Completeness of Asynchronous Session Tree Subtyping in Coq

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

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

Abstract

Multiparty session types (MPST) serve as a foundational framework for formally specifying and verifying message passing protocols. *Asynchronous subtyping* in MPST allows for typing optimised programs preserving type safety and deadlock freedom under asynchronous interactions where the message order is preserved and sending is non-blocking. The optimisation is obtained by message reordering, which allows for sending messages earlier or receiving them later. Sound subtyping algorithms have been extensively studied and implemented as part of various programming languages and tools including C, Rust and C-MPI. However, formalising all such permutations under sequencing, selection, branching and recursion in session types is an intricate task. Additionally, checking asynchronous subtyping has been proven to be undecidable.

This paper introduces the first formalisation of asynchronous subtyping in MPST within the Coq proof assistant. We first decompose session types into *session trees* that do not involve branching and selection, and then establish a coinductive refinement relation over them to govern subtyping. To showcase our formalisation, we prove example subtyping schemas that appear in the literature, all of which cannot be verified, at the same time, by any of the existing decidable sound algorithms.

Additionally, we take the (inductive) negation of the refinement relation from a prior work by Ghilezan et al. [22] and re-implement it, significantly reducing the number of rules (from eighteen to eight). We establish the completeness of subtyping with respect to its negation in Coq, addressing the issues concerning the negation rules outlined in the previous work [22].

In the formalisation, we use the greatest fixed point of the least fixed point technique, facilitated by the `paco` library, to define coinductive predicates. We employ parametrised coinduction to prove their properties. The formalisation consists of roughly 10K lines of Coq code, accessible at: `https://github.com/ekiciburak/sessionTreeST/tree/itp2024`.

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 asynchronous multiparty session types, session trees, subtyping, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.13

Supplementary Material
Software (Source Code): `https://github.com/ekiciburak/sessionTreeST/tree/itp2024` [17]
archived at `swh:1:dir:33823a0054801bcf4ea95f2dffe733579cbd53c8`

Funding This work is partially supported by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We would like to thank Dawit Tirore, Marco Giunti and Mukesh Tiwari for their feedback on the previous versions of this paper; Jovanka Vanja Pantovic and Alceste Scalas for a comprehensive discussion on the negation of the refinement relation. We also thank anonymous referees for their constructive input.

© Burak Ekici and Nobuko Yoshida;
licensed under Creative Commons License CC-BY 4.0
15th International Conference on Interactive Theorem Proving (ITP 2024).
Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 13; pp. 13:1–13:20
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

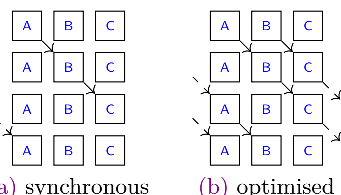
1 Introduction

Software systems often consist of concurrent and distributed components that interact through message-passing based on predefined communication protocols. Ensuring that each component adheres to the specified protocol is crucial to prevent runtime failures like communication errors and deadlocks. Session types have emerged as a successful solution to this challenge [27, 36], originally devised to two-party protocols like client-server interactions and later expanded to handle multiparty protocols as well [21, 41]. Session types offer a type-based approach to statically validate if a process conforms to a specified protocol.

A crucial challenge in employing session types lies in determining whether it is feasible to replace a part of the protocol $\mathbb{T}$ with another $\mathbb{T}'$ without violating safety and deadlock-freedom. This concept is referred to as session *subtyping* [18, 16], denoted by $\mathbb{T}' \leqslant \mathbb{T}$, when $\mathbb{T}'$ is a subtype of $\mathbb{T}$.

It becomes even more challenging to formalise the precise subtyping in *asynchronous* interactions where the send operation is *non-blocking*. The asynchronous nature permits message reordering, facilitating the sending of messages earlier or delaying their reception and opening up the possibility for *protocol optimisations*. To exemplify this, we take the `ring-choice` protocol in [13], which orchestrates three participants A, B and C:

1. A sends an integer n to B with the label *add*.
2. B sends an integer m to C, labelled either *add* or *sub*.
   a. If C receives the integer m labelled *add*, it sends an integer $m+k$ back to A with the *add* label, and the protocol restarts from step 1.
   b. If C receives the integer m labelled *sub*, it sends an integer $m-k$ to A with the *sub* label, and the protocol restarts from step 1.



(a) synchronous interactions (b) optimised interactions

Source: [13]

Certainly, during *synchronous* interactions (a), no data flow would occur from B to C or from C to A before B receives data from A. However, under *asynchronous* interactions (b), assuming that each participant begins with its own initial value, B can *concurrently* send data (with different labels) to C *before* receiving data from A, letting C to start the next iteration by sending data to A.

The synchronous interactions from B's local viewpoint could be represented by a *session type* $\mathbb{T}_\mathsf{B}$, which can then be optimised into the type $\mathbb{T}_\mathsf{B}{}^{\text{opt}}$ under asynchronous interactions as specified in Figure 1. The notation "!" is read as "send to" while "?" denotes "receive from"

$$\mathbb{T}_\mathsf{B} = \mu\mathbf{t}.\mathsf{A}?add(\text{i32}).\bigoplus \mathsf{C}!\begin{cases} add(\text{i32}).\mathbf{t} \\ sub(\text{i32}).\mathbf{t}\end{cases} \qquad \mathbb{T}_\mathsf{B}^{\text{opt}} = \mu\mathbf{t}.\bigoplus \mathsf{C}!\begin{cases} add(\text{i32}).\mathsf{A}?add(\text{i32}).\mathbf{t} \\ sub(\text{i32}).\mathsf{A}?add(\text{i32}).\mathbf{t}\end{cases}$$

Figure 1 Local type $\mathbb{T}_\mathsf{B}$ and its optimised local type $\mathbb{T}_\mathsf{B}^{\text{opt}}$ (view) of B.

actions, and "i32" is the integer sort of payloads. The *selection type* $\bigoplus$ denotes the internal choice of actions (label, payload sort, continuation triples) directed towards a particular participant. Dually, the *branching type* $\&$ is the external choice of actions from a participant. The symbol μ denotes the recursion binder.

The optimisation pictured in Figure 1 is handled simply by reordering "send to C" and "receive from A" actions. The type of optimised interactions $\mathbb{T}_\mathsf{B}{}^{\text{opt}}$ is said to be an *asynchronous subtype* [21, 22] of the type $\mathbb{T}_\mathsf{B}$, and can safely replace it within the protocol while maintaining deadlock-freedom.

▶ Note 1. Throughout the paper, we hyperlink the related Coq sources to the symbol 🐓.

We provide the first Coq [38] library (`https://github.com/ekiciburak/sessionTreeST/tree/itp2024`) that handles the internal dynamics of asynchronous subtyping for MPTS [22] and proves the optimisation summarised in Figure 1 and four more examples from the literature: Examples 3.17 3.19 and 4.14 in [22] 🐓. Notice that no decidable sound subtyping algorithm in the literature [14, 5, 10, 2] can verify examples all together (see § 6).

We then prove a completeness theorem of subtyping with respect to its negation. The Coq proof of completeness involves reorganising the subtyping relation by reformulating the underlying refinement relation and its negation, proposed by Ghilezan et al. [21, 22] 🐓.

In the reformulation of refinement, we accommodate the possibility of including the empty prefix ε in term syntax, leading to the definition of the relation with two fewer rules than [22, Definition 3.3] (see Figure 5). This simplification facilitates the proof of an inversion lemma, as elaborated in Remark 6 and Lemma 7.

Regarding the reformulation of the refinement negation, we reduce the number of rules from eighteen in [22, Fig. 6] to eight, thereby rendering the remaining ten rules provable. This is done by introducing a new sort of term prefixing ($\mathcal{C}^{(\mathsf{p})}$ – Lemma 12) and using it to modify some of the original rules in order to adopt a better structural shape of rules and become more readily applicable within proofs. Further details are covered in Lemmata 17, 14, 18 and 20, and in Remark 19.

The accompanying Coq library can be used to certify additional asynchronous protocol optimisations in MPST. This entails defining both the original and optimised protocols, then applying either of the two main refinement rules (see Figure 5) to show that the latter is a subtype of the former. In addition, provided that the subtyping is obtained by the use of coinductive structures in Coq, applications dealing with infinite trees could also leverage the structures and lemmata present in the library.

2 Session Trees and Subtyping

The subtyping of session types [16, 18] plays a crucial role in process calculi, as a process that instantiates a session type $\mathbb{T}$ can securely substitute another process inhabiting a supertype $\mathbb{T}'$ of $\mathbb{T}$. Such substitution contributes to the development of more optimised protocols [21, 22].

Each *closed* asynchronous session type $\mathbb{T}$ is associated with a corresponding session tree $\mathsf{T} = \mathcal{T}(\mathbb{T})$. Refer to [20, Def. A.14] for the definition of the translation function $\mathcal{T}\colon \mathbb{T} \to \mathsf{T}$. Therefore, the definition of a subtyping relation could be captured by the use of session trees. With this perspective, in this work we introduce a Coq library that implements asynchronous session trees together with various property proofs.

A session tree is *coinductively* defined with the following syntax that reflects in Coq 🐓 in the way listed alongside:

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

```
CoInductive st: Type ≜
  | st_end     : st
  | st_receive: participant → list (label*sort*st) → st
  | st_send    : participant → list (label*sort*st) → st.
Notation "p '?'' l" ≜ (st_receive p l).
Notation "p '!' l" ≜ (st_send p l).
```

The constructor $\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i$ denotes *branching* (or *external choice*) interactions and represents a set of messages towards participant p with labels ℓ_i, payload sorts S_i and continuations T_i. While $\bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i$ stands for *selection* (or *internal choice*) and specifies a set of messages from p with labels ℓ_i, payload sorts S_i and continuations T_i (for some $i \in I$); the constructor `end` signals termination of interactions.

In both the code alongside and the rest of the paper, we use the notation “?’” and `st_receive` constructor interchangeably, as well as the notation ‘!’ and `st_send` constructor. We exclude the symbols $\&$ and $\bigoplus$ but maintain their functionality using Coq lists. That is, labels, sorts and continuations for selections and branchings are represented in Coq by lists of label-sort-continuation triples. See the constructors `st_receive` and `st_send` in the above code snippet.

The objective in checking “whether a session tree T qualifies a subtype (subtree) of another tree T' ($\mathsf{T} \leqslant \mathsf{T}'$)” is twofold:

1. the *decomposition* of both trees into sets of *single-input-single-output* (SISO) trees, and
2. checking whether it is possible to find SISO trees W, from the decomposition of considered subtree, and W', from the decomposition of considered supertree, such that W is a *refinement* of W'. That is, there exist certain ways to reorder the actions in W so that it matches the structure of actions in W'. See Definition 5 for further details on refinement.

2.1 SI, SO and SISO Trees

In [22], the decomposition of a given session tree T into a set of SISO trees is not accomplished all at once; instead, it involves intermediate steps. Initially, T is partitioned into a set of trees where each tree is characterised by singleton choices in their selections (referred to as *single-output* (*SO*) trees). Subsequently, for each individual SO tree, a further set of trees is formed where the members exhibit singleton branchings (referred to as *single-input* (*SI*) trees). Therefore, consecutively applying SO and SI decompositions (in any order) to a session tree eventually yields in a set of SISO trees.

In what follows, we *coinductively* present SO (denoted U), SI (denoted V) and SISO (denoted W) trees. In SO trees, there is a list of branchings but a single selection while SI trees contain a list of selections with a single branching

$$\mathsf{U} \ ::= \ \mathsf{end} \ \mid \ \mathop{\&}_{i \in I} \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{U}_i \ \mid \ \mathsf{p}!\ell(\mathsf{S}).\mathsf{U} \qquad \mathsf{V} \ ::= \ \mathsf{end} \ \mid \ \mathsf{p}?\ell(\mathsf{S}).\mathsf{V} \ \mid \ \bigoplus_{i \in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{V}_i$$

and SISO trees are made of single branching and single selection 🐓:

$$\mathsf{W} \ ::= \ \mathsf{end} \ \mid \ \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \ \mid \ \mathsf{p}!\ell(\mathsf{S}).\mathsf{W}$$

```
Inductive singletonI (R: st → Prop): st → Prop ≜
  | ends : singletonI R st_end
  | sends: ∀ p l s w, R w → singletonI R (st_send p [(l,s,w)])
  | recvs: ∀ p l s w, R w → singletonI R (st_receive p [(l,s,w)]).
Definition singleton s ≜ paco1 (singletonI) bot1 s.
Class siso: Type ≜ mk_siso { und: st; sprop: singleton und }.
```

Formalisation of SISO trees in Coq initiates with the declaration of a `Prop` valued inductive predicate `singletonI` which serves for verifying “whether the selections and branchings within a given session tree are singletons”. We then leverage the “greatest fixed point of the least fixed point” technique facilitated by the `paco` library [29], and generate the type `singleton` as the greatest fixed point of `singletonI`; so that the latter is applicable to infinite session trees. We then formulate SISO trees as a sigma type of a session tree `und` such that `und` respects `singleton`.

This technique has been employed by Zakowski et al. [43] to define weak bisimilarity on streams, and Tirore et al. [40] to define (sound and complete) projection of global session types onto local types.

▶ Remark 2. The use of `paco` library is beneficial – in many constructions presented through our the paper – since it allows for coinductive reasoning *parametrised* by “accumulated knowledge” so that proof goals could be closed upon encountering something that is already in

the knowledge set through out coinductive folding steps. Furthermore, `paco` utilises semantic guardedness rather than relying on syntactic guard checks, which can be problematic and compromised even through straightforward setoid rewrites.

We bypass the intermediate SO and SI decompositions and directly build a coinductive relation that inhabits SISO tree and session tree pairs such that former is obtained by decomposing the latter. This approach is slightly different from the one outlined in § 3.4 of [22], but it better aligns with Coq formalisation.

▶ **Definition 3.** *The SISO decomposition of a session tree is governed by the coinductive relation* $\leftarrow\!\!\sim\!\!\sim$ *with the following rules:*

$$\frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{W} \leftarrow\!\!\sim\!\!\sim \mathsf{T}_k}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \leftarrow\!\!\sim\!\!\sim \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\textsc{siso-rcv}] \qquad \frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{W} \leftarrow\!\!\sim\!\!\sim \mathsf{T}_k}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \leftarrow\!\!\sim\!\!\sim \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\textsc{siso-snd}]$$

$$\frac{}{\mathtt{end} \leftarrow\!\!\sim\!\!\sim \mathtt{end}}\ [\textsc{siso-end}]$$

for some finite set of indices I*.*

The relation $\leftarrow\!\!\sim\!\!\sim$ provided in Definition 3 is coinductively implemented in Coq under the name `st2sisoC`, as shown below. This implementation operates at the level of session trees instead of SISO trees, to avoid the need for singleton checks at each step of rule application. However, we ensure that the relation is instantiated with the underlying `und` of a `siso` tree whenever it is called. Refer to the formal subtyping definition, `subtype`, outlined in Definition 10, for instance. We maintain this methodology until § 4.2, where we establish the negation of the refinement relation, `nRefinement`, directly over `siso` trees.

```
Inductive st2siso (R: st → st → Prop): st → st → Prop ≜
  | st2siso_end: st2siso R st_end st_end
  | st2siso_rcv: ∀ l s x xs p, R x (pathsel l xs) → st2siso R (p ?' [(l,s,x)]) (st_receive p xs)
  | st2siso_snd: ∀ l s x xs p, R x (pathsel l xs) → st2siso R (p ! [(l,s,x)]) (st_send p xs) .
Definition st2sisoC s1 s2 ≜ paco2 (st2siso) bot2 s1 s2.
```

The function `pathsel` selects the path among the list of selections and branchings that matches the label of the current SISO action.

```
Fixpoint pathsel (u: label) (l: list (label*sort*st)): st ≜
  match l with
    | (lbl,s,x)::xs ⇒ if eqb u lbl then x else pathsel u xs
    | nil           ⇒ st_end
  end.
```

It returns T_k under the condition $\ell = \ell_k$ within the context of the rules [siso-rcv] and [siso-snd].

2.2 SISO Tree Refinement

The other key component of checking whether a given session tree is a subtree (or supertree) of another is the support for action reordering. Conceptually, the subtree has the capability to “anticipate” certain input/output actions that are expected to take place in the supertree. This anticipation is captured by action reordering [22, Def. 3.2], namely executing actions earlier or later than their prescribed occurrence.

▶ **Definition 4.** (🔗, 🔗). To elucidate action reorderings, a pair of input/output sequences are recursively defined below

$$\begin{array}{lcl} \mathcal{A}^{(\mathsf{p})} & ::= & \varepsilon \mid \mathsf{q}?\ell(\mathsf{S}) \mid \mathsf{q}?\ell(\mathsf{S}).\mathcal{A}^{(\mathsf{p})} \\ \mathcal{B}^{(\mathsf{p})} & ::= & \varepsilon \mid \mathsf{r}?\ell(\mathsf{S}) \mid \mathsf{q}!\ell(\mathsf{S}) \mid \mathsf{r}?\ell(\mathsf{S}).\mathcal{B}^{(\mathsf{p})} \mid \mathsf{q}!\ell(\mathsf{S}).\mathcal{B}^{(\mathsf{p})} \quad (\mathsf{q} \neq \mathsf{p}) \end{array}$$

The $\mathcal{A}^{(\mathsf{p})}$ prefix refers to a finite sequence of actions containing all possible receives excluding those from the participant p. $\mathcal{B}^{(\mathsf{p})}$, on the other hand, indicates a finite sequence that involves all receives and all sends but not those towards participant p.

▶ **Definition 5.** *The refinement relation $\lesssim$ over SISO trees is coinductively defined with:*

$$\frac{\mathsf{S}' \leq: \mathsf{S} \quad \mathsf{W} \lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{W}' \quad \mathsf{act}(\mathsf{W}) = \mathsf{act}(\mathcal{A}^{(\mathsf{p})}.\mathsf{W}')}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'} \text{ [REF-}\mathcal{A}\text{]}$$

$$\frac{\mathsf{S} \leq: \mathsf{S}' \quad \mathsf{W} \lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{W}' \quad \mathsf{act}(\mathsf{W}) = \mathsf{act}(\mathcal{B}^{(\mathsf{p})}.\mathsf{W}')}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'} \text{ [REF-}\mathcal{B}\text{]} \qquad \frac{}{\mathsf{end} \lesssim \mathsf{end}} \text{ [REF-END]}$$

The symbol $\leq:$ denotes the least reflexive relation over payload sorts (i.e., `nat` $\leq:$ `int`) while the function `act` coinductively accumulates the actions, participant-*dir* pairs where $dir \in \{!, ?\}$, of a given SISO tree into a stream (or a colist/coseq). Action equality checks serve the purpose of ensuring that rule applications neither introduce nor remove actions.

The rule [REF-$\mathcal{B}$] in general captures the reordering backed by $\mathcal{B}^{(\mathsf{p})}$ prefixes. It allows for the reordering, a finite number of times, of an output directed towards a participant p with any input and output combinations, excluding other outputs directed towards the participant p. While the rule [REF-$\mathcal{A}$] anticipates the reordering of an input from a participant p with any input combination but not those from p.

▶ Remark 6. As opposed to the original definition of refinement relation given in [22, Def. 3.2], we allow prefixes $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ to include the empty prefix ε. This deviation indeed introduces an important flexibility in the framework. By permitting this, we are essentially acknowledging the possibility of contexts without any actions, which can be crucial for certain proofs and reasoning processes. It particularly allows for the proof of an inversion lemma, which asserts that *SISO trees with action dis-equality cannot refine each other*. This is one of the key results of our Coq formalisation.

▶ **Lemma 7.** $\forall \mathsf{W}\ \mathsf{W}', \quad \mathsf{act}(\mathsf{W}) \neq \mathsf{act}(\mathsf{W}') \implies \neg(\mathsf{W} \lesssim \mathsf{W}')$.

This lemma is a significant result, as it establishes a fundamental property regarding the relationship between terms with action mismatch. Note that the action the dis-equality definition here is obtained by negating the statement in Definition 15.

Below is a representation of the refinement relation $\lesssim$ in a Coq implementation where the rule [REF-$\mathcal{B}$] is referred to as `ref_b` while [REF-$\mathcal{A}$] is named `ref_a`.

```
Inductive dir: Type ≜ rcv: dir | snd: dir.
CoFixpoint act (t: st): coseq (participant * dir) ≜
  match t with
    | st_receive p [(l,s,t')] ⇒ cocons (p, rcv) (act t')
    | st_send p [(l,s,t')]    ⇒ cocons (p, snd) (act t')
    | _                       ⇒ conil
  end.
Inductive refinementR (R: st → st → Prop): st → st → Prop ≜
  | ref_a  : ∀ w w' p l s s' a n, subsort s s' → seq w (merge_ap_contn p a w' n) →
             ( ∃ L1, ∃ L2, coseqInLC (act w) L1 ∧ coseqInLC (act (merge_ap_contn p a w' n)) L2 ∧
                           coseqInR L1 (act w) ∧ coseqInR L2 (act (merge_ap_contn p a w' n)) ∧
                           (∀ x, List.In x L1 ↔ List.In x L2) ) →
             refinementR R (st_receive p [(l,s,w)]) (merge_ap_contn p a (st_receive p [(l,s',w')]) n)
  | ref_b  : ∀ w w' p l s s' b n, subsort s s' → seq w (merge_bp_contn p b w' n) →
             ( ∃ L1, ∃ L2, coseqInLC (act w) L1 ∧ coseqInLC (act (merge_bp_contn p b w' n)) L2 ∧
                           coseqInR L1 (act w) ∧ coseqInR L2 (act (merge_bp_contn p b w' n)) ∧
                           (∀ x, List.In x L1 ↔ List.In x L2) ) →
             refinementR R (st_send p [(l,s,w)]) (merge_bp_contn p b (st_send p [(l,s',w')]) n)
  | ref_end: refinementR seq st_end st_end.
Definition refinement: st → st → Prop ≜ fun s1 s2 ⇒ paco2 refinementR bot2 s1 s2.
```

The function merge_bp_contn takes a participant p, an instance b of $\mathcal{B}^{(\mathsf{p})}$, a natural number n and a session tree w. It repeats the action b, n times, and prefixes it to the tree w. There are analogous constructs for $\mathcal{A}^{(\mathsf{p})}$ type of prefixes which we omit elucidating here. The code blocks under the existential quantifiers ∃ validate action equalities according to Definition 8.

2.3 Action Equalities

One key point in the context of the refinement relation is to decide the equalities over streams of actions modulo action reordering. There, the focus lies not on assessing structural equality between streams, but rather on discerning the similarity of their constituent elements. A potential strategy to achieve this is having a coinductive definition of stream membership, and checking if a pair of streams have matching members.

```
Inductive coseqInC {A: Type} (R: A → coseq A → Prop): A → coseq A → Prop ≜
  | CoInSplit1A x xs {ys}: xs = cocons x ys → coseqInC R x xs
  | CoInSplit2A x xs y ys: xs = cocons y ys → x ≠ y → R x ys → coseqInC R x xs.
Definition coseqCoIn {A} ≜ paco2 (@coseqInC A) bot2.
```

This coinductive approach turns out to be unsound as it allows for proving the existence of a 'b' within the stream of 'a's where a ≠ b.

```
CoFixpoint W {A: Type} (a: A): coseq A ≜ cocons a (W a).
Lemma unsound_coseqCoIn: ∀ A (a b: A), a ≠ b → coseqCoIn b (W a).
```

We proceed by assuming that any stream of actions adheres to a reasonable notion of finiteness, meaning it comprises only a finite set of distinct actions. This assumption naturally aligns with the framework of multiparty session types. Even in scenarios where sessions involve an infinite number of interactions, these interactions must occur among a finite number of participants [21, 22], leading to finitely many distinct actions. Consequently, it becomes feasible to state that a pair of streams share identical members if and only if the list of (distinct) actions is contained within the stream of actions. We do not explicitly state this as an axiom in Coq, rather massage it into the action equality check formulated in Definition 8.

▶ **Definition 8.** *For a pair of SISO trees* W *and* W'*, we define action equality as follows:*

$$\exists l_1\ l_2,\quad l_1 \in_I \mathtt{act}(\mathsf{W}) \ \wedge\ \mathtt{act}(\mathsf{W}) \in_C l_1 \ \wedge\ l_2 \in_I \mathtt{act}(\mathsf{W}') \ \wedge\ \mathtt{act}(\mathsf{W}') \in_C l_2 \ \wedge$$
$$(\forall x,\ \mathtt{mem}\ x\ l_1 \iff \mathtt{mem}\ x\ l_2)$$

where the relation $\in_I$ inductively traverses a given action list and checks if every list member is in the stream, while $\in_C$ coinductively folds a provided stream of actions, and checks if every stream member is in the list. These relations are formally defined employing the following constructors:

$$\frac{}{nil \in_I w}\,[\text{I-NIL}] \qquad \frac{x \in w \quad xs \in_I w}{(x :: xs) \in_I w}\,[\text{I-CONS}] \qquad \frac{}{conil \in_C l}\,[\text{C-NIL}] \qquad \frac{\mathtt{mem}\ x\ l \quad xs \in_C l}{(cocons\ x\ xs) \in_C l}\,[\text{C-CONS}]$$

The symbol ∈ represents the inductive stream membership check, associated with the predicate coseqIn in the following code snippet, whereas mem denotes the typical list membership check.

▶ **Lemma 9.** *Given* $\mathsf{W} := \mathsf{p}!\ell_1(\mathsf{S}_1).\mathsf{p}?\ell_2(\mathsf{S}_2).\mathsf{q}!\ell_3(\mathsf{S}_3).\mathsf{W}$ *and* $l := \mathsf{p}? :: \mathsf{p}! :: \mathsf{q}! :: nil$*, we have* (1) $\mathtt{act}(\mathsf{W}) \in_C l$ *and* (2) $l \in_I \mathtt{act}(\mathsf{W})$.

Proof. To close the first item, we apply the constructor [C-CONS] three times making sure that p!, p? and q! are in l, and then employ the coinduction hypothesis. The proof of the second item proceeds by applying the constructor [I-CONS] three times, ensuring that p?, p!, and q! are in act(W). Finally, one more application of [I-NIL] suffices to demonstrate the goal. ◀

We formalise the relation $\in_C$ (resp., $\in_I$) in Coq, denoted as coseqInLC (resp., coseqInR).

```
Inductive coseqIn: (participant * dir) → coseq (participant * dir) → Prop ≜
  | CoInSplit1 x xs y ys: xs = cocons y ys → x = y → coseqIn x xs
  | CoInSplit2 x xs y ys: xs = cocons y ys → x ≠ y → coseqIn x ys → coseqIn x xs.
Inductive coseqInL (R: coseq (participant * dir) → list (participant * dir) → Prop):
                   coseq (participant * dir) → list (participant * dir) → Prop ≜
  | c_nil : ∀ ys, coseqInL R conil ys
  | c_incl: ∀ x xs ys, List.In x ys → R xs ys → coseqInL R (cocons x xs) ys.
Definition coseqInLC ≜ fun s1 s2 ⇒ paco2 (coseqInL) bot2 s1 s2.
Inductive coseqInR: list (participant * dir) → coseq (participant * dir) → Prop ≜
  | i_nil : ∀ ys, coseqInR nil ys
  | i_incl: ∀ x xs ys, coseqIn x ys → coseqInR xs ys → coseqInR (x::xs) ys.
```

This strategy remains effective for proving subtyping examples discussed in § 3.1 as it allows us to store actions of specifically given trees into finite lists and perform action equality checks via list membership comparisons. However, it becomes cumbersome when aiming to prove completeness of subtyping. This is because it is not useful for proving general properties about tree shapes with prefixes; see § 4.1, especially Lemmata 12 and 13.

3 Asynchronous Subtyping

▶ **Definition 10.** *The asynchronous subtyping relation $\leqslant$ over session trees is defined as:*

$$\frac{\exists\{(\mathsf{W}_i, \mathsf{W}'_i) \mid i \in I\} \qquad \mathsf{W}_i \leftharpoonup\!\!\sim\!\sim \mathsf{T} \qquad \mathsf{W}'_i \leftharpoonup\!\!\sim\!\sim \mathsf{T}' \qquad \mathsf{W}_i \lesssim \mathsf{W}'_i}{\mathsf{T} \leqslant \mathsf{T}'}$$

for some finite set of indices I.

We revisit the original subtyping definition in [22, Def. 3.13] such that it relies on the direct decomposition into SISO trees as in Definition 3. The intuitive idea is then to plug in a list of SISO tree pairs $(\mathsf{W}_i, \mathsf{W}'_i)$, where each W_i is part of the SISO decomposition of T, similarly each W'_i is part of the decomposition of T', and check whether W_i refines W'_i.

```
Definition subtype (T T': st): Prop ≜ ∃ (l: list (siso*siso)), decomposeL l T T' ∧ listSisoPRef l.
```

The function `decomposeL` verifies if the first projection of each pair in the list `l` is a SISO tree taken from decomposing T, and if the second projections are from T'. While, the function `listSisoPRef` is used to conduct refinement checks employing the `refinement` relation.

3.1 Subtyping Example

An instance of optimisation managed by asynchronous subtyping arises in the protocol for distributed batch processing (Example 4.14 in [22]), where a particular segment of the protocol is replaced with another. We have this subtyping proof formalised in Coq, not at the level of session types but their induced session trees. In consideration of space limitations, we omit this proof and instead introduce another optimisation case addressed by subtyping (Example 3.19 in [22]), which is more involved as it contains coinductive but non-cyclic derivations. Consider the following session types:

$$\mathbb{T} = \mu\mathbf{t}_1.\,\&\,\mathsf{p}?\begin{cases}\ell_1(\mathsf{S}).\mathsf{p}!\ell_3(\mathsf{S}).\mathsf{p}!\ell_3(\mathsf{S}).\mathsf{p}!\ell_3(\mathsf{S}).\mathbf{t}_1\\ \ell_2(\mathsf{S}).\mu\mathbf{t}_2.\mathsf{p}!\ell_3(\mathsf{S}).\mathbf{t}_2\end{cases} \qquad \mathbb{T}' = \mu\mathbf{t}_1.\,\&\,\mathsf{p}?\begin{cases}\ell_1(\mathsf{S}).\mathsf{p}!\ell_3(\mathsf{S}).\mathbf{t}_1\\ \ell_2(\mathsf{S}).\mu\mathbf{t}_2.\mathsf{p}!\ell_3(\mathsf{S}).\mathbf{t}_2\end{cases}$$

Figure 2 Example session types with $\mathbb{T}'$ is a subtype of $\mathbb{T}$ from [22, Example 3.19].

We translate the types $\mathbb{T}$ and $\mathbb{T}'$ into respective session trees `TB` and `TB'` manually, and then develop them in Coq. The operational aspect of the recursion binder μ is addressed by Coq's `CoFixpoint` vernacular as session trees are coinductively defined.

```
CoFixpoint TS : st ≜ "p"![("l3",I,TS)].
CoFixpoint TB : st ≜ "p"?'[("l1",I,"p"![("l3",I,"p"![("l3",I,"p"![("l3",I,TB)])])]); ("l2",I,TS)].
CoFixpoint TB': st ≜ "p"?'[("l1",I,"p"! [("l3",I,TB')]); ("l2",I,TS)].
```

Before proceeding with the proof steps, we introduce some tree terms and prefixes that will be used later within the proof.

```
CoFixpoint WB : st ≜ "p"?'[("l1",I,"p"![("l3",I,"p"![("l3",I,"p"![("l3",I,WB)])])])].
CoFixpoint WB': st ≜ "p"?'[("l1",I,"p"![("l3",I,WB')])].
Definition pi1: Dp ≜ "p"?' "l1" I ("p!" "l3" I).
Definition pi3: Dp ≜ "p"?' "l1" I ("p!" "l3" I ("p!" "l3" I ("p!" "l3" I))).
CoFixpoint WD: st ≜ "p"!'[("l3",I,WD)].
Definition WA: st ≜ "p"?'[("l2",I,WD)].
```

The type `Dp` 🐓 is inhabiting SISO style (without branching and selection) term prefixes. To prove that `TB` $\leqslant$ `TB'` holds, we are supposed to

(1) show that `(WB,TB)`, `(WB',TB')`, `(WA,TB)` and `(WA,TB')` are in `st2sisoC`

(2) and demonstrate that `WB` $\lesssim$ `WB'` with `(pi3)`^{n} $\cdot$ `WA` $\lesssim$ `(pi1)`^{n} $\cdot$ `WA` for all naturals n. The infix function `t` $\cdot$ `W` glues a prefix term `t` (of type `Dp`) to a SISO tree `W`, and the superscript n denotes the repetition of the prefix n times before glueing.

The complication in refinement stated in item (2) above is due to the complex co-recursive structure of terms `TB` and `TB'`. Intuitively, the former case covers the refinement for the outer recursive structure while the latter is supposed to deal with the inner one.

Proof. 🐓 We skip the last two cases of the item (1) and the last case of the item (2) due to space constraints, and begin by proving that pairs (`WB`,`TB`) and (`WB'`,`TB'`) are in `st2sisoC`. To address the former case, we apply the rule `st2siso_rcv` once and `st2siso_snd` three times. We then employ the coinductive hypothesis that saves the initial proof state. The proof of the second case shares commonalities. It can be effectively handled by first applying the `st2siso_rcv` rule followed by `st2siso_snd`, and then invoking the coinductive hypothesis.

Proving that `WB` $\lesssim$ `WB'` holds however presents a more intriguing scenario. For that, a pen-and-paper proof is structured in Figure 3 (read: bottom left $\rightarrow$ top left $\xrightarrow{\text{generalised by}}$ bottom right $\rightarrow$ top right). Steps on the left are straightforward refinement rule applications,

$$\begin{array}{c}
\mathsf{WB} \lesssim \mathsf{p}?\ell_1(\mathsf{S}).\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{WB}' \\
\hline\hline
\mathsf{p}!\ell_3(\mathsf{S}).\mathsf{WB} \lesssim \mathsf{p}?\ell_1(\mathsf{S}).\mathsf{WB}' \\
\hline\hline
(\mathsf{p}!\ell_3(\mathsf{S}))^2.\mathsf{WB} \lesssim \mathsf{WB}' \\
\hline\hline
(\mathsf{p}!\ell_3(\mathsf{S}))^3.\mathsf{WB} \lesssim \mathsf{p}!\ell_3(\mathsf{S}).\mathsf{WB}' \\
\hline\hline
\mathsf{WB} \lesssim \mathsf{WB}'
\end{array}
\begin{array}{l}
[\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{A}]
\end{array}
\qquad
\begin{array}{c}
\mathsf{WB} \lesssim (\mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{p}?\ell_1(\mathsf{S}))^{n+1}.\mathsf{WB}' \\
\hline\hline
\mathsf{p}!\ell_3(\mathsf{S}).\mathsf{WB} \lesssim (\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{p}?\ell_1(\mathsf{S}_1))^n.\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{WB}' \\
\hline\hline
(\mathsf{p}!\ell_3(\mathsf{S}))^2.\mathsf{WB} \lesssim (\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{p}?\ell_1(\mathsf{S}))^n.\mathsf{WB}' \\
\hline\hline
(\mathsf{p}!\ell_3(\mathsf{S}))^3.\mathsf{WB} \lesssim (\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{p}?\ell_1(\mathsf{S}))^n.\mathsf{p}!\ell_3(\mathsf{S}).\mathsf{WB}' \\
\hline\hline
\mathsf{WB} \lesssim (\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{p}?\ell_1(\mathsf{S}))^n.\mathsf{WB}'
\end{array}
\begin{array}{l}
[\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{B}] \\ [\text{REF-}\mathcal{A}]
\end{array}$$

Figure 3 Proof steps of $\mathsf{WB} \lesssim \mathsf{WB}'$. (*Source:* [22, Example 3.19])

where the topmost derivation is complemented by the helper steps on the right for every natural number n. These auxiliary steps can be proven by conducting a case analysis on n, supported by a "stronger" coinduction hypothesis universally quantifying over n.

In a Coq implementation, however, we take a slightly different approach. We consider merging WB' with a single prefix $\mathsf{p}?\ell_1(\mathsf{S})$ and ensure that this happens an even number of times. Below lemma aligns with the bottommost line of the helper steps in Figure 3.

```
Lemma WBRef: ∀ n, ev n → refinement WB  (merge_bp_contn "p" (bp_receivea "p" "l1" sint) WB' n).
```

The term bp_receivea in the lemma statement corresponds to the $\mathsf{r}?\ell(\mathsf{S}).\mathcal{B}^{(\mathsf{p})}$ constructor of $\mathcal{B}^{(\mathsf{p})}$, enabling the prefixing of a receive action from any participant to a given session tree. Consequently, in the statement, the right-hand side of the refinement represents a SISO tree where the action p?' is executed an even number (n) of times before being succeeded by actions from WB'.

To develop this lemma in Coq, we begin by storing the proof state within a coinduction hypothesis CIH, universally quantified over n. We then conduct a case analysis based on whether n is even. This results in two subgoals: one where n = 0 and another where n ⩾ 0 is an even number. The former case involves demonstrating the validity of WB $\lesssim$ WB' is omitted here due to space constraints. We proceed with the latter case, which is outlined below:

```
CIH : ∀ n : nat, ev n → r WB (merge_bp_contn "p" (bp_receivea "p" "l1" (I)) WB' n)
H   : ev n
______________________________________(1/1)
paco2 refinementR r WB (merge_bp_contn "p" (bp_receivea "p" "l1" (I)) WB' n.+2)
```

▶ Remark 11. To center the attention on actions and continuations, we will no longer use list notation, labels, or sorts in the rest of the proof text. This is because both sides of $\lesssim$ are made of streamline of actions (nested singleton lists), where all elements (labels and sorts) align. Just that we employ a dot to delineate prefixes from the infinite terms.

Unfolding WB and applying the rule ref_a transforms the goal into p!p!p!.WB $\lesssim$ $(\texttt{p?'})^{\texttt{n+1}}$.WB'. This term corresponds to the one given in second-to-last line on the right-hand side of the proof steps in Figure 3. Note that the rule application permits the discharge of the leftmost receive prefixes on both sides.

After unfolding WB' inside the goal, it takes the form p!p!p!.WB $\lesssim$ $(\texttt{p?'})^{\texttt{n+2}}$p!.WB'). We then apply ref_b with n ≜ n+2, resulting in p!p!.WB $\lesssim$ $(\texttt{p?'})^{\texttt{n+2}}$.WB'. Notice that this application effectively shifts the send action on the right to the leftmost position through reordering and cancels the leftmost send prefixes.

We keep unfolding WB' followed by the application of ref_b with n ≜ n+3 and n ≜ n+4 respectively and obtain the goal in the following shape: WB $\lesssim$ $(\texttt{p?'})^{\texttt{n+4}}$.WB' which could easily be closed by instantiating the coinduction hypothesis CIH with n ≜ n+4. Note also that we separately prove the action equalities after every single application of rules ref_a and ref_b employing the idea in Definition 8. ◀

4 Subtyping Negation

The negation of refinement relation $\not\lesssim$ over SISO trees structurally displays the shape of trees which cannot refine each other. It serves as a framework for the complement of subtyping within the context of session types thus session trees. Originally comprised of eighteen inductively stated rules as outlined in [22, Fig. 6], we are able to shrink the set by eliminating ten rules, and present the new set of rules in Figure 4. Completeness with respect to refinement is elaborated in § 5. Before delving into the specifics of the new set of rules, we need to revisit the way action equalities are handled.

4.1 Action Equalities, Refinement and Subtyping Revisited

We revisit the rationale behind membership and action equality checks outlined in § 2.3, and restate refinement and subtyping relations accordingly. We establish an inductive membership relation over streams of actions which is crucial for deriving useful lemmas regarding the structure of terms containing specific actions. For instance, Lemma 12 and 13 cannot be proven unless the membership check $\in$ is inductively defined. This is because it is impossible to infer term shapes from a coinductively defined membership relation. This can only be achieved through the induction schema for an inductively defined membership relation. These lemmata are key in proving completeness of refinement thus subtyping with respect to negations.

▶ **Lemma 12.** $\forall \mathsf{p}\ \mathsf{W}, \exists \mathcal{C}^{(\mathsf{p})}\ \ell\ \mathsf{S}\ \mathsf{W}', \quad \mathsf{p}? \in \mathsf{act}(\mathsf{W}) \implies \mathsf{W} = \mathcal{C}^{(\mathsf{p})}.\mathsf{p}?\ell(\mathsf{S}).\mathsf{W}'.$

where $\mathcal{C}^{(\mathsf{p})}$ is a sort of prefixing

$$\mathcal{C}^{(\mathsf{p})} \quad ::= \quad \varepsilon \ \mid\ \mathsf{r}!\ell(\mathsf{S}) \ \mid\ \mathsf{q}?\ell(\mathsf{S}) \ \mid\ \mathsf{r}!\ell(\mathsf{S}).\mathcal{C}^{(\mathsf{p})} \ \mid\ \mathsf{q}?\ell(\mathsf{S}).\mathcal{C}^{(\mathsf{p})} \quad (\mathsf{q} \neq \mathsf{p})$$

that allows all sends alongside all receives but not those from a particular participant p.

▶ **Lemma 13.** $\forall \mathsf{p}\ \mathsf{W}, \exists \mathcal{B}^{(\mathsf{p})}\ \ell\ \mathsf{S}\ \mathsf{W}', \quad \mathsf{p}! \in \mathsf{act}(\mathsf{W}) \implies \mathsf{W} = \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell(\mathsf{S}).\mathsf{W}'.$

Notice that $\mathcal{C}^{(\mathsf{p})}$ sort of prefixing amounts to the $\mathcal{A}^{(\mathsf{p})}$ sort in the absence of send actions.

▶ **Lemma 14.** $\forall \mathsf{p}\ \ \mathcal{C}^{(\mathsf{p})}\ \ \mathsf{W}, \quad \mathsf{p}! \notin \mathcal{C}^{(\mathsf{p})} \implies (\exists \mathcal{A}^{(\mathsf{p})}, \quad \mathcal{C}^{(\mathsf{p})}.\mathsf{W} = \mathcal{A}^{(\mathsf{p})}.\mathsf{W}).$

▶ **Definition 15.** *For a pair of SISO trees* W *and* W'*, we define action equality as follows:*

$$\forall a, \quad a \in \mathsf{act}(\mathsf{W}) \iff a \in \mathsf{act}(\mathsf{W}').$$

```
Definition act_eq (t t': st) ≜ ∀ a, coseqIn a (act t) ↔ coseqIn a (act t').
Definition act_neq (t t': st) ≜ ∃ a, coseqIn a (act t) ∧ (coseqIn a (act t') → False) ∨
                                     coseqIn a (act t') ∧ (coseqIn a (act t) → False).
```

We redefine the refinement relation by incorporating the action equality check described in Definition 15. This adjustment enables us to establish its completeness in regard to negations as discussed in § 4.2 and § 5.

```
Inductive refinementR2 (seq: st → st → Prop): st → st → Prop ≜
 | ref2_a: ∀ w w' p l s s' a n,
           subsort s' s → seq w (merge_ap_contn p a w' n)  →
           act_eq w (merge_ap_contn p a w' n) →
           refinementR2 seq (st_receive p [(l,s,w)]) (merge_ap_contn p a (st_receive p [(l,s',w')]) n)
 | ref2_b: ∀ w w' p l s s' b n,
           subsort s s'  → seq w (merge_bp_contn p b w' n) →
           act_eq w (merge_bp_contn p b w' n) →
           refinementR2 seq (st_send p [(l,s,w)]) (merge_bp_contn p b (st_send p [(l,s',w')]) n)
 | ref2_end: refinementR2 seq st_end st_end.
Definition refinement2: st → st → Prop ≜ fun s1 s2 ⇒ paco2 refinementR2 bot2 s1 s2.
Notation "x '~<' y" ≜ (refinement2 x y) (at level 50, left associativity).
```

The subyping relation therefore takes the following shape:

```
Definition subtype2 (T T': st): Prop ≜ ∃ (l: list (siso*siso)), decomposeL l T T' ∧ listSisoPRef2 l.
```

The sole difference between the functions `listSisoPRef` and `listSisoPRef2` is that the former employs the `refinement` relation while the latter makes use of the `refinement2` relation.

▶ Remark 16. In Coq, there is no way to bridge the gap between the action equality checks in Definitions 8 and 15. One potential avenue involves assuming that

$$\forall \mathsf{W}\ l,\ \mathtt{act}(\mathsf{W}) \in_C l \implies \mathtt{act}(\mathsf{W}) \in_C^I l \tag{1}$$

holds for the inductively defined version $\in_C^I$ of $\in_C$; and deducing that the statement in Definition 8 implies that in Definition 15. However, this approach would lead to inconsistency in Coq. It is because the predicate $\in_C^I$ forces its first argument to be finite whereas $\in_C$ can hold for some infinite stream. Therefore, implication 1 cannot be an instance of the *coinductive extensionality* (`cext`) principle. An example of the affirmative case is presented in [1, Appendix B], where `cext` effectively establishes the Leibniz equality from the bisimulation $=_{\mathbb{N}^{co}}$ over conats. This is because conats modulo $=_{\mathbb{N}^{co}}$ is isomorphic to $\mathbb{N} + 1$.

In our current context, such an isomorphism is not available. We are dealing with non-structural equality over streams of actions. Considering this, we decided to employ a pair of refinement (thus subtyping) relations that solely vary in their action equality checks. It is evident that they essentially serve the same purpose provided the `finiteness` assumption that "in a session with a potentially infinite number of interactions, there can only exist a finite number of distinct actions".

Note also that in the rest of the paper, we overload the symbol $\lesssim$ to denote the refinement relation (`refinement2`) based on the check given in Definition 15.

4.2 Negation of Refinement

The negation of the refinement relation is inductively defined as a counterpart of the coinductively given refinement relation. We revisit the set of rules originally stated in [22, Fig. 6] and list them in Figure 4. The rule [N-ACT] states that if a pair of trees do not have

$$\frac{\mathtt{act}(\mathsf{W}) \neq \mathtt{act}(\mathsf{W}')}{\mathsf{W} \not\lesssim \mathsf{W}'}\ [\text{N-ACT}] \qquad \frac{\mathsf{q}! \in \mathcal{C}^{(\mathsf{p})}}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{C}^{(\mathsf{p})}.\mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'}\ [\text{N-I-O-2}]$$

$$\frac{\ell \neq \ell'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{A}\text{-}\ell] \qquad \frac{\mathsf{S}' \not\leq: \mathsf{S}}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{A}\text{-S}] \qquad \frac{\mathsf{S}' \leq: \mathsf{S} \quad \mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{W}'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{A}\text{-W}]$$

$$\frac{\ell \neq \ell'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell'(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{B}\text{-}\ell] \qquad \frac{\mathsf{S} \not\leq: \mathsf{S}'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{B}\text{-S}] \qquad \frac{\mathsf{S} \leq: \mathsf{S}' \quad \mathsf{W} \not\lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{W}'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'}\ [\text{N-}\mathcal{B}\text{-W}]$$

Figure 4 The negation of the refinement relation $\not\lesssim$ over SISO trees.

the same set of actions (in terms of Definition 15) then they cannot refine each other. We managed prove in Coq that such kind of terms cannot be in the refinement relation thus they must be in the negation of the refinement; see Lemma 7. The rule [N-ACT] in fact proves four rules given in the original definition.

▶ **Lemma 17.** $\forall \mathsf{p}\ \ell\ \mathsf{S}\ \mathsf{W}\ \mathsf{W}'$,

$$\begin{array}{llll} (1)\ \mathsf{p}! \notin \mathtt{act}(\mathsf{W}') \implies \mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{W}' & (2)\ \mathsf{p}? \notin \mathtt{act}(\mathsf{W}') \implies \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{W}' \\ (3)\ \mathsf{p}! \notin \mathtt{act}(\mathsf{W}) \implies \mathsf{W} \not\lesssim \mathsf{p}!\ell(\mathsf{S}).\mathsf{W}' & (4)\ \mathsf{p}? \notin \mathtt{act}(\mathsf{W}) \implies \mathsf{W} \not\lesssim \mathsf{p}?\ell(\mathsf{S}).\mathsf{W}' \end{array}$$

The rule [N-I-O-2] states that within a pair of terms, if the left term starts with a receive action from a fixed participant p, it cannot refine the right term if the latter contains an arbitrary send action occurring before a receive action from the participant p. This restriction arises from the inability to reorder the actions of the right term such that a p? action moves to the beginning and becomes the leftmost action.

Another crucial aspect of this rule is the renovation of its shape, compared to the one in the original definition:

$$\text{original definition} \qquad\qquad\qquad \text{reformulated shape}$$

$$\frac{}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'} \quad \Longrightarrow \quad \frac{\mathsf{q}! \in \mathcal{C}^{(\mathsf{p})}}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{C}^{(\mathsf{p})}.\mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'}$$

This renovation is advantageous because the rule now adopts a similar structural shape to the other rules. It becomes more readily applicable since the right-hand side exhibits the general structural form of terms with receive actions. This connection is shown in Lemma 12. Also, Lemma 14 becomes applicable when the rule premise fails to be satisfied. Moreover, it makes one of the rules in the original definition ([N-I-O-1]) provable with the help of [N-ACT].

▶ **Lemma 18** ([N-I-O-1]). $\forall \mathsf{p}\ \mathsf{q}\ \ell\ \ell'\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}', \quad \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'$.

The last six rules ensure subtle cases involving asynchronous reorderings. The rule [N-$\mathcal{A}$-ℓ] claims that terms with mismatching labels cannot refine each other even under $\mathcal{A}^{(\mathsf{p})}$ kind of reordering; [N-$\mathcal{A}$-§] and [N-$\mathcal{A}$-W] are variants where sorts and continuations mismatch. In the last line, we have similar kind of rules this time for $\mathcal{B}^{(\mathsf{p})}$ style reorderings.

▶ Remark 19. We can prove six more rules from the original definition of negation relation simply by allowing inductive prefixes $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ to contain the empty prefix ε. We suffice to state in Lemma 20 only those related with [N-$\mathcal{A}$-ℓ] and [N-$\mathcal{B}$-ℓ].

▶ **Lemma 20.** $\forall \mathsf{p}\ \ell\ \ell'\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}'$,
(1) $\ell \neq \ell' \implies \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'$ (2) $\ell \neq \ell' \implies \mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}!\ell'(\mathsf{S}').\mathsf{W}'$

The negation relation is represented by a standard inductive type called `nRefinement`.

```
Inductive nRefinement: siso → siso → Prop ≜
  | n_act  : ∀ w w', act_neq (@und w) (@und w') → nRefinement w w'
  | n_i_o_2: ∀ w w' p l l' s s' c P Q, isInCp p c = true →
             nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) P)
                         (mk_siso (merge_cp_cont p c (st_receive p [(l',s',(@und w'))])) Q)
  | n_a_l  : ∀ w w' p l l' s s' a n P Q, l ≠ l' →
             nRefinement (mk_siso (p?'[(l,s,(@und w))]) P)
                         (mk_siso (merge_ap_contn p a (p?'[(l',s',(@und w'))]) n) Q)
  | n_a_s  : ∀ w w' p l s s' a n P Q, nsubsort s' s →
             nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) P)
                         (mk_siso (merge_ap_contn p a (st_receive p [(l,s',(@und w'))]) n) Q)
  | n_a_w  : ∀ w w' p l s s' a n P Q R, subsort s' s →
             nRefinement w (mk_siso (merge_ap_contn p a (@und w') n) P) →
             nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) Q)
                         (mk_siso (merge_ap_contn p a (st_receive p [(l,s',(@und w'))]) n) R)
  | n_b_s  : ∀ w w' p l s s' b n P Q, nsubsort s s' →
             nRefinement (mk_siso (st_send p [(l,s,(@und w))]) P)
                         (mk_siso (merge_bp_contn p b (st_send p [(l,s',(@und w'))]) n) Q)
  | ...
```

The function `merge_cp_cont` takes a participant `p`, an instance `c` of $\mathcal{C}^{(\mathsf{p})}$ and a session tree `w` and prefixes the actions of `c` to the tree `w`. The relation involves constructors `n_b_l` and `n_b_w`, omitted in the snippet, serving as alternatives to rules `n_a_l` and `n_a_w` respectively, with $\mathcal{B}^{(\mathsf{p})}$ sort of prefixing. We develop the relation in Coq over SISO trees, therefore the proof obligation `singleton` needs to be satisfied each time a session tree is used in this context. The parameters `P`, `Q` and `R` denote instances of such proofs.

With all the essential components in place, we are now equipped to define the negation of the subtyping relation, for which the refinement negation $\not\lesssim$ serves as the basis.

▶ **Definition 21** (negation of subtyping). *For any pair of session trees* T, T',

$$\mathsf{T} \not\leqslant \mathsf{T}' \triangleq \forall i \in I \ \ \forall (\mathsf{W}_i, \mathsf{W}'_i) \quad (\mathsf{W}_i \leftsquigarrow \mathsf{T}) \implies (\mathsf{W}'_i \leftsquigarrow \mathsf{T}') \implies \mathsf{W}_i \not\lesssim \mathsf{W}'_i.$$

5 Completeness

Completeness serves as the primary meta property of the subtyping relation (with respect to negation) that we successfully formulated and verified in Coq. In essence, it asserts that for any given pair of session trees T and T', T is either a subtype of T' or it is linked to T' by the negation of the subtyping relation, leaving no room for a third possibility. The subtyping completeness proof relies on the completeness of the revisited refinement relation (§ 4.1), as formally delineated below.

▶ **Lemma 22** (refinement completeness). *For any pair of SISO trees* W, W'*, we have*

$$\neg(\mathsf{W} \lesssim \mathsf{W}') \iff \mathsf{W} \not\lesssim \mathsf{W}'.$$

Proof. ($\Rightarrow$) To establish the left-to-right implication, we initially prove $\neg(\mathsf{W} \not\lesssim \mathsf{W}') \implies \mathsf{W} \lesssim \mathsf{W}'$, followed by deducing its contrapositive. This choice is motivated by the observation that in Coq proofs, the presence of the negation of a coinductively defined term within the goal context lacks utility, as its inversion fails to produce useful equations.

```
Lemma nRefLH: ∀ w w', (nRefinement w w' → False) → refinement2 (@und w) (@und w').
```

The proof proceeds by storing the proof state in a coinduction hypothesis `CIH` following the decomposition of `w` and `w'` into pairs of their respective underlying session trees and proofs confirming that they are singletons, namely into `(w, Pw)` and `(w', Pw')`.

```
CIH: ∀ (w': st) (Pw' singleton w') (w: st) (Pw: singleton w),
(nRefinement {|und≜ w; sprop≜ Pw|} {|und≜ w'; sprop≜ Pw'|} → False) → r w w'
```

`CIH` is parametrised by the binary relation `r` over session trees which signifies the accumulated knowledge derived from coinductive foldings of the refinement relation. The rest relies on the inversion lemma `sinv` over SISO trees which discusses the possible shapes they could exhibit: a SISO tree is a streamline of actions that initiates with a send or receive action, or it is simply an end.

```
Lemma sinv: ∀ w, singleton w →
                  (∃ p l s w', w = st_send p [(l,s,w')] ∧ singleton w') ∨
                  (∃ p l s w', w = st_receive p [(l,s,w')] ∧ singleton w') ∨ (w = st_end).
```

Therefore considering the potential shapes of `w` and `w'`, the left-to-right proof is made of nine distinct cases. Here we focus on the one where both of the trees start with receive actions such that `w1 = (p?'[(l,s,w1)])` and `w2 = (q?'[(l',s',w2)])` for some `p`, `q`, `l`, `l'`, `s`, `s'`, `w1` and `w2` such that `w1` and `w2` are indeed singleton trees.

1. We have a case distinction on the fact that `p?' ∈ act(w')`. If `w'` does not contain the `p?'` action, the goal is a trivial application of the rule `n_act`. Otherwise, we get `w' = merge_cp_cont p c (p?'[(l1,s1,w3)])` thanks to Lemma 12 for some prefix `c`, label `l1`, sort `s1` and term `w3`.

   We then apply a further case analysis on `p! ∈ c`. The positive case is a direct implication of the rule `n_i_o_2`. In the negative case, `w'` takes the shape of `merge_ap_cont p a (p?'[(l1,s1,w3)])` for some prefix `a` due to Lemma 14, transforming the goal into `paco2 refinementR2 r (p?'[(l,s,w1)]) (merge_ap_cont p a (p?'[(l1,s1,w3)]))` which is solved by case distinctions described in below items 2 to 4.
2. When `l = l'`, `s'` is a subsort of `s` and `w1` and `(merge_ap_cont p a w3)` are of the same actions, we apply the constructor `ref_a` with the prefix `a ≜ a` which entails a subgoal `upaco2 refinementR2 r w1 (merge_ap_cont p a w3)`.

To close the subgoal, we do not further fold the coinductive relation `refinementR2`, instead employ the coinduction hypothesis `CIH`. Then, the objective is to show that `nRefinement w1 (merge_ap_cont p a w3) → False` holds under the initial assumption `nRefinement w w' → False`. This is addressed by the rule `n_a_w` with `a` $\triangleq$ `a`.

3. In case `l = l'`, `s'` is a subsort of `s` and `w1` and `(merge_ap_cont p a w3)` are of the different actions, we can deduce that `nRefinement w w'` thanks to the rule `n_act`. This contradicts with the initial assumption of `nRefinement w w' → False` and closes the case.
4. The cases where `l` $\neq$ `l'` or `s'` is not a subsort of `s` are handled by rules `n_a_l` and `n_a_s`. ◀

Proof. (⇐) . The right-to-left implication reflects into Coq as follows.

```
Lemma nRefR: ∀ w w', nRefinement w w' → (refinement2 (@und w) (@und w') → False).
```

The proof argues by structural induction over the negation relation and is made of eight cases. Here, we present a selected case associated to the rule `n_b_s`. The refinement assumption in this case is of the shape: `H: refinementR2 (upaco2 refinementR2 bot2) (p![(l,s,w)]) (merge_bp_contn p b (p![(l,s',w')]) n)` for some `n`, `w`, `w'`, `s` and `s'` such that `s` is not a subsort of `s'`. Inverting `H` results in proving `False` provided following equations.

1. `p![(l,s'0,w'0)] = merge_bp_contn p b (p![(l,s',w')]) n` for some `w'0`, `s'0` such that `w` refines `w'0` and `s` is a subsort of `s'0`;
2. `merge_bp_contn p b0 (p![(l,s'0,w'0)]) n0 = merge_bp_contn p b (p![(l,s',w')]) n` for some `n0`, `b0`, `w'0`, `s'0` such that `w` refines `merge_bp_contn p b0 w'0 n0`, `s` is a subsort of `s'0` and `w` contains the same actions with `merge_bp_contn p b0 w'0 n0`.

The initial falsity is demonstrated through a case analysis over `n` (number of times the prefix is repeated) and subsequent case analysis over `b` (the prefix) when `n` $\geqslant$ `0`. Each of these cases is resolved by establishing contradictions within the context: either an equality between a send and a receive action, a dis-equality between the same actions, or hypotheses asserting both that `s` is a subsort of `s'` and its negation at the same time.

The proof of the second falsity is somewhat more intricate and relies on the meqBp lemma provided below. This lemma establishes the structural equality between *merging a term once with a single sequence of actions captured after n iterations of appending a given prefix with itself* and *merging the term with the given prefix n times*.

The function `BpnA` handles the appending of a given prefix `b` to itself `n` times and constructs a sequence of actions from it. And, the function `merge_bp_cont` is a variant of `merge_bp_contn` with `n` set to 1.

```
Lemma meqBp: ∀ n p b w, merge_bp_cont p (BpnA p b n) w = merge_bp_contn p b w n.
```

We rewrite the lemma meqBp in the hypothesis and transform it into `merge_bp_cont p (Bpn p b0 n0) (p![(l,s'0,w'0)]) = merge_bp_cont p (Bpn p b n) (p![(l,s',w')])`. It is now possible to infer that `(p![(l,s'0,w'0)]) = (p![(l,s',w')])`, hence `s'0 = s'` and `w'0 = w'`, thanks to the lemma BpBpeqInv2 .

```
Lemma BpBpeqInv2: ∀ p b1 b2 l1 l2 s1 s2 w1 w2,
  merge_bp_cont p b1 (p![(l1,s1,w1)]) = merge_bp_cont p b2 (p![(l2,s2,w2)]) →
  (p![(l1,s1,w1)]) = (p![(l2,s2,w2)]).
```

We can now close the goal simply by plugging `s0' = s'` in. This equation leads to inconsistency in the proof context as we then obtain proofs of "`s` *is not* a subsort of `s'`" and "`s` *is* a subsort of `s'`" at the same time. ◀

▶ **Corollary 23** (subtyping completeness). *For any pair session trees* T, T'*, we have*

$$\neg(\mathsf{T} \leqslant \mathsf{T}') \iff \mathsf{T} \nleqslant \mathsf{T}'.$$

Proof. Follows from Lemma 22. ◀

```
Lemma subNeqL: ∀ T T', (subtype2 T T' → False) → nsubtype T T'.
Lemma subNeqR: ∀ T T', nsubtype T T' → (subtype2 T T' → False).
Theorem completeness: ∀ T T', (subtype2 T T' → False) ↔ nsubtype T T'.
Proof. split; [ apply (subNeqL T T') | intros. apply (subNeqR T T'); easy ]. Qed.
```

Axiomatic Base and Mechanisation Effort. In the accompanying library, we employ classical reasoning to conduct case analysis primarily over coinductively defined predicates. We also use the proof irrelevance axiom to obtain that different proofs of dis-equality among the same pair of participants are treated the same. The library comprises around 10K lines of code, containing 250 proven lemmata and 166 definitions, with 35 of them being coinductively stated. Initially, integrating inductive and coinductive reasoning seemed challenging, but it scaled remarkably well with the aid of the `paco` library.

6 Related Work and Conclusion

Asynchronous session subtyping was first introduced to achieve message optimization in session-based high-performance computing platforms, i.e., multicore C programming [28, 42] and MPI-C [34, 33]. Then, numerous theoretical and practical advancements have emerged.

In theory, Chen et al. [11] introduced and proved *preciseness* of synchronous [18, 15] and asynchronous subtyping for the binary (2-party) session types. Later, the asynchronous subtyping was found undecidable, independently by Bravetti et al. [6], and by Lange and Yoshida [32]. This provoked active studies on (1) identifying a set of binary session types where asynchronous subtyping is decidable [7, 32]; and (2) proposing *sound* algorithms extending the formalism to *binary* communicating automata [4] in [5, 2] (also to fair refinement in [8]). In the multiparty setting, Ghiezan et al. [20, 21, 22] proposed precise synchronous and asynchronous session subtyping employing coinductive axiomatisation.

In practice, Castro-Perez and Yoshida [10] examined a constrained version of multiparty asynchronous subtyping algorithm where permutations across unrolling recursions are prohibited. This framework has been used for the cost analysis of optimised C code. Cutner et al. [14] proposed a sound multiparty synchronous subtyping algorithm and integrated it into Rust. Neither of the multiparty algorithms in [10] and [14] nor the one for binary sessions types in [5] can validate [22, Example 3.19]. In a recent study [2, Figure 6], Bocchi et al. presented an extended version of the algorithm proposed in [5], incorporating program analysis techniques. They effectively validated the example, albeit the algorithm is limited to the binary setting. We mechanised and proved this example in Coq (Figure 5) within a multiparty setting.

The mechanisation approach we employ is not bounded by the undecidability of asynchronous subtyping, as subtyping is axiomatised as a coinductive relation in Coq. It is non-computational. The key point we make in Figure 5 is to show that our subtyping technique and its implementation in Coq are expressive enough to cover several examples that have been proven using different automated tools.

	[5]	[2]	[10]	[14]	Ours
`ring-choice` [13]	✗	✗	✓	✓	✓
Example 3.17 [22]	✗	✗	✓	✓	✓
Example 3.19 [22]	✗	✓	✗	✗	✓
Example 4.14 [22]	✗	✗	✗	✓	✓

Figure 5 Examples and related work.

Mechanisation recently emerges as a pivotal facet in the concurrent communication models. Tirore et al. [40] introduced a novel projection function that maps global multiparty session types to local types. This function has been demonstrated to be both sound and complete with respect to its coinductive counterpart. It has been implemented in Coq and its mentioned properties have been formally proven there. Castro-Perez et al. [9] built a domain-specific language named Zooid, and implemented in Coq. Zooid allows for the extraction of certified synchronously interacting programs built upon MPST. Brady [3] devised secure communication protocols for binary sessions in Idris. Thiemann et al. [39] formalised progress and preservation properties for binary session types in Agda. Tassarotti et al. [37] developed a compiler grounded in a simplified version of the GV system [19] for a functional language equipped with binary session types. The correctness of this compiler has been proven in Coq. Jacobs et al. [30] expanded a functional language with multiparty session types (MPGV) and formally verified, in Coq, that the language is deadlock-free. Hinrichsen et al. [25, 23, 24] introduced Actris, a Coq tool that integrates separation logics and asynchronous binary session types with an asynchronous subtyping mechanism. Actris is developed as an extension to the Iris program logic. Jacobs et al. extended Actris by linear logic into LinearActris in their recent work [31] to freely obtain deadlock and leak freedom for binary session types from linearity. Choreographic programming paradigm allows one to implement distributed programs as single programs, ensuring coherence between send and receive operations by consolidating them into a single construct. Deadlock freedom is inherit in the design. Cruz-Filipe et al. [12] formalised the theory of choreographic programming in Coq. In their work [26], Hirsch and Garg introduced Pirouette, a choreographic language designed with formal guarantees, which are rigorously verified in Coq. Similarly, in [35], Pohjola et al. presented Kalas, a compiler for a choreographic language correctness of which has been verified using the HOL4 theorem prover.

Conclusion and Future Work. In this paper, we present the first formalisation of asynchronous subtyping for session trees, establishing a framework for asynchronous subtyping in MPST. The formalisation (1) decomposes arbitrary session trees into SISO trees that are free of choice and selection, and (2) governs the subtyping relation through *refinement* of these trees. It has been used to certify four illustrative protocol optimisation examples presented in the literature. See Figure 5.

In the development, we redefined the negation of the refinement relation, addressing the incompleteness issue spotted in the prior work [22], and proved that subtyping is complete with respect to the renovated negation. To determine the precise configuration for refinement and its negation, we employed a new sort of term prefixing.

Our future plan includes establishing a Coq proof of the soundness of subtyping with respect to *liveness*, a behavioural property of typing environments ensuring that every pending send eventually enqueues messages and every pending reception is eventually executed.

References

1. Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. Inductive reasoning for coinductive types. *CoRR*, abs/2301.09802, 2023. doi:10.48550/arXiv.2301.09802.
2. Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.
3. Edwin C. Brady. Type-driven development of concurrent communicating systems. *Comput. Sci.*, 18(3), 2017. doi:10.7494/CSCI.2017.18.3.1413.
4. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
5. Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science*, Volume 17, Issue 1, March 2021. doi:10.23638/LMCS-17(1:20)2021.
6. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
7. Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.
8. Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair refinement for asynchronous session types. In *FoSSaCS*, Lecture Notes in Computer Science, 2021.
9. 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.
10. David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020. doi:10.1145/3428223.
11. Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *LMCS*, 13:1–62, 2017.
12. Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.
13. Zak Cutner and Nobuko Yoshida. Safe Session-Based Asynchronous Coordination in Rust. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2021. doi:10.1007/978-3-030-78142-2_5.
14. 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.
15. Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *22nd International Conference on Concurrency Theory*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.
16. Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*,

volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:10.1007/978-3-642-32940-1_20.

17 Burak Ekici and Nobuko Yoshida. https://github.com/ekiciburak/sessionTreeST/tree/localtype. Software, swhId: swh:1:dir:33823a0054801bcf4ea95f2dffe733579cbd53c8 (visited on 2024-08-20). URL: https://github.com/ekiciburak/sessionTreeST/tree/itp2024.

18 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

19 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.

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

21 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, January 2021.

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

23 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.

24 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. doi:10.46298/LMCS-18(2:16)2022.

25 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.

26 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.

27 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. doi:10.1007/BFb0053567.

28 Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Type-Directed Compilation for Multicore Programming. *ENTCS*, 241:101–111, 2009.

29 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.

30 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.

31 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.

32 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.

33 Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

34 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *50th International Conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

35 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. `doi:10.4230/LIPICS.ITP.2022.27`.

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

37 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`.

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

39 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`.

40 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. `doi:10.4230/LIPICS.ITP.2023.28`.

41 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`.

42 Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. Session-based compilation framework for multicore programming. In *FMCO 2008*, pages 226–246, 2008. `doi:10.1007/978-3-642-04167-9_12`.

43 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`.