/978-3-030-99524-9_20.

32 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2020.9, doi:10.4230/LIPICS.ECOOP.2020.9.

33 Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. Validating iot devices with rate-based session types. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1589–1617, 2023. doi:10.1145/3622854.

34 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

35 Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Deadlock-free Separation Logic: Linearity yields progress for dependent higher-order message passing. *Proc. ACM Program. Lang.*, 8(POPL):1385–1417, 2024. doi:10.1145/3632889.

36 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016. doi:10.1145/2951913.2951943.

37 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent Separation Logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

38 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

39 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.*, 155:52–75, 2018. URL: https://doi.org/10.1016/j.scico.2017.10.006, doi:10.1016/J.SCICO.2017.10.006.

40 Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent Separation Logic. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer, 2017. doi:10.1007/978-3-662-54434-1_26.

41 Nicolas Lagaillardie, Ping Hou, and Nobuko Yoshida. Fearless Asynchronous Communications with Timed Session Types in Rust (Artifact). *Dagstuhl Artifacts Ser.*, 10(2):10:1–10:3, 2024. doi:10.4230/DARTS.10.2.10.

42 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.4, doi:10.4230/LIPICS.ECOOP.2022.4.

43 Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based verification of message-passing parallel programs. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 280–298. ACM, 2015. doi:10.1145/2814270.2814302.

44 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in TypeScript with routed multiparty session types. In Aaron Smith, Delphine Demange, and Rajiv Gupta, editors, *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, pages 94–106. ACM, 2021. doi:10.1145/3446804.3446854.

45 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017. URL: https://doi.org/10.1007/s00165-017-0420-8, doi:10.1007/S00165-017-0420-8.

46 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.

47 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In Peng Wu and Sebastian Hack, editors, *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, pages 98–108. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3033031.

48 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Log. Methods Comput. Sci.*, 13(1), 2017. doi:10.23638/LMCS-13(1:17)2017.

49 Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In Björn Franke, editor, *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18,* *2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 212–232. Springer, 2015. doi:10.1007/978-3-662-46663-6_11.

50 Kirstin Peters and Nobuko Yoshida. Separation and encodability in mixed choice multiparty sessions. In Pawel Sobocinski, Ugo Dal Lago, and Javier Esparza, editors, *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024*, pages 62:1–62:15. ACM, 2024. doi:10.1145/3661814.3662085.

51 Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.27, doi:10.4230/LIPICS.ITP.2022.27.

52 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.24, doi:10.4230/LIPICS.ECOOP.2017.24.

53 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

54 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE 1994*, pages 398–413, 1994. URL: http://dx.doi.org/10.1007/3-540-58184-7_118, doi:10.1007/3-540-58184-7_118.

55 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, 2017. doi:10.1007/978-3-662-54434-1_34.

56 The Coq Development Team. The Coq reference manual – release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman, 2023.

57 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 19:1–19:15. ACM, 2019. doi:10.1145/3354166.3354184.

58 Dawit Tirore, Jesper Bengtson, and Marco Carbone. Multiparty asynchronous session types: A mechanised proof of subject reduction. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 33 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:30. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025. All proofs mechanised in Coq; supplementary material at https://github.com/Tirore96/subject_reduction/tree/ECOOP2025/theories. doi:10.4230/LIPIcs.ECOOP.2025.33.

59 Dawit Legesse Tirore. *A Mechanisation of Multiparty Session Types.* PhD thesis, ITU Copenhagen, December 2024.

60 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.28, doi:10.4230/LIPICS.ITP.2023.28.

61 Thien Udomsrirungruang and Nobuko Yoshida. Top-down or bottom-up? complexity analyses of synchronous multiparty session types. *Proc. ACM Program. Lang.*, 9(POPL):1040–1071, 2025. doi:10.1145/3704872.

62 Thien Udomsrirungruang and Nobuko Yoshida. Top-Down or Bottom-Up? Complexity Analyses of Synchronous Multiparty Session Types. *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2025. doi:10.1145/3704872.

63 Martin Vassor and Nobuko Yoshida. Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*, volume 313 of *LIPIcs*, pages 41:1–41:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2024.41, doi:10.4230/LIPICS.ECOOP.2024.41.

64 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.

65 Nobuko Yoshida. Programming language implementations with multiparty session types. In Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan, editors, *Active Object Languages: Current Research Trends*, volume 14360 of *Lecture Notes in Computer Science*, pages 147–165. Springer, 2024. doi:10.1007/978-3-031-51060-1_6.

66 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D'Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.

67 Nobuko Yoshida and Ping Hou. Less is More Revisited. *CoRR*, abs/2402.16741, 2024. URL: https://doi.org/10.48550/arXiv.2402.16741, arXiv:2402.16741, doi:10.48550/ARXIV.2402.16741.

68 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 71–84. ACM, 2020. doi:10.1145/3372885.3373813.

69 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020. doi:10.1145/3428216.