Formalising Asynchronous Session Subtyping

BURAK EKICI, University of Oxford, UK
NOBUKO YOSHIDA, University of Oxford, UK

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 order of messages sent (resp. received) to (resp. from) particular participant is preserved and sending is non-blocking. The optimisation is achieved by reordering send actions with any other action, except sends to the same participant, and by reordering receive actions with other receive actions, except those from the same participant.

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 presents the first formalisation of asynchronous subtyping for multiparty session types within the Coq proof assistant. We begin by translating session types into *session trees*, unfolding recursion coinductively. These trees are then decomposed into new tree forms that incorporate singleton branching and/or selection constructs. On these trees, which involve both singleton branching and selections, we define action reorderings within a coinductive refinement relation that governs subtyping.

To demonstrate the expressiveness of our formalisation, we verify several subtyping schemas drawn from the literature—none of which can be simultaneously validated by existing decidable but 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.

We establish the correctness of the refinement relation, in Coq, showing that it preserves the ordering of send (resp. receive) actions to (resp. from) a specific participant. Additionally, we formally demonstrate in Coq that refinement is transitive, a property crucial for closing certain cases in subtyping proofs.

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 32K lines of Coq code, and is available on GitHub at https://github.com/ekiciburak/async-mpst-st/tree/acm and on Zenodo at https://doi.org/10.5281/zenodo.18268293.

CCS Concepts: • **Theory of computation** → **Logic**; **Type theory**; *Concurrency*; • **Software and its engineering** → *Formal methods*.

Additional Key Words and Phrases: asynchronous multiparty session types, session trees, subtyping, formal logic

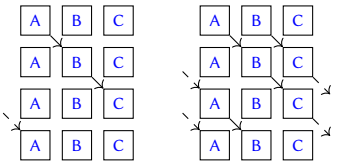
1 INTRODUCTION

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 [28, 45], originally devised to two-party protocols like client-server interactions and later expanded to handle multiparty protocols as well [21, 51]. 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* [16, 18], denoted by $\mathbb{T}' \leqslant \mathbb{T}$, when $\mathbb{T}'$ is a subtype of some type $\mathbb{T}$.

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 start the next iteration by sending data to A.

The synchronous interactions from B's local viewpoint could be represented by a *session type* $\mathbb{T}_B$, which can then be optimised into the type $\mathbb{T}_B^{\text{opt}}$ under asynchronous interactions as specified in Figure 1. The notation "!" is read as "send to", while "?" denotes 'receive from" actions, and "i32" is

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

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

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

The optimisation illustrated in Figure 1 is achieved by simply reordering the "send to C" and "receive from A" actions. The type of the optimised interactions, $\mathbb{T}_B^{\text{opt}}$, is considered an *asynchronous subtype* [21, 22] of the type $\mathbb{T}_B$. Any process of type $\mathbb{T}_B^{\text{opt}}$ can be safely replaced by a process of type $\mathbb{T}_B$ within the protocol, while preserving deadlock-freedom.

The asynchronous nature of communication is not immediately visible at the level of type syntax. It becomes apparent in the process language, where send actions are non-blocking due to the use of message queues. It is also manifested in the subtyping relation, where certain actions may be safely reordered. In this paper, we focus on the latter and formalise several properties of subtyping for an asynchronous setting of multiparty session types in the proof assistant Coq.

Note 1. *Throughout the paper, we use the symbol* 🐓 *to hyperlink to the corresponding Coq sources, and mark excerpts from the Coq development using* `highlighted` *text.*

Contributions

(1) We provide the first Coq [47] library
  on GitHub at https://github.com/ekiciburak/async-mpst-st/tree/acm
  on Zenodo at https://doi.org/10.5281/zenodo.18268293

that handles the internal dynamics of asynchronous subtyping for MPST [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 [2, 5, 10, 14] can verify examples all together (see § 7).

(2) We prove a completeness theorem of subtyping (Theorem 5.2) with respect to its negation. The Coq proof involves reorganising the subtyping relation by reformulating the underlying refinement relation and its negation, proposed by Ghilezan et al. [21, 22] .
   (a) 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 3.5). This simplification facilitates the proof of an inversion lemma, as elaborated in Remark 5 and Lemma 3.6.
   (b) 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 ($C^{(\mathsf{p})}$ – Lemma 3.7) 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 4.2, 3.9, 4.3 and 4.4, and in Remark 7.

(3) We prove in Coq the correctness of the refinement relation (Theorem 3.25): it ensures that send (resp., receive) actions to (resp., from) a particular participant cannot be reordered.

(4) We also prove in Coq that the refinement relation is transitive (Theorem 6.5), a property that is useful in subtyping proofs.

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 3.5) to show that the latter is a subtype of the former.

As part of the future directions, we plan to implement the full type system for asynchronous MPST with subtyping and prove properties such as fairness and liveness exploiting the structures and proofs existing in the current Coq code.

Changes wrt. conference version

This article revises and extends the earlier conference version of this work [17], incorporating additional material and mechanised proofs in Coq. In particular, we provide Coq proofs for the correctness and transitivity of the *refinement* relation (§ 3.2), which are presented in the newly added sections § 3.4 and § 6.2, respectively.

To establish correctness, we first define a new refinement relation that is equivalent to the original one, prove the property for this new relation, and then transport the result via the equivalence. The original refinement relation is based on a dependently typed implementation of $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ *prefixes* (Definition 3.4), which inherently include proofs of "participant dis equality". This design introduces difficulties when converting between prefixes. To address this, we develop a non-dependently typed implementation of prefixing for the new relation, and carry out the necessary proofs in this framework.

The transitivity proof follows a similar approach: we use the non-dependently typed prefixing to complete the proof and then exploit the equivalence to derive the result for the original relation.

Both the correctness and transitivity proofs in Coq are technically challenging. The inversion lemmas arising from instances of the refinement relation are particularly intricate due to the potential for action reordering introduced by asynchronous communication. These proofs are nearly as complex as the completeness proof (which was presented solely in the previous version [17]), collectively amounting to around 21K lines of Coq code and effectively tripling the size of the library.

We also complete the Coq proof of the optimisation in the ring choice example (see Figure 1), which was unintentionally left incomplete in the previous version [17]. § 6.3 provides a detailed account of this proof and discusses the previously omitted case, which contributes approximately 1K additional lines of Coq code.

We present the full proof of Example 3.19 in [22] in § 6.4, explicitly identifying the proof state where the transitivity argument is invoked—previously concluded by assuming transitivity.

We lift the subtyping schema from the level of trees to the level of types in the Coq implementation, providing a translation mechanism (Definition 3.1). This mechanism is implemented as a corecursive function to circumvent certain form of difficulties raised by Coq (Remark 3).

2 MULTIPARTY SESSION TYPES

In this section, we introduce standard MPST syntax inductively:

$$\mathtt{S} \;::=\; \mathtt{nat} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{unit} \qquad \mathbb{T} \;::=\; \mathtt{end} \mid \&_{i\in I}\, \mathtt{p}?\ell_i(\mathtt{S}_i).\mathbb{T}_i \mid \bigoplus_{i\in I} \mathtt{p}!\ell_i(\mathtt{S}_i).\mathbb{T}_i \mid \mu\mathbf{t}.\mathbb{T} \mid \mathbf{t}$$

Sorts represent the types of values—such as `nat` for natural numbers, `int` for integers, `bool` for booleans, `unit` for the singleton sort. A session type $\mathbb{T}$ specifies how a participant behaves within a multiparty session.

The external choice, written as $\&_{i\in I}\, \mathtt{p}?\ell_i(\mathtt{S}_i).\mathbb{T}_i$, models a participant waiting to receive a message from another participant p. The message will carry a label ℓ_i and a payload of sort $\mathtt{S}_i$, and depending on which $i \in I$ is chosen, the interaction proceeds as described by $\mathbb{T}_i$. Conversely, the internal choice is expressed as $\bigoplus_{i\in I} \mathtt{p}!\ell_i(\mathtt{S}_i).\mathbb{T}_i$, and represents the act of sending a message to participant p, where one of the possible labels ℓ_i is chosen and the corresponding value of sort $\mathtt{S}_i$ is transmitted; interaction then continues according to the associated continuation $\mathbb{T}_i$. Recursive types are introduced via $\mu\mathbf{t}.\mathbb{T}$, where $\mathbf{t}$ is a recursion variable bound within $\mathbb{T}$. The recursion is contractive: in any recursive type $\mu\mathbf{t}.\mathbb{T}$, it is required that $\mathbb{T}$ is not merely a recursion variable. This ensures that recursive calls are nested under some form of communication action (e.g., input or output). The type `end` indicates that the participant has completed all interactions.

A session type is said to be closed if every recursion variable it contains is properly bound within a corresponding μ binder.

We introduce a Coq library that formalises asynchronous session types with an inductive syntax, their coinductive tree representations, the notion of subtyping over trees, and various property proofs. The syntax for sorts and types is folklore (🐓, 🐓):

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

```
Inductive local: Type ≜
  | lt_end    : local
  | lt_receive: participant → list (label*sort*local) → local
  | lt_send   : participant → list (label*sort*local) → local
  | lt_mu     : local → local
  | lt_var    : nat → local.
```

Participants and labels are represented using names (i.e., `String` instances). Continuations in branching and selection constructs are encoded using lists, since the number of choices is finite. Recursion variables are encoded using de Bruijn indices. In doing so, we rely on the `autosubst2` [43, 44] library, which automatically generates the substitution function along with a suite of useful properties. On top of this foundation, we implement several basic functions 🐓:

```
Definition unf l ≜ if l is lt_mu l' then l' [l .: lt_var]  else l.
Definition full_unf l ≜ (iter (depth l) unf l).
```

Here, the notation `.:` represents the substitution function, and the term `l' [l .: lt_var]` expresses the result of replacing all occurrences of the recursion variable in `l'` with the type `l`. The `full_unf` function fully unfolds recursive types by applying `unf` iteratively using `iter` on the input type `l`; this unfolding proceeds up to the type's `depth`, defined as the number of leading `lt_mu` binders. The full unfolding is used in the translation of types into their corresponding trees (Definition 3.1).

3 SESSION TREES: DECOMPOSITION AND REFINEMENT

The notion of subtyping within the scope 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].

Given that the language of asynchronous MPST enables repeating protocols through the recursive μ-binder, potentially infinite unfoldings of μ can be interpreted as infinite tree structures: each (closed) session type $\mathbb{T}$ corresponds to a session tree $\mathbb{T} \xrightarrow{\mathcal{T}} \mathsf{T}$ (see Definition 3.1). The subtyping relation is then coinductively defined over the tree representations of these types, offering greater flexibility by abstracting away the challenges of recursion.

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

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

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

Session trees are used to interpret session types; thus, the meaning of the constructors remains implicit and unchanged. They are simply stated coinductively to account for infinite behaviour. The constructor $\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i$ represents *branching* (or *external choice*) interactions: it captures a set of incoming messages from participant p, each carrying a label ℓ_i, a payload of sort S_i, and continuing as T_i. In contrast, $\bigoplus_{i\in I}\, \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i$ denotes *selection* (or *internal choice*), describing a set of outgoing messages to participant p, each with a label ℓ_i, a payload of sort S_i, and a continuation T_i (for some $i \in I$). The constructor `end` indicates that no further interactions follow, marking the termination point.

In the Coq implementation of session trees (`st`), we opted to represent the continuations of communication actions—selections and branchings—using infinite lists (`coseq`), even though such continuations are, by construction, finite within the scope of well-formed session types thus trees. This decision was primarily motivated by the need to define the translation from types to trees (`lt2st` below) as a *corecursive function* in Coq. Using a finite data structure at this point would prevent a corecursive definition, as the resulting function would not pass Coq's guardedness (productivity) check. This issue is discussed in detail in Remark 3.

Note 2. *In the code above 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 omit the symbols* $\&$ *and* $\bigoplus$*, instead retaining their functionality by encoding selections and branchings in Coq as colists (i.e.,* `coseq`*) of label–sort–continuation triples.*

Definition 3.1 (types → trees). The translation from types to trees is governed by the relation $\mathbb{T} \xrightarrow{\mathcal{T}} \mathsf{T}$, defined using the following coinductive rules:

$$\frac{\mathbb{T}[\mu \mathbf{t}.\mathbb{T}/\mathbf{t}] \xrightarrow{\mathcal{T}} \mathsf{T}}{\mu \mathbf{t}.\mathbb{T} \xrightarrow{\mathcal{T}} \mathsf{T}}\ [\textsc{rec}] \qquad \frac{}{\mathsf{end} \xrightarrow{\mathcal{T}} \mathsf{end}}\ [\textsc{end}]$$

$$\frac{\forall i \in I, \quad \mathbb{T}_i \xrightarrow{\mathcal{T}} \mathsf{T}_i}{\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i \xrightarrow{\mathcal{T}} \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\textsc{snd}] \qquad \frac{\forall i \in I, \quad \mathbb{T}_i \xrightarrow{\mathcal{T}} \mathsf{T}_i}{\bigoplus_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i \xrightarrow{\mathcal{T}} \bigoplus_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\textsc{rcv}]$$

```
CoFixpoint lt2st (l: local): st ≜
  match full_unf l with
    | lt_receive p xs ⇒
      let cofix next xs ≜ match xs with
                          | (l1,s1,t1)::ys ⇒ cocons (l1,s1,lt2st t1) (next ys)
                          | nil            ⇒ conil
                          end
      in st_receive p (next xs)
    | ...
  end.
```

We implement the translation as a cofixpoint `lt2st` in Coq: it takes a type, unfolds all outermost μ binders using the function `full_unf`, and then translates each type in the external and internal choices coinductively; end is translated to end.

Remark 3. *Note that we could have used finite lists or other finite data structures to represent* `st` *continuations. However, doing so would expose a fundamental limitation imposed by Coq's guardedness checker: the inner recursive function (namely* `next` *inside* `lt2st`*) would fail to satisfy the productivity requirement. In particular, the recursion would not be syntactically guarded by a coinductive constructor and would therefore be rejected by the kernel.*

An alternative approach would be to formalise branching and selection employing a finite data structure and define the translation as a corecursive relation. However, this gives rise to a different class of difficulties, particularly when dealing with proofs involving existential quantification. The typical inductive proof strategy in such cases involves (1) applying the induction hypothesis to obtain an intermediate witness, and (2) processing this witness to construct the final value, after which the `exists` *tactic is applied. This workflow, however, is fundamentally incompatible with the requirements of coinductive proofs: the coinduction hypothesis must be guarded immediately by a constructor. Since existential quantifiers are inductively defined, statements involving them cannot themselves be directly used as coinductive hypotheses. Consequently, we may be unable to register the current proof state as the coinduction hypothesis until an existential witness has been constructed. This intermediate dependency often makes the proof non-scalable or unmanageable.*

Implementing the translation as a corecursive function avoids this issue entirely as it directly computes the desired value. This not only enables a more straightforward mechanisation of the translation but also simplifies existence proofs that would otherwise require assumptions (e.g., constructive indefinite description) to accommodate the restrictions of guarded corecursion.

The main goal of this section is to introduce the *refinement* relation that lies at the heart of session tree subtyping (or session sub-treeing). The subtyping relation itself is then presented in detail in § 4. To this end, we first present session tree decompositions in § 3.1 into *linear sequences of actions*. This decomposition allows us to align the actions of a pair of trees and to determine whether they are indeed *reordered* in a way that satisfies the refinement relation, as detailed in

§ 3.2. We then devote § 3.3 to a Coq formalisation of the refinement relation. Finally, in § 3.4, we establish the formal correctness of refinement in Coq, showing that it permits only those action reorderings that are deemed legal in § 3.2, disallowing all others.

3.1 SI, SO and SISO Trees

A session tree is called *single-input-single-output* (SISO) if all branches and selections are singletons, i.e., the tree corresponds to a linear sequence of actions.

SISO trees play a central role in session-tree subtyping, as checking "whether a session tree T qualifies as a subtree (subtype) of another tree T' ($\mathsf{T} \leqslant \mathsf{T}'$)" is twofold:

(1) *Decomposing* (Definition 3.2) both trees into sets of SISO trees, and
(2) Checking whether it is possible to find SISO trees W, from the decomposition of the considered subtree, and W', from the decomposition of the considered super-tree, such that W' *refines* 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 3.5 for further details on refinement.

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; see Lemma 3.3.

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

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

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

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

```
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 }.
```

The notation `[|...|]` denotes a finite colist of members of an arbitrary type, namely `(cocons ... conil)`.

Formalising SISO trees coinductively in Coq follows the *greatest fixed point of the least fixed point* technique, using the Paco library [31, 53]. We define an inductive **`Prop`** predicate `singletonI`, which acts as a generating function. It is *parametrised* by a relation `R` of the same signature, allowing for *knowledge accumulation* during coinductive foldings. The greatest fixed point `singleton` is derived using `paco1`, provided that the generating function is *monotone* (condition met by `singletonI` 🐓) and is initialised with the empty relation `bot1`. The suffix 1 indicates that the generating function takes one argument: `st`. This implementation enables us to apply constructors of the generating function `singletonI` infinitely often, making the declaration of `singleton` coinductive.

We then formulate SISO trees as a sigma type of a session tree und such that und respects `singleton`.

$$\frac{\forall i \in I \quad \mathsf{T}'_i \lhd_{\mathsf{SO}} \mathsf{T}_i}{\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}'_i \lhd_{\mathsf{SO}} \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SO-RCV}] \qquad \frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{T} \lhd_{\mathsf{SO}} \mathsf{T}_k}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{T} \lhd_{\mathsf{SO}} \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SO-SND}]$$

$$\frac{}{\mathsf{end} \lhd_{\mathsf{SO}} \mathsf{end}}\ [\text{SO-END}]$$

$$\frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{T} \lhd_{\mathsf{SI}} \mathsf{T}_k}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{T} \lhd_{\mathsf{SI}} \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SI-RCV}] \qquad \frac{\forall i \in I \quad \mathsf{T}'_i \lhd_{\mathsf{SI}} \mathsf{T}_i}{\bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}'_i \lhd_{\mathsf{SI}} \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SI-SND}]$$

$$\frac{}{\mathsf{end} \lhd_{\mathsf{SI}} \mathsf{end}}\ [\text{SI-END}]$$

$$\frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{W} \lhd_{\mathsf{SISO}} \mathsf{T}_k}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \lhd_{\mathsf{SISO}} \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SISO-RCV}] \qquad \frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad \mathsf{W} \lhd_{\mathsf{SISO}} \mathsf{T}_k}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \lhd_{\mathsf{SISO}} \bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathsf{T}_i}\ [\text{SISO-SND}]$$

$$\frac{}{\mathsf{end} \lhd_{\mathsf{SISO}} \mathsf{end}}\ [\text{SISO-END}]$$

Fig. 2. SI, SO and SISO Decompositions

Parametrised coinduction with `Paco` has been used by Zakowski et al. [53] to define weak bisimilarity on streams, and by Tirore et al. [50] to define a sound and complete projection of global session types onto local types. It has also been employed in Zooid by Castro-Perez et al. [9] to interpret global and local types using trees similar to ours, though their setting is synchronous.

Remark 4. *The use of the* `Paco` *library is beneficial—across many constructions presented throughout the paper—as it enables coinductive reasoning parametrised by "accumulated knowledge". This allows proof goals to be discharged when an element already present in the knowledge set is encountered during coinductive folding steps. Moreover,* `Paco` *relies on semantic guardedness rather than syntactic guard checks, which can fail even under straightforward setoid rewrites.*

In a slightly different way from the one outlined in § 3.4 of [22], we construct coinductive relations consisting of pairs of the form SO tree–session tree, SI tree–session tree, and SISO tree–session tree, where the former is obtained by decomposing the latter.

Definition 3.2. (🐓, 🐓, 🐓) The SO, SI, and SISO decompositions of a session tree are specified by the coinductive relations $\lhd_{\mathsf{SO}}$, $\lhd_{\mathsf{SI}}$, and $\lhd_{\mathsf{SISO}}$, which are defined by the rules given in Figure 2, with continuations ranging over some finite index set I.

The relations introduced in Definition 3.2 are implemented coinductively in Coq as `st2soC`, `st2siC`, and `st2sisoC`. In what follows, we concentrate on the implementation of the last one.

This implementation operates over session trees rather than SISO trees in order to avoid checking for singleton structures at each rule application step. Nevertheless, when the relation is used, it is always applied to the underlying `und` component of a `siso` tree. For instance, see the formal subtyping definition `subtype` in Definition 6.3.

We adopt this strategy throughout, up until § 4.1, where the negation of the refinement relation, `nRefinement`, is defined directly on `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 y p, R x y → copathsel l s xs y → st2siso R (p & [|(l,s,x)|]) (st_receive p xs)
 | st2siso_snd:∀ l s x xs y p, R x y → copathsel l s xs y → st2siso R (p ! [|(l,s,x)|]) (st_send p xs).
```

```
Definition st2sisoC s1 s2 ≜ paco2 (st2siso) bot2 s1 s2.
```

The relation `copathsel` is defined inductively to ensure that, within the given colist `xs` of selections and branchings (triples consisting of a label, sort, and continuation tree), the label `l` and sort `s` matches the continuation tree `y`.

Lemma 3.3. (🐓, 🐓)

(1) $\forall \mathsf{T}\ \mathsf{V}\ \mathsf{W},\quad \mathsf{T} \lhd_{\mathsf{SI}} \mathsf{V}$ *and* $\mathsf{V} \lhd_{\mathsf{SO}} \mathsf{W}$ *implies that* $\mathsf{T} \lhd_{\mathsf{SISO}} \mathsf{W}$

(2) $\forall \mathsf{T}\ \mathsf{U}\ \mathsf{W},\quad \mathsf{T} \lhd_{\mathsf{SO}} \mathsf{U}$ *and* $\mathsf{U} \lhd_{\mathsf{SI}} \mathsf{W}$ *implies that* $\mathsf{T} \lhd_{\mathsf{SISO}} \mathsf{W}$.

3.2 SISO Tree Action Reordering and Refinement

The key factor in determining whether a given session tree is a subtree (or super-tree) of another is considering the possibility of action reordering [22, Def. 3.2]. In this context, the subtree has the ability to "anticipate" certain input/output actions, allowing them to be executed earlier or later than their prescribed order in the super-tree.

Definition 3.4. (🐓, 🐓). In order to clarify action reorderings, a pair of input/output sequences is recursively defined as follows:

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

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.

In our Coq implementation (`Ap p` for $\mathcal{A}^{(\mathsf{p})}$ and `Bp p` for $\mathcal{B}^{(\mathsf{p})}$), we adopt a dependently typed approach, carrying participant dis-equality proofs within the type. While this approach is useful for ensuring correctness at the type level, it becomes cumbersome when proving related properties (See Note 6).

Definition 3.5. The *refinement* relation $\lesssim$ over SISO trees is coinductively defined with:

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

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

The symbol $\leq:$ denotes the least reflexive and transitive relation over payload sorts (e.g., `nat` $\leq:$ `int`). The function `act` coinductively abstracts a given SISO tree into a set of actions, collecting participant–*dir* pairs with $dir \in !, ?$. For instance, $\mathsf{act}(\mathsf{W})$ denotes the set of actions occurring in W. Action-set equality tests are then used to ensure that applications of refinement rules preserve this set of actions, that is, they neither introduce new actions nor remove existing ones.

The rule [REF-$\mathcal{B}$] captures the reordering backed by $\mathcal{B}^{(\mathsf{p})}$ type of prefixing. It allows for swapping the order of an output directed towards a participant p (a finite number of times) with any combination of input actions, as well as with output actions—excluding those that are also to p. On the other hand, the rule [REF-$\mathcal{A}$] governs the reordering of an input from a participant p with any combination of input actions, except those from p.

Remark 5. *In contrast to the original definition of the refinement relation in [22, Def. 3.2], we allow the prefixes $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ to include the empty prefix ε. This deviation introduces flexibility into the framework by permitting contexts with no actions. Notably, this modification simplifies the proof of an*

inversion lemma stating that SISO trees with action dis-equality cannot refine each other. This forms one of the key points of our Coq formalisation.

Lemma 3.6. $\forall W\ W',\quad \mathtt{act}(W) \neq \mathtt{act}(W') \implies \neg(W \lesssim W')$.

This lemma is a significant result, as it establishes a fundamental property concerning the relationship between terms with mismatched actions. Note that the definition of action dis-equality used here is obtained by negating the statement in Definition 3.10.

3.3 Refinement Relation in Coq

This section presents a Coq formalisation of the refinement relation defined in Definition 3.5. The corecursive function `act` accumulates the actions (i.e., `participant–dir` pairs) of a session tree W into a stream (coseq), encoding the corresponding action-set $\mathtt{act}(W)$:

```
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.
```

We then define an inductively specified membership predicate `coseqIn` on streams of actions, which constitutes the core mechanism for expressing mutual inclusion checks over such streams:

```
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.
```

Defining the membership test inductively is essential, as it enables the proof of several key properties. For example, Lemmas 3.7 and 3.8 cannot be proved if membership (`coseqIn`, denoted $\in$) were defined coinductively, since coinductive definitions do not support reasoning about the structure of terms. These lemmas are instrumental in establishing the correctness of the refinement relation and completeness of the subtyping relation, with respect to negations.

Lemma 3.7. $\forall \mathtt{p}\ W, \mathtt{p}? \in \mathtt{act}(W) \implies \exists C^{(\mathtt{p})}\ \ell\ S\ W',\quad W = C^{(\mathtt{p})}.\mathtt{p}?\ell(S).W'$.

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

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

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

Lemma 3.8. $\forall \mathtt{p}\ W, \mathtt{p}! \in \mathtt{act}(W) \implies \exists \mathcal{B}^{(\mathtt{p})}\ \ell\ S\ W',\quad W = \mathcal{B}^{(\mathtt{p})}.\mathtt{p}!\ell(S).W'$.

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

Lemma 3.9. $\forall \mathtt{p}\ C^{(\mathtt{p})}\ W,\quad \mathtt{p}! \notin C^{(\mathtt{p})} \implies \left(\exists \mathcal{A}^{(\mathtt{p})},\quad C^{(\mathtt{p})}.W = \mathcal{A}^{(\mathtt{p})}.W\right)$.

Definition 3.10. For a pair of SISO trees W and W′, we define action equality as follows:

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

```
Definition act_eq  (W W': st) ≜ ∀ a, coseqIn a (act W) ↔ coseqIn a (act W').
Definition act_neq (W W': st) ≜ ∃ a, coseqIn a (act W) ∧ (coseqIn a (act W') → False) ∨
                                     coseqIn a (act W') ∧ (coseqIn a (act W) → False).
```

In order to encode the action-set equality tests act(W) = act(W′) in Definition 3.5, we employ the predicate `act_eq W W'` (with `act_neq W W'` denoting dis-equality test). This predicate is specified via mutual inclusion—using the membership check `coseqIn`—between the coseqs `act W` and `act W'`, discarding both the order of actions and their multiplicity.

We then formalise the refinement relation, incorporating the action equality test `act_eq` (Definition 3.10) . In the Coq development, the rule [REF-$\mathcal{B}$] is implemented by the constructor `ref_b`, while [REF-$\mathcal{A}$] is implemented by `ref_a`.

```
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 (p & [|(l,s,w)|]) (merge_ap_contn p a (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 (p ! [|(l,s,w)|]) (merge_bp_contn p b (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 function `merge_ap_cont` (resp. `merge_bp_cont`, `merge_cp_cont`) takes an argument `a: Ap p` (resp. `b: Bp p`, `c: Cp p`), a SISO tree `w`, and prefixes `a` (resp. `b`, `c`) to `w`. As a variation, we define `merge_ap_contn` (resp. `merge_bp_contn`, `merge_cp_contn`), which also takes a natural number `n`, merges `a` (resp. `b`, `c`) with itself `n` times, and prefixes the result to `w`.

3.4 Correctness of Refinement

The main goal of this section is to establish the correctness of the refinement relation, in the sense that it does not permit the reordering of send (resp. receive) actions to (resp. from) a given participant `p`. In other words, any reordering allowed by refinement is restricted to inputs from distinct participants or to outputs directed to distinct participants, possibly interleaved with arbitrary input actions. To prove this property, we first introduce a number of preliminary definitions and results.

NOTE 6. *To encode the lemmas, theorems (and proofs) presented in this section within a Coq implementation, we adopt non-dependently typed versions of the prefixing constructs $\mathcal{A}$ (`Apf`), $\mathcal{B}$ (`Bpf`), and C (`Cpf`), where declarations are no longer parameterised by the participant `p`. Instead, we explicitly capture the absence of certain actions using predicates `isInA` , `isInB` , and `isInC` . For example, `isInA a p = false` guarantees that the prefix `a: Apf` contains no receive-from-`p` actions.*

It is possible to build an instance of the type `Ap p` given an instance `a` of `Apf` and the fact that `isInA a p = false`. Conversely, an instance `a` of the type `Apf` along with `isInA a p = false` can be constructed from an instance of `Ap p`. We can therefore prove variants of Lemmas 3.7, 3.8 and 3.9 using non-parametrised prefixes.

This non-parametric approach becomes particularly useful when formalising prefixes that exclude receive-from (and send-to) actions for multiple participants simultaneously. For example, one can state the propositions `isInA a p = false` and `isInA a q = false` for some prefix `a : Apf`, to establish that `a` contains neither receive-from-`p` nor receive-from-`q` actions. Crucially, this formulation supports compositional reasoning about prefixes, as such exclusion properties can be combined and reused independently of the prefix structure. This, in turn, simplifies the related Coq proofs involving `a`, as it is no longer necessary to carry participant dis-equality proofs within the prefix itself—such information is independently available through the given propositions. As a

result, this approach enables simpler and more modular proofs. See Lemma 3.13 for reference; this approach also avoids Coq complaints in prefix comparisons: $\mathcal{B}_1 \neq \mathcal{B}_2$ and $\mathcal{A}_1 \neq \mathcal{A}_2$.

We define the glue functions `merge_apf_cont` 🐓, `merge_bpf_cont` 🐓, and `merge_cpf_cont` 🐓, which replicate the functionalities of `merge_ap_cont`, `merge_bp_cont`, and `merge_cp_cont` in the non-parametric setting.

Introducing new prefixing sorts naturally leads to the definition of a new refinement relation, `refinement4` 🐓.

```
Inductive refinementR4 (seq: st → st → Prop): st → st → Prop ≜
  | ref4_a  : ∀ w w' p l s s' a n, subsort s' s → isInA a p = false → seq w (merge_apf_contn a w' n)  →
              act_eq w (merge_apf_contn a w' n) →
              refinementR4 seq (p & [|(l,s,w)|]) (merge_apf_contn a (p & [|(l,s',w')|]) n)
  | ref4_b  : ∀ w w' p l s s' b n, subsort s s' → isInB b p = false → seq w (merge_bpf_contn b w' n) →
              act_eq w (merge_bpf_contn b w' n) →
              refinementR4 seq (st_send p [|(l,s,w)|]) (merge_bpf_contn b (st_send p [|(l,s',w')|]) n)
  | ref4_end: refinementR4 seq st_end st_end.

Definition refinement4: st → st → Prop ≜ fun s1 s2 ⇒ paco2 refinementR4 bot2 s1 s2.
```

The function `merge_apf_contn` 🐓 (resp. `merge_bpf_contn` 🐓, `merge_cpf_contn` 🐓) mirrors the behaviour of `merge_ap_contn` (resp. `merge_bp_contn`, `merge_cp_contn`), but is defined using the non-dependently typed prefixing `Apf` (resp. `Bpf`, `Cpf`).

Regarding naming, we adopt the identifiers `refinement2` and `refinement4` to distinguish them from the relations `refinement` and `refinement3`, which are introduced in § 6.1. The latter correspond, respectively, to `refinement2` and `refinement4`, but incorporate changes solely in the way actions are compared, under a *finiteness* assumption—namely, that in a session with a potentially infinite number of interactions, there are only finitely many distinct actions (see Definition 6.1 and the preceding discussion). These revised relations are used to verify concrete subtyping examples drawn from the literature (§ 6.4 and 6.3), whereas the original relations `refinement2` and `refinement4` are employed to establish meta-properties of subtyping, such as correctness (Theorem 3.25) and completeness (Lemma 5.1). Under *finiteness*, relations `refinement2` and `refinement4` coincide as do `refinement` and `refinement3`.

We prove that `refinement2` and `refinement4` are equivalent which allows for transporting properties from one end to the other 🐓.

```
Theorem refEquiv24: ∀ w w', refinement2 w w' ↔ refinement4 w w'.
```

Table 1 presents a comprehensive comparison between `refinement2` and `refinement4`, highlighting the key differences in their treatment of action prefixing, prefix well-formedness, and proof engineering. In particular, adopting a simple (i.e., non-dependently typed) notion of action prefixing in `refinement4` enables compositional reasoning: prefixes that exclude send-to or receive-from actions for multiple participants (e.g., p, q, and others) can be expressed directly. This simplifies proofs by eliminating the need to carry participant dis-equality arguments within the prefix itself, leading to simpler and more modular proofs. However, this increased flexibility requires prefix well-formedness to be enforced explicitly, which is achieved via the predicates `isInA` and `isInB`.

The correctness result, Theorem 3.25, is first proven in Coq with respect to `refinement4`, and then shown to hold for `refinement2` by leveraging their equivalence.

In what follows, we proceed with a slight overloading/abuse of notation and employ $\lesssim$ both for `refinement2` and `refinement4`. However, we explicitly indicate action memberships and prefixing types in the subsequent statements to remain consistent with our Coq implementation.

Aspect	`refinement2`	`refinement4`
Action prefixing	Dependently typed	Non-dependently typed
Prefix well-formedness	Enforced by types	Explicit via predicates (`isInA`, `isInB`)
Reasoning about reordering	Implicit in typing	Explicit and compositional
Proof engineering	Rigid, less flexible	Simpler and more modular

Table 1. Comparison of `refinement2` and `refinement4`

Prefixing Facts. When a term has multiple distinct prefix-continuation pairs, the continuations and prefixes can be expressed in terms of each other.

Lemma 3.11. $\forall \mathsf{p}\ \mathcal{B}\ \mathrm{W}\ \mathrm{W}',\quad \mathsf{p}!\notin \mathcal{B}$ *and* $\mathsf{p}!\ell(\mathrm{S}).\mathrm{W} = \mathcal{B}.\mathrm{W}'$ *implies that* $\mathcal{B} = \varepsilon$ *and* $\mathrm{W}' = \mathsf{p}!\ell(\mathrm{S}).\mathrm{W}$.

Lemma 3.12. (🐓, 🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathcal{A}_1^{(\mathsf{p})}\ \mathcal{A}_2^{(\mathsf{p})}\ \ell_1\ \ell_2\ \mathrm{S}_1\ \mathrm{S}_2\ \mathrm{W}_1\ \mathrm{W}_2,\quad \mathcal{A}_1^{(\mathsf{p})}.\mathsf{p}?\ell_1(\mathrm{S}_1).\mathrm{W}_1 = \mathcal{A}_2^{(\mathsf{p})}.\mathsf{p}?\ell_2(\mathrm{S}_2).\mathrm{W}_2$ *implies that* $\mathsf{p}?\ell_1(\mathrm{S}_1).\mathrm{W}_1 = \mathsf{p}?\ell_2(\mathrm{S}_2).\mathrm{W}_2$.

(2) $\forall \mathsf{p}\ \mathcal{B}_1^{(\mathsf{p})}\ \mathcal{B}_2^{(\mathsf{p})}\ \ell_1\ \ell_2\ \mathrm{S}_1\ \mathrm{S}_2\ \mathrm{W}_1\ \mathrm{W}_2,\quad \mathcal{B}_1^{(\mathsf{p})}.\mathsf{p}!\ell_1(\mathrm{S}_1).\mathrm{W}_1 = \mathcal{B}_2^{(\mathsf{p})}.\mathsf{p}!\ell_2(\mathrm{S}_2).\mathrm{W}_2$ *implies that* $\mathsf{p}!\ell_1(\mathrm{S}_1).\mathrm{W}_1 = \mathsf{p}!\ell_2(\mathrm{S}_2).\mathrm{W}_2$.

(3) $\forall \mathsf{p}\ C_1^{(\mathsf{p})}\ C_2^{(\mathsf{p})}\ \ell_1\ \ell_2\ \mathrm{S}_1\ \mathrm{S}_2\ \mathrm{W}_1\ \mathrm{W}_2,\quad C_1^{(\mathsf{p})}.\mathsf{p}?\ell_1(\mathrm{S}_1).\mathrm{W}_1 = C_2^{(\mathsf{p})}.\mathsf{p}?\ell_2(\mathrm{S}_2).\mathrm{W}_2$ *implies that* $\mathsf{p}?\ell_1(\mathrm{S}_1).\mathrm{W}_1 = \mathsf{p}?\ell_2(\mathrm{S}_2).\mathrm{W}_2$.

Lemma 3.13. (🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathsf{q}\ \mathcal{A}_1\ \mathcal{A}_2\ \mathrm{W}\ \mathrm{W}_1\ \mathrm{W}_2,\quad \mathsf{p}?\notin \mathcal{A}_1$ *and* $\mathsf{q}?\notin \mathcal{A}_2$ *and* $\mathcal{A}_1 \neq \mathcal{A}_2$ *and* $\mathrm{W} = \mathcal{A}_1.\mathrm{W}_1 = \mathcal{A}_2.\mathrm{W}_2$ *implies that*
$\exists \mathcal{A}_3,\quad \mathsf{q}?\notin \mathcal{A}_3$ *and* $\mathrm{W} = \mathcal{A}_1.\mathcal{A}_3.\mathrm{W}_2$ *and* $\mathcal{A}_2 = \mathcal{A}_1.\mathcal{A}_3$ *and* $\mathrm{W}_1 = \mathcal{A}_3.\mathrm{W}_2$ *or*
$\exists \mathcal{A}_3,\quad \mathsf{p}?\notin \mathcal{A}_3$ *and* $\mathrm{W} = \mathcal{A}_2.\mathcal{A}_3.\mathrm{W}_1$ *and* $\mathcal{A}_1 = \mathcal{A}_2.\mathcal{A}_3$ *and* $\mathrm{W}_2 = \mathcal{A}_3.\mathrm{W}_1$.

(2) $\forall \mathsf{p}\ \mathsf{q}\ \mathcal{B}_1\ \mathcal{B}_2\ \mathrm{W}\ \mathrm{W}_1\ \mathrm{W}_2,\quad \mathsf{p}!\notin \mathcal{B}_1$ *and* $\mathsf{q}!\notin \mathcal{B}_2$ *and* $\mathcal{B}_1 \neq \mathcal{B}_2$ *and* $\mathrm{W} = \mathcal{B}_1.\mathrm{W}_1 = \mathcal{B}_2.\mathrm{W}_2$ *implies that*
$\exists \mathcal{B}_3,\quad \mathsf{q}!\notin \mathcal{B}_3$ *and* $\mathrm{W} = \mathcal{B}_1.\mathcal{B}_3.\mathrm{W}_2$ *and* $\mathcal{B}_2 = \mathcal{B}_1.\mathcal{B}_3$ *and* $\mathrm{W}_1 = \mathcal{B}_3.\mathrm{W}_2$ *or*
$\exists \mathcal{B}_3,\quad \mathsf{p}!\notin \mathcal{B}_3$ *and* $\mathrm{W} = \mathcal{B}_2.\mathcal{B}_3.\mathrm{W}_1$ *and* $\mathcal{B}_1 = \mathcal{B}_2.\mathcal{B}_3$ *and* $\mathrm{W}_2 = \mathcal{B}_3.\mathrm{W}_1$.

Lemma 3.14. (🐓, 🐓, 🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathsf{q}\ \mathcal{A}\ \ell_1\ \ell_2\ \mathrm{S}\ \mathrm{S}'\ \mathrm{W}\ \mathrm{W}',\quad \mathsf{p}?\notin \mathcal{A}$ *and* $\mathsf{p} \neq \mathsf{q}$ *and* $\mathsf{q}?\ell_1(\mathrm{S}).\mathrm{W} = \mathcal{A}.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *implies that* $\exists \mathcal{A}_1,\quad \mathsf{p}?\notin \mathcal{A}_1$ *and* $\mathrm{W} = \mathcal{A}_1.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *and* $\mathcal{A} = \mathsf{q}?\ell_1(\mathrm{S}).\mathcal{A}_1$.

(2) $\forall \mathsf{p}\ \mathsf{q}\ \mathcal{B}\ \ell_1\ \ell_2\ \mathrm{S}\ \mathrm{S}'\ \mathrm{W}\ \mathrm{W}',\quad \mathsf{p}!\notin \mathcal{B}$ *and* $\mathsf{p} \neq \mathsf{q}$ *and* $\mathsf{q}!\ell_1(\mathrm{S}).\mathrm{W} = \mathcal{B}.\mathsf{p}!\ell_2(\mathrm{S}').\mathrm{W}'$ *implies that* $\exists \mathcal{B}_1,\quad \mathsf{p}!\notin \mathcal{B}_1$ *and* $\mathrm{W} = \mathcal{B}_1.\mathsf{p}!\ell_2(\mathrm{S}').\mathrm{W}'$ *and* $\mathcal{B} = \mathsf{q}!\ell_1(\mathrm{S}).\mathcal{B}_1$.

(3) $\forall \mathsf{p}\ \mathsf{q}\ C\ \ell_1\ \ell_2\ \mathrm{S}\ \mathrm{S}'\ \mathrm{W}\ \mathrm{W}',\quad \mathsf{p}?\notin C$ *and* $\mathsf{p} \neq \mathsf{q}$ *and* $\mathsf{q}?\ell_1(\mathrm{S}).\mathrm{W} = C.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *implies that* $\exists C_1,\quad \mathsf{p}?\notin C_1$ *and* $\mathrm{W} = C_1.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *and* $C = \mathsf{q}?\ell_1(\mathrm{S}).C_1$.

(4) $\forall \mathsf{p}\ \mathsf{q}\ C\ \ell_1\ \ell_2\ \mathrm{S}\ \mathrm{S}'\ \mathrm{W}\ \mathrm{W}',\quad \mathsf{p}?\notin C$ *and* $\mathsf{q}!\ell_1(\mathrm{S}).\mathrm{W} = C.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *implies that* $\exists C_1,\quad \mathsf{p}?\notin C_1$ *and* $\mathrm{W} = C_1.\mathsf{p}?\ell_2(\mathrm{S}').\mathrm{W}'$ *and* $C = \mathsf{q}!\ell_1(\mathrm{S}).C_1$.

These statements are used to establish proofs of Lemmas 3.15, 3.17, and 3.18, which play a crucial role in constructing the proof of the correctness Theorem 3.25.

Refinement Inversion Facts. When a term with an action $\mathsf{p}?\ell(\mathrm{S})$ (resp. $\mathsf{p}!\ell(\mathrm{S})$) is refined by a term $\mathcal{D}_1.\mathrm{W}'$, then either the prefix $\mathcal{D}_1$ (with no constraints) contains a receive (resp. send) action from (resp. to) p with the same label and a subsort (resp. super-sort), or the term W'.

Lemma 3.15. (🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathcal{A}_1\ \mathcal{D}_1\ \ell\ \mathsf{S}\ \mathsf{W}\ \mathsf{W}',\quad \mathsf{p}?\notin \mathcal{A}_1$ *and* $\mathcal{A}_1.\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{D}_1.\mathsf{W}'$ *implies that*
$\exists \mathcal{A}_2\ \mathcal{D}_2\ \mathsf{S}',\quad \mathsf{p}?\notin \mathcal{A}_2$ *and* $\mathsf{S}' \leq: \mathsf{S}$ *and* $\mathcal{D}_1 = \mathcal{A}_2.\mathsf{p}?\ell(\mathsf{S}').\mathcal{D}_2$ *or*
$\exists \mathcal{A}_2\ \mathsf{W}_1\ \mathsf{S}',\quad \mathsf{p}?\notin \mathcal{A}_2$ *and* $\mathsf{S}' \leq: \mathsf{S}$ *and* $\mathsf{p}?\notin \mathcal{D}_1$ *and* $\mathsf{W}' = \mathcal{A}_2.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}_1$.

(2) $\forall \mathsf{p}\ \mathcal{B}_1\ \mathcal{D}_1\ \ell\ \mathsf{S}\ \mathsf{W}\ \mathsf{W}',\quad \mathsf{p}!\notin \mathcal{B}_1$ *and* $\mathcal{B}_1.\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{D}_1.\mathsf{W}'$ *implies that*
$\exists \mathcal{B}_2\ \mathcal{D}_2\ \mathsf{S}',\quad \mathsf{p}!\notin \mathcal{B}_2$ *and* $\mathsf{S} \leq: \mathsf{S}'$ *and* $\mathcal{D}_1 = \mathcal{B}_2.\mathsf{p}!\ell(\mathsf{S}').\mathcal{D}_2$ *or*
$\exists \mathcal{B}_2\ \mathsf{W}_1\ \mathsf{S}',\quad \mathsf{p}!\notin \mathcal{B}_2$ *and* $\mathsf{S} \leq: \mathsf{S}'$ *and* $\mathsf{p}!\notin \mathcal{D}_1$ *and* $\mathsf{W}' = \mathcal{B}_2.\mathsf{p}!\ell(\mathsf{S}').\mathsf{W}_1$.

A pair of terms, one refining the other, have exactly the same set of actions.

Lemma 3.16. (🐓, 🐓) $\forall \mathsf{W}\ \mathsf{W}',\quad \mathsf{W} \lesssim \mathsf{W}'$ *implies that*

(1) $\forall a,\quad a \in \mathtt{act}(\mathsf{W})$ *if and only if* $a \in \mathtt{act}(\mathsf{W}')$.
(2) $\forall a,\quad a \notin \mathtt{act}(\mathsf{W})$ *if and only if* $a \notin \mathtt{act}(\mathsf{W}')$.

The first receive (resp. send) actions from (resp. to) a particular participant p within a pair of terms, one refining the other, have matching labels and correctly organised payload sorts — that is, the refined term has the super-sort when a receive action is considered, and the subsort otherwise.

Lemma 3.17. (🐓, 🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathcal{A}_1\ \mathcal{A}_2\ \ell_1\ \ell_2\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}?\notin \mathcal{A}_1$ *and* $\mathsf{p}?\notin \mathcal{A}_2$ *and* $\mathcal{A}_1.\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{W} \lesssim \mathcal{A}_2.\mathsf{p}?\ell_2(\mathsf{S}').\mathsf{W}'$ *implies that* $\ell_1 = \ell_2$ *and* $\mathsf{S}' \leq: \mathsf{S}$.
(2) $\forall \mathsf{p}\ \mathcal{B}_1\ \mathcal{B}_2\ \ell_1\ \ell_2\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}!\notin \mathcal{B}_1$ *and* $\mathsf{p}!\notin \mathcal{B}_2$ *and* $\mathcal{B}_1.\mathsf{p}!\ell_1(\mathsf{S}).\mathsf{W} \lesssim \mathcal{B}_2.\mathsf{p}!\ell_2(\mathsf{S}').\mathsf{W}'$ *implies that* $\ell_1 = \ell_2$ *and* $\mathsf{S} \leq: \mathsf{S}'$.
(3) $\forall \mathsf{p}\ C_1\ C_2\ \ell_1\ \ell_2\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}?\notin C_1$ *and* $\mathsf{p}?\notin C_2$ *and* $C_1.\mathsf{p}?\ell_1(\mathsf{S}).\mathsf{W} \lesssim C_2.\mathsf{p}?\ell_2(\mathsf{S}').\mathsf{W}'$ *implies that* $\ell_1 = \ell_2$ *and* $\mathsf{S}' \leq: \mathsf{S}$.

From a pair of terms, one refining the other, it is possible to drop the first receive (resp. send) actions from (resp. to) a certain participant p with matching labels.

Lemma 3.18. (🐓, 🐓, 🐓)

(1) $\forall \mathsf{p}\ \mathcal{A}_1\ \mathcal{A}_2\ \ell\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}?\notin \mathcal{A}_1$ *and* $\mathsf{p}?\notin \mathcal{A}_2$ *and* $\mathcal{A}_1.\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{A}_2.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'$ *implies that* $\mathcal{A}_1.\mathsf{W} \lesssim \mathcal{A}_2.\mathsf{W}'$.
(2) $\forall \mathsf{p}\ \mathcal{B}_1\ \mathcal{B}_2\ \ell\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}!\notin \mathcal{B}_1$ *and* $\mathsf{p}!\notin \mathcal{B}_2$ *and* $\mathcal{B}_1.\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \lesssim \mathcal{B}_2.\mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'$ *implies that* $\mathcal{B}_1.\mathsf{W} \lesssim \mathcal{B}_2.\mathsf{W}'$.
(3) $\forall \mathsf{p}\ C_1\ C_2\ \ell\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}',\ \mathsf{p}?\notin C_1$ *and* $\mathsf{p}?\notin C_2$ *and* $C_1.\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \lesssim C_2.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'$ *implies that* $C_1.\mathsf{W} \lesssim C_2.\mathsf{W}'$.

A term that starts with a receive action cannot be refined by a term starting with a send action.

Lemma 3.19. 🐓 $\forall \mathsf{p}\ \mathsf{q}\ \ell_1\ \ell_2\ \mathsf{S}_1\ \mathsf{S}_2\ \mathsf{W}_1\ \mathsf{W}_2,\quad \mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{W}_1 \lesssim \mathsf{q}!\ell_2(\mathsf{S}_2).\mathsf{W}_2$ *implies false.*

SISO Projections. We define input and output projections of SISO trees onto participants, intuitively, filtering all unmatching actions out.

Definition 3.20 (SISO projections). (🐓, 🐓) Output and input projection of some SISO tree W onto a participant p, respectively denoted $\mathsf{W} \upharpoonright^! \mathsf{p}$ and $\mathsf{W} \upharpoonright^? \mathsf{p}$, are coinductively defined by the following equations:

$$\mathsf{end} \upharpoonright^! \mathsf{p} = \mathsf{end}$$
$$(\mathsf{p}!\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^! \mathsf{p} = \mathsf{p}!\ell(\mathsf{S}).(\mathsf{W}' \upharpoonright^! \mathsf{p})$$
$$(\mathsf{q}!\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^! \mathsf{p} = \begin{cases} \mathsf{W}' \upharpoonright^! \mathsf{p} & \text{if } \mathsf{q} \neq \mathsf{p} \wedge \mathsf{p}! \in \mathsf{W}' \\ \mathsf{end} & \text{if } \mathsf{q} \neq \mathsf{p} \wedge \mathsf{p}! \notin \mathsf{W}' \end{cases}$$
$$(\mathsf{q}?\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^! \mathsf{p} = \begin{cases} \mathsf{W}' \upharpoonright^! \mathsf{p} & \text{if } \mathsf{p}! \in \mathsf{W}' \\ \mathsf{end} & \text{if } \mathsf{p}! \notin \mathsf{W}' \end{cases}$$

$$\mathsf{end} \upharpoonright^? \mathsf{p} = \mathsf{end}$$
$$(\mathsf{p}?\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^? \mathsf{p} = \mathsf{p}?\ell(\mathsf{S}).(\mathsf{W}' \upharpoonright^? \mathsf{p})$$
$$(\mathsf{q}?\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^? \mathsf{p} = \begin{cases} \mathsf{W}' \upharpoonright^? \mathsf{p} & \text{if } \mathsf{q} \neq \mathsf{p} \wedge \mathsf{p}? \in \mathsf{W}' \\ \mathsf{end} & \text{if } \mathsf{q} \neq \mathsf{p} \wedge \mathsf{p}? \notin \mathsf{W}' \end{cases}$$
$$(\mathsf{q}!\ell(\mathsf{S}).\mathsf{W}') \upharpoonright^? \mathsf{p} = \begin{cases} \mathsf{W}' \upharpoonright^? \mathsf{p} & \text{if } \mathsf{p}? \in \mathsf{W}' \\ \mathsf{end} & \text{if } \mathsf{p}? \notin \mathsf{W}' \end{cases}$$

```
Inductive projS (R: st → participant → st → Prop): st → participant → st → Prop ≜
  | ...
  | pjs_snde: ∀ p l s w' w, R w' p w → projS R (p ! [|(l,s,w')|]) p (p ! [|(l,s,w)|])
  | pjs_sndI: ∀ q l s w' p w, p ≠ q → coseqIn (p, snd) (act w') →
              R w' p w → projS R (q ! [|(l,s,w')|]) p w
  | pjs_rcvI: ∀ q l s w' p w, coseqIn (p, snd) (act w') → R w' p w → projS R (q & [|(l,s,w')|]) p w.

Definition projSC w1 p w2 ≜ paco3 projS bot3 w1 p w2.
```

We develop the output (resp. input) projection of a stream line of actions `w` onto a participant `p` in Coq via the coinductive relation `projSC w p w'` (resp. `projRC w p w'`), which retains the `(p, snd)` (resp. `(p, rcv)`) actions while discarding all others.

In the implementation of `projSC`, the output projection of the term `q ! [|(l,s,w')|]` onto the participant `p` is `p ! [|(l,s,w)|]` when `p = q` and `w'` coinductively projects into `w` (constructor `pjs_snde`). If `p` and `q` differ, the term projects into `w` only if `(p, snd)` is in the actions of `w'` and `w'` projects into `w` (constructor `pjs_sndI`).

Similarly, the output projection of the term `q & [|(l,s,w')|]` is `w` provided that `(p, snd)` appears in the actions of `w'` and `w'` projects into `w` (constructor `pjs_rcvI`).

SISO Projection Facts. Output (resp. input) projection of some term `w` onto `p` results in `st_end` if `(p, snd)` (resp. `(p, rcv)`) is not involved in the set of actions of `w`.

Lemma 3.21. (🐓, 🐓)

(1) $\forall p\ W,\quad p? \notin \mathsf{act}(W) \implies W \upharpoonright^{?} p = \mathsf{end}.$
(2) $\forall p\ W,\quad p! \notin \mathsf{act}(W) \implies W \upharpoonright^{!} p = \mathsf{end}.$

```
Lemma pjr_notin_end: ∀ w p, ~coseqIn (p, rcv) (act (@und w)) → projRC (@und w) p st_end.
Lemma pjs_notin_end: ∀ w p, ~coseqIn (p, snd) (act (@und w)) → projSC (@und w) p st_end.
```

Given an output (resp. input) projection of a term with prefixing, the participants, labels, and sorts match those of the resulting term. Moreover, the continuation of the original term projects into the continuation of the resulting term.

Lemma 3.22. (🐓, 🐓, 🐓)

(1) $\forall p\ q\ \mathcal{A}\ \ell_1\ \ell_2\ S\ S'\ W\ W',\ p? \notin \mathcal{A}$ *and* $(\mathcal{A}.p?\ell_1(S).W) \upharpoonright^{?} p = q?\ell_2(S').W'$ *implies that* $p = q$ *and* $\ell_1 = \ell_2$ *and* $S = S'$ *and* $W \upharpoonright^{?} p = W'$.
(2) $\forall p\ q\ \mathcal{B}\ \ell_1\ \ell_2\ S\ S'\ W\ W',\ p! \notin \mathcal{B}$ *and* $(\mathcal{B}.p!\ell_1(S).W) \upharpoonright^{!} p = q!\ell_2(S').W'$ *implies that* $p = q$ *and* $\ell_1 = \ell_2$ *and* $S = S'$ *and* $W \upharpoonright^{!} p = W'$.
(3) $\forall p\ q\ C\ \ell_1\ \ell_2\ S\ S'\ W\ W',\ p? \notin C$ *and* $(C.p?\ell_1(S).W) \upharpoonright^{?} p = q?\ell_2(S').W'$ *implies that* $p = q$ *and* $\ell_1 = \ell_2$ *and* $S = S'$ *and* $W \upharpoonright^{?} p = W'$.

Prefixing receive and non-matching send (resp. send and non-matching receive) actions to a term does not affect its resulting output (resp. input) projection onto a particular participant.

Lemma 3.23. (🐓, 🐓, 🐓, 🐓, 🐓, 🐓)

(1) $\forall p\ q\ \mathcal{A}\ \ell\ S\ W\ W',\ p? \notin \mathcal{A}$ *and* $W \upharpoonright^{?} p = W'$ *implies that* $(q!\ell(S).\mathcal{A}.W) \upharpoonright^{?} p = W'$.
(2) $\forall p\ q\ \mathcal{A}\ \ell\ S\ W\ W',\ p? \notin \mathcal{A},\ p \neq q$ *and* $W \upharpoonright^{?} p = W'$ *implies that* $(q?\ell(S).\mathcal{A}.W) \upharpoonright^{?} p = W'$.
(3) $\forall p\ q\ \mathcal{B}\ \ell\ S\ W\ W',\ p! \notin \mathcal{B}$ *and* $W \upharpoonright^{!} p = W'$ *implies that* $(q?\ell(S).\mathcal{B}.W) \upharpoonright^{!} p = W'$.
(4) $\forall p\ q\ \mathcal{B}\ \ell\ S\ W\ W',\ p! \notin \mathcal{B}$ *and* $p \neq q$ *and* $W \upharpoonright^{!} p = W'$ *implies that* $(q!\ell(S).\mathcal{B}.W) \upharpoonright^{!} p = W'$.
(5) $\forall p\ q\ C\ \ell\ S\ W\ W',\ p? \notin C$ *and* $W \upharpoonright^{?} p = W'$ *implies that* $(q!\ell(S).C.W) \upharpoonright^{?} p = W'$.
(6) $\forall p\ q\ C\ \ell\ S\ W\ W',\ p? \notin C,\ p \neq q$ *and* $W \upharpoonright^{?} p = W'$ *implies that* $(q?\ell(S).C.W) \upharpoonright^{?} p = W'$.

Strict Refinement. We introduce a variant of the refinement relation (Definition 3.5) in which action reordering is not permitted, while payload subsorting is allowed. When combined with SISO projections, strict refinement guarantees that no unintended action reordering occurs between a pair of SISO trees, one refining the other. See Theorem 3.25.

Definition 3.24 (Strict refinement). The strict refinement relation $\sqsubseteq$ over a pair SISO trees W and W' is coinductively defined by the following rules:

$$\frac{\mathsf{S}' \leq: \mathsf{S} \quad \mathsf{W} \sqsubseteq \mathsf{W}'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \sqsubseteq \mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\ [\textsc{str-in}] \qquad \frac{\mathsf{S} \leq: \mathsf{S}' \quad \mathsf{W} \sqsubseteq \mathsf{W}'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \sqsubseteq \mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'}\ [\textsc{str-out}] \qquad \frac{}{\mathsf{end} \sqsubseteq \mathsf{end}}\ [\textsc{str-end}]$$

```
Inductive sRefinementR (R: st → st → Prop): st → st → Prop ≜
  | sref_a  : ∀ w w' p l s s', subsort s' s → R w w' →
                               sRefinementR R (p & [|(l,s,w)|]) (p & [|(l,s',w')|])
  | sref_b  : ∀ w w' p l s s', subsort s s' →  R w w' →
                               sRefinementR R (p ! [|(l,s,w)|]) (p ! [|(l,s',w')|])
  | sref_end: sRefinementR R st_end st_end.

Definition sRefinement w1 w2 ≜ paco2 sRefinementR bot2 w1 w2.
```

We implement strict refinement as the coinductive relation `sRefinement` in Coq. A stream line of actions `w` being strictly refined by another stream `w'` guarantees that all send and receive actions appear in exactly the same order within `w` and `w'`, while allowing for payload subsorting. Crucially, no action reordering is permitted.

Leveraging projections and strict refinement, we show that if $\mathsf{W} \lesssim \mathsf{W}'$ holds, then all send (resp. receive) actions to (resp. from) a particular participant p in W and W' are preserved in exactly the same order within themselves—refinement allows for no order alternation in this sense. This is achieved by showing that output and input projections of W' onto p strictly refine those of W, respectively. This result confirms that the refinement relation behaves as expected. Theorem 3.25 formalises this fact.

Theorem 3.25 (correctness). $\forall \mathsf{p}\ \mathsf{W}\ \mathsf{W}', \quad \mathsf{W} \lesssim \mathsf{W}'$ *implies that* $(\mathsf{W} \upharpoonright^{!} \mathsf{p}) \sqsubseteq (\mathsf{W}' \upharpoonright^{!} \mathsf{p})$ *and* $(\mathsf{W} \upharpoonright^{?} \mathsf{p}) \sqsubseteq (\mathsf{W}' \upharpoonright^{?} \mathsf{p})$.

```
Lemma correctness_dep: ∀ w w' p w1 w2 w3 w4,
  refinement2 (@und w) (@und w') →
  projSC (@und w) p (@und w1) → projSC (@und w') p (@und w2) →
  projRC (@und w) p (@und w3) → projRC (@und w') p (@und w4) →
  sRefinement (@und w1) (@und w2) ∧ sRefinement (@und w3) (@und w4).
```

The proof unfolds within a broader coinductive framework, within which we primarily proceed inductively by means of an inversion lemma (`sinv` below). This lemma enables case analysis on the shapes of SISO terms and guides the inductive reshaping of terms using prefixing facts, refinement inversion facts, and SISO projection facts. These transformations bring the goal into a form where an appropriate constructor of `sRefinement` can be applied, after which the coinduction hypothesis is used to conclude the corresponding case.

Proof. We examine the second argument of the thesis regarding the provably equivalent relation `refinement4`.

```
Lemma correctnessR: ∀ w w' p w1 w2, refinement4 (@und w) (@und w') → projRC (@und w) p (@und w1) →
                    projRC (@und w') p (@und w2) → sRefinement (@und w1) (@und w2).
```

The proof begins by saving the coinduction hypothesis

```
CIH: ∀ (p : string) (w2 : st),  singleton w2 → ∀ w1 : st, singleton w1 → ∀ w' : st, singleton w' →
     ∀ w : st, singleton w → refinement4 w w' → projRC w p w1 → projRC w' p w2 → r w1 w2
```

into the goal context for some relation `r : st → st → Prop` (acts as the knowledge accumulator) over session trees. The goal looks like `paco2 sRefinementR r w1 w2`.

The remainder of the argument hinges on the inversion lemma `sinv`, which characterises the possible structural forms of SISO trees: a SISO tree is either a sequence starting 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).
```

There are nine cases to consider arising from case analyses on `w1` and `w2`, making use of `sinv`, but we focus on the specific case where both begin with a receive action: `w1 = q1 & [|(l1, s1, wa)|]` and `w2 = q2 & [|(l2, s2, wb)|]`. Therefore, in the goal context, we have the following assumptions available:

$$\begin{cases} \texttt{H0: refinement4 w w'}, \\ \texttt{H1: projRC w p (q1 \& [|(l1, s1, wa)|])}, \text{ and} \\ \texttt{H2: projRC w' p (q2 \& [|(l2, s2, wb)|])}. \end{cases}$$

Obviously, the goal takes the form

```
paco2 sRefinementR r (q1 & [|(l1, s1, wa)|]) (q2 & [|(l2, s2, wb)|]).
```

The main idea of the proof is to apply inversion on the hypotheses `H1` and `H2` to derive the shapes of `w` and `w'`, substitute them into `H0`, and extract useful information to discharge the goal(s). Out of `H1` inversion, we have `w` taking three different shapes:

(1) `w = (q1 & [|(l1, s1, w'0)|])` with `p = q1` and `H3: projRC w'0 q1 wa`. Inverting `H2` results in three further cases:

(a) `w' = (q2 & [|(l2, s2, w'1)|])` with `q2 = q1` and `H4: projRC w'1 q2 wb`. By plugging `w` and `w'` into `H0`, and applying Lemmas 3.17(1) and 3.18(1) on two separate copies of `H0`, we respectively obtain:

$$\begin{cases} \texttt{H5: l1 = l2} \text{ and } \texttt{subsort s2 s1}, \text{ and} \\ \texttt{H6: refinement4 w'0 w'1}. \end{cases}$$

We then substitute `l1` with `l2` throughout the context, apply the constructor `sref_a`, and invoke the coinduction hypothesis using `H3`, `H4`, and `H6` on the goal to get it closed. The singletonness cases are straightforward to establish.

(b) `w' = q & [|(l, s, w'1)|]` with `q1 ≠ q`, `H4: projRC w'1 q1 (q2 & [|(l2, s2, wb)|])`. Applying Lemma 3.15(1) to `H0`, with both $\mathcal{A}_1$ and $\mathcal{D}_1$ set to the empty prefix, yields a disjunction: the left yields to a contradiction as the empty prefix cannot be rewritten in terms of a receive prefix merged with some other prefix. The right reveals

$$\begin{cases} \texttt{H5: q \& [|(l, s, w'1)|] = merge\_apf\_cont a3 (q1 \& [|(l1, s3, w3)|])}, \\ \quad \texttt{subsort s3 s1} \text{ and } \texttt{isInA a3 q1 = false}. \end{cases}$$

Applying Lemma 3.14(1) to H5, we obtain

$$\begin{cases} \texttt{H6: w'1 = merge\_apf\_cont a4 (q1 \& [|(l1, s3, w3)|])}, \\ \quad \texttt{a3 = apf\_receive q l s a4} \text{ and } \texttt{isInA a4 q1 = false}. \end{cases}$$

We rewrite w'1 in H4, and apply Lemma 3.22(1) to equate q1 and q2, l1 and l2, and s3 and s2, obtaining H7: projRC w3 q1 wb. These equalities turn the goal into

```
paco2 sRefinementR r (q2 & [|(l2, s1, wa)|]) (q2 & [|(l2, s2, wb)|]).
```

We now rewrite w, w' and w'1 in H0, apply Lemma 3.18(1), and get

$$\{\ \texttt{H8: refinement4 w'0 (q \& [|(l, s, merge\_apf\_cont a4 w3)|])}.$$

Applying the constructor sref_a followed by the coinduction hypothesis CIH on the goal generates the following subgoals (straightforward singletonness are omitted):

```
_____________________________________(1)
refinement4 w'0 (q & [|(l, s, merge_apf_cont a4 w3)|])
_____________________________________(2)
projRC w'0 q2 wa
_____________________________________(3)
projRC (q & [|(l, s, merge_apf_cont a4 w3)|]) q2 wb.
```

The first two are immediate from assumptions H8 and H3, while the last follows from Lemma 3.23(2) and H7.

(c) w' = q ! [|(l, s, w'1)|] with p = q1 and coseqIn (q1, rcv) (act w'1). Rewriting w and w' into H0 leads to a contradiction by Lemma 3.19.

(2) w = q & [|(l, s, w'0)|] with p ≠ q, H3: projRC w'0 p (q1 & [|(l1, s1, wa)|]) and H4: coseqIn (p, rcv) (act w'0). We invert H2 and have three cases to show:

(a) w' = q2 & [|(l2, s2, w'1)|] with p = q2, q2 ≠ q and H5: projRC w'1 q2 wb. Making use of Lemma 3.7 (non-parametric variant 🐓) with H4, we obtain

$$\begin{cases} \texttt{H6: w'0 = merge\_cpf\_cont c3 (q2 \& [|(l3, s3, w3)|])}, \\ \quad \texttt{isInC c3 q2 = false}. \end{cases}$$

Rewriting w'0 in H3, and applying Lemma 3.22(3) equates q1 and q2, l1 and l3, and s1 and s3, and further reveals H7: projRC w3 q1 wa. We plug w, w'0, and w' into H0, and apply Lemmas 3.17(3) and 3.18(3) sequentially on two copies of H0 to respectively obtain:

$$\begin{cases} \texttt{H8: l1 = l2} \text{ and } \texttt{subsort s2 s1}, \text{ and} \\ \texttt{H9: refinement4 (q \& [|(l, s, merge\_cpf\_cont c3 w3)|]) w'1}. \end{cases}$$

These equalities turn the goal into

```
paco2 sRefinementR r (q1 & [|(l2, s1, wa)|]) (q1 & [|(l2, s2, wb)|]).
```

Applying the constructor sref_a followed by the coinduction hypothesis CIH on the goal generates the following subgoals:

```
_____________________________________(1)
refinement4 (q & [|(l, s, merge_cpf_cont c3 w3)|]) w'1
_____________________________________(2)
projRC (q & [|(l, s, merge_cpf_cont c3 w3)|]) q1 wa
```

```
_____________________________________(3)
projRC w'1 q1 wb
```

The first and the third goals are immediate from assumptions H9 and H5, while the second follows from Lemma 3.23(6) and H7.

(b) `w' = q0 & [|(l0, s0, w'1)|]` with `p ≠ q0`, `H5: projRC w'1 p (q2 & [|(l2, s2, wb)|])` and `H6: coseqIn (p, rcv) (act w'1)`. Employing Lemma 3.7 (non-parametric variant 🏃) with H4 and H6, we obtain

$$\begin{cases} \texttt{H7: w'0 = merge\_cpf\_cont c3 (p \& [|(l3, s3, w3)|]),} \\ \quad \texttt{isInC c3 p = false} \\ \texttt{H8: w'1 = merge\_cpf\_cont c6 (p \& [|(l6, s6, w6)|]),} \\ \quad \texttt{isInC c6 p = false.} \end{cases}$$

Substituting `w'0` in H3 and `w'1` in H5 and applying Lemma 3.22(3) yields `p = q1 = q2`, `l3 = l1`, `s3 = s1`, `l6 = l2`, `s6 = s2`, `H9: projRC w3 p wa` and `H10: projRC w6 p wb`. We rewrite `w`, `w'0`, `w'`, and `w'1` in H0, and apply Lemmas 3.17(3) and 3.18(3) on two separate copies of H0, which respectively yield:

$$\begin{cases} \texttt{H11: l1 = l2} \text{ and } \texttt{subsort s2 s1}, \text{ and} \\ \texttt{H12: refinement4 (q \& [|(l, s, merge\_cpf\_cont c3 w3)|])} \\ \qquad\qquad\qquad \texttt{(q0 \& [|(l0, s0, merge\_cpf\_cont c6 w6)|]).} \end{cases}$$

These equalities turn the goal into

```
paco2 sRefinementR r (q1 & [|(l2, s1, wa)|]) (q1 & [|(l2, s2, wb)|]).
```

The constructor `sref_a` and the coinduction hypothesis `CIH` on the goal generates three subgoals:

```
_____________________________________(1)
refinement4 (q & [|(l, s, merge_cpf_cont c3 w3)|]) (q0 & [|(l0, s0, merge_cpf_cont c6 w6)|])
_____________________________________(2)
projRC (q & [|(l, s, merge_cpf_cont c3 w3)|]) q2 wa
_____________________________________(3)
projRC (q0 & [|(l0, s0, merge_cpf_cont c6 w6)|]) q2 wb.
```

The first one trivially follows from H12 while the second and third ones follow from Lemma 3.23(6), H9 and Lemma 3.23(6), H10 respectively.

(c) `w' = q0 ! [|(l0, s0, w'1)|]` with `H4: projRC w'1 p (q2 & [|(l2, s2, wb)|])` and `H5: coseqIn (p, rcv) (act w'1)`. Rewriting `w` and `w'` into H0 leads to a contradiction by Lemma 3.19.

(3) `w = (q ! [|(l, s, w'0)|])` with `H3: projRC w'0 p (q1 & [|(l1, s1, wa)|])` and also `H4: coseqIn (p, rcv) (act w'0)`. We again invert H2 and have three cases to demonstrate:

(a) `w' = (q2 & [|(l2, s2, w'1)|])` with `p = q2` and `H5: projRC w'1 q2 wb`. This case proceeds along the exact same lines as in (2)-(a), with the sole difference that, at the end of the proof, after applying the coinduction hypothesis, we invoke Lemma 3.23(5) instead to close the corresponding goal, since the term `w` begins with a send action.

(b) `w' = (q0 & [|(l0,s0,w'1)|])` with `p ≠ q0`, `H5: projRC w'1 p (q2 & [|(l2,s2,wb)|])` and `H6: coseqIn (p, rcv) (act w'1)`. This case follows exact similar lines with those of (2)-(b). Just that at the end, after making use of the coinduction hypothesis, we

employ Lemma 3.23(5) instead to close the corresponding goal, as the term `w` begins with a send action.

(c) `w' = q0 ! [|(l0, s0, w'1)|]` with `H5: projRC w'1 p (q2 & [|(l2, s2, wb)|])` and `H6: coseqIn (p, rcv) (act w'1)`. This case follows the exact same lines as in (2)-(b). The only difference arises at the end: after applying the coinduction hypothesis, we use Lemma 3.23(5) to discharge two subgoals, since both terms `w` and `w'` begin with send actions.

The remaining eight cases either proceed by similar reasoning or lead to contradictions, and therefore hold vacuously. Leveraging the equivalence `refEquiv24`, we thereby establish the correctness proof for `refinement2`. □

4 ASYNCHRONOUS SUBTYPING AND NEGATIONS

Intuitively, T is said to be an asynchronous subtype of T' if, for every single output decomposition U of the tree T and every single input decomposition V' of the tree T', there exist action streams W and W'—respectively extracted from the single input decomposition of U and the single output decomposition of V'—such that W' refines W. In other words, within the supertype, one can always find action streams that align with the subtype via appropriate reorderings, as governed by the prefixing contexts $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$.

Definition 4.1. The asynchronous subtyping relation $\leqslant$ over session trees is defined as:

$$\frac{\forall \mathsf{U},\ \mathsf{U} \lhd_{\mathrm{SO}} \mathsf{T} \qquad \forall \mathsf{V}',\ \mathsf{V}' \lhd_{\mathrm{SI}} \mathsf{T}' \qquad \exists \mathsf{W},\ \mathsf{W} \lhd_{\mathrm{SI}} \mathsf{U} \qquad \exists \mathsf{W}',\ \mathsf{W}' \lhd_{\mathrm{SO}} \mathsf{V}' \qquad \mathsf{W} \lesssim \mathsf{W}'}{\mathsf{T} \leqslant \mathsf{T}'}$$

The translation function lifts the subtyping relation to the level of types: $\mathbb{T} \leqslant \mathbb{T}'$ if and only if $\mathbb{T} \xrightarrow{\mathcal{T}} \mathsf{T} \leqslant \mathsf{T}' \xleftarrow{\mathcal{T}} \mathbb{T}'$. We formalise subtyping in Coq as follows :

```
Inductive subtypeI: st → st → Prop ≜
  | stc: ∀ T T', (∀ U, st2soC U T → ∀ V', st2siC V' T' →
         (∃ W W', st2sisoC (@und W) U ∧ st2sisoC (@und W') V' ∧ (@und W) ~< (@und W'))) →
         subtypeI T T'.

Definition subltype (T T': local) ≜ subtypeI (lt2st T) (lt2st T').
```

Note that asynchronous subtyping is undecidable [7, 38] for both binary and multiparty session types; as such, it cannot be implemented as a total decision procedure. For this reason, it is axiomatised (together with the underlying refinement) as a relation in Coq.

4.1 Negation of Refinement

The negation of the refinement relation, denoted $\not\leqslant$, over SISO trees captures the structural properties of trees that cannot refine each other. This relation is inductively defined as the counterpart of the coinductive refinement. It provides a framework for the complement of subtyping within the context of session trees and, by extension, session types. Initially consisting of eighteen inductively defined rules, as outlined in [22, Fig. 6], we have reduced the set by eliminating ten rules. The revised set of rules is presented in Figure 3 . Completeness of refinement, and thus subtyping, with respect to negations is discussed in § 5.

The rule [N-ACT] states that if a pair of trees do not share the same set of actions (as defined in Definition 3.10), then they cannot refine each other. In Coq, we proved that such terms cannot belong to the refinement relation; see Lemma 3.6. With this rule, we explicitly include such cases in

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

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

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

Fig. 3. The negation of the refinement relation $\not\leqslant$ over SISO trees.

the negation of refinement. Notably, the rule [N-ACT] subsumes and effectively proves four separate rules given in the original definition:

Lemma 4.2. $\forall \mathsf{p}\ \ell\ \mathsf{S}\ \mathsf{W}\ \mathsf{W}'$,
(1) $\mathsf{p}! \notin \mathsf{act}(\mathsf{W}') \implies \mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathsf{W}'$ (2) $\mathsf{p}? \notin \mathsf{act}(\mathsf{W}') \implies \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathsf{W}'$
(3) $\mathsf{p}! \notin \mathsf{act}(\mathsf{W}) \implies \mathsf{W} \not\leqslant \mathsf{p}!\ell(\mathsf{S}).\mathsf{W}'$ (4) $\mathsf{p}? \notin \mathsf{act}(\mathsf{W}) \implies \mathsf{W} \not\leqslant \mathsf{p}?\ell(\mathsf{S}).\mathsf{W}'$

The rule [N-I-O-2] states that, in a pair of terms, if the first term begins with a receive action from a fixed participant p, it cannot refine the second term if the latter contains an arbitrary send action occurring before any receive action from p. This restriction stems from the fact that the actions in the second term cannot be reordered to bring a p? action to the front as the leftmost action.

Another crucial aspect of this rule is the revision of its structure compared to the version presented in the original definition:

original definition		reformulated shape
$\dfrac{}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathcal{A}^{(\mathsf{p})}.\mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'}$	$\implies$	$\dfrac{\mathsf{q}! \in C^{(\mathsf{p})}}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\leqslant C^{(\mathsf{p})}.\mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'}$

This revision is beneficial because the rule now aligns structurally with the other rules, making it more consistent and easier to apply. In particular, the right-hand side takes on the general form of terms with receive actions, which directly connects to Lemma 3.7. Furthermore, Lemma 3.9 becomes applicable when the premise of the rule does not hold. In addition, this revised formulation also allows one of the rules from the original definition, [N-I-O-1], to be derived with the help of [N-ACT]:

Lemma 4.3 ([N-I-O-1]). $\forall \mathsf{p}\ \mathsf{q}\ \ell\ \ell'\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}', \quad \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'.$

The last six rules handle subtle cases involving asynchronous reorderings. The rule [N-$\mathcal{A}$-ℓ] states that terms with mismatching labels cannot refine each other, even when $\mathcal{A}^{(\mathsf{p})}$ sort of prefixing permits valid reordering. The rules [N-$\mathcal{A}$-S] and [N-$\mathcal{A}$-W] are variants that address mismatches in sorts and continuations, respectively. The final three rules in this group apply similar reasoning, but for terms involving $\mathcal{B}^{(\mathsf{p})}$ sort of prefixing.

Remark 7. *We prove six additional rules from the original definition of the negation relation simply by allowing the prefix sorts $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ to include the empty prefix ε. In Lemma 4.4, we only state the instances related to [N-$\mathcal{A}$-ℓ] and [N-$\mathcal{B}$-ℓ], as the remaining cases follow analogously.*

Lemma 4.4. $\forall \mathsf{p}\ \ell\ \ell'\ \mathsf{S}\ \mathsf{S}'\ \mathsf{W}\ \mathsf{W}'$,
(1) $\ell \neq \ell' \implies \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'$ (2) $\ell \neq \ell' \implies \mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\leqslant \mathsf{p}!\ell'(\mathsf{S}').\mathsf{W}'$

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 relation includes the constructors `n_b_l` and `n_b_w` (omitted in the snippet), which act as alternatives to the rules `n_a_l` and `n_a_w`, respectively, with $\mathcal{B}^{(\mathsf{p})}$ style prefixing.

We develop the relation in Coq over SISO trees, so the proof obligation `singleton` must be satisfied each time a session tree is used in this context. The parameters `P`, `Q`, and `R` represent the corresponding instances of these proofs.

With these essential components in place, we are now ready to define the negation of the subtyping relation, with the refinement negation $\not\lesssim$ serving as its basis.

Definition 4.5 (negation of subtyping). For any pair of session trees T, T', $\mathsf{T} \not\leqslant \mathsf{T}' \triangleq \exists \mathsf{U},\ \mathsf{U} \lhd_{\mathsf{SO}} \mathsf{T}$ and $\exists \mathsf{V}',\ \mathsf{V}' \lhd_{\mathsf{SI}} \mathsf{T}'$ and $\forall \mathsf{W},\ \mathsf{W} \lhd_{\mathsf{SI}} \mathsf{U}$ entails $\forall \mathsf{W}',\ \mathsf{W}' \lhd_{\mathsf{SO}} \mathsf{V}'$ entails $\mathsf{W} \not\lesssim \mathsf{W}'$.

```
Definition nsubtypeI (T T': st): Prop ≜
  ∃ U,  (st2soC U T) ∧  ∃ V', (st2siC V' T') ∧
  (∀ W W', st2sisoC (@und W) U → st2sisoC (@und W') V' → nRefinement W W').
```

We can, of course, lift it to the level of session types:

```
Definition nsubltypeI (T T': local): Prop ≜ nsubtypeI (lt2st T) (lt2st T').
```

5 COMPLETENESS

Completeness is the primary meta-property of the subtyping relation (with respect to negation) that we have successfully formulated and verified in Coq. Essentially, it asserts that for any pair of session trees T and T', either T is a subtype of T' or it is related to T' by the negation of the subtyping relation, leaving no room for a third possibility. The proof of subtyping completeness hinges on the completeness of the refinement relation, as formally outlined below.

Lemma 5.1 (refinement completeness). *For any pair of SISO trees* W, W', *we have*

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

As in the proof of Theorem 3.25, the argument proceeds within a coinductive schema. We transform the term structures in the goal to enable the application of `refinement2` constructors, and then invoke the coinduction hypothesis to discharge the corresponding cases.

Proof. ($\Rightarrow$) To establish the left-to-right implication, we initially prove $\neg(W \not\lesssim W') \implies W \lesssim W'$, 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 an 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 `w = (p & [|(l,s,w1)|])` and `w' = (q & [|(l',s',w2)|])`.

(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 3.7.
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 3.9, 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

```
paco2 refinementR2 r w1 (merge_ap_cont p a w3) ∨ r w1 (merge_ap_cont p a w3)
```

To close the subgoal, we do not further fold the coinductive relation `refinementR2`, and instead employ the coinduction hypothesis `CIH` on the right side of the disjunction. 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 ≜ 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 ≠ l'` or `s'` is not a subsort of `s` are handled by rules `n_a_l` and `n_a_s`. □

The statement is proved by structural induction on the given instance of the `nRefinement` relation. We then invert the `refinement2` hypothesis to extract relevant information and derive a contradiction.

Proof. ($\Leftarrow$) 🐓. 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).
```

such that `s` is not a subsort of `s'`. Inverting `H` yields `False` under the following additional hypothesis.

```
Ha: merge_bp_contn p b0 (p![(l,s'0,w'0)]) n0 = merge_bp_contn p b (p![(l,s',w')]) n.
```

The proof of the falsity is somewhat 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.*

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

The function `BpnA` appends the given prefix `b` to itself `n` times. Recall that the function `merge_bp_cont` is a special case of `merge_bp_contn` with `n` set to 1.

We rewrite the lemma `meqBp` 🐓 in `Ha` and transform it into

```
Ha: merge_bp_cont p (BpnA p b0 n0) (p ! [|(l,s'0,w'0)|]) =
    merge_bp_cont p (BpnA 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 3.12(2).

We can now close the goal simply by plugging `s'0 = 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 5.2 (subtyping completeness). 🐓 *For any pair session trees* T, T'*, we have*

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

Proof. Follows from Lemma 5.1. □

```
Lemma sublNeqL: ∀ T T', (subltypeI T T' → False) → nsubltypeI T T'.
Lemma sublNeqR: ∀ T T', nsubltypeI T T' → (subltypeI T T' → False).

Theorem lcompletenessI: ∀ T T', (subltypeI T T' → False) ↔ nsubltypeI T T'.
Proof. split; [ apply (sublNeqL T T') | intros; apply (sublNeqR T T'); easy]. Qed.
```

6 SUBTYPING AT WORK

In order to verify that $\mathsf{T} \leqslant \mathsf{T}'$ holds for arbitrary trees T and T′ in Coq, we are supposed to manually provide the SISO decompositions (W, W′), since decompositions are defined via relations rather than corecursive functions. Additionally, we require a mechanism for checking whether the resulting decompositions exhibit the same actions, as the inductive property in Definition 3.10, while suitable for proving facts, is not directly usable for such actual checks.

In § 6.1, we present our approach for verifying action equality between pairs of SISO trees and revisit the refinement and subtyping relations accordingly. In § 6.2, we prove that refinement is a transitive relation. Finally, in § 6.3 and § 6.4, we present subtyping examples from the literature to showcase the use of our Coq tool.

6.1 Refinement and Subtyping Revisited

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 → 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).
```

Since every session tree is derived from a session type, we proceed under the assumption that the streams of actions extracted from these trees satisfy a reasonable notion of `finiteness`—that is, they contain only a finite set of distinct actions. This assumption is naturally aligned with the *inductive* framework of multiparty session types. Even when sessions involve infinitely many interactions, these interactions necessarily take place among a finite number of participants [21, 22], resulting in a finite number of distinct actions. Consequently, we can assert that a pair of streams s_1 and s_2 have identical members if and only if there exist lists l_1 and l_2 such that l_1 is involved in s_1 and, conversely, s_1 is involved in l_1, and likewise, l_2 is involved in s_2 and s_2 is involved in l_2, with l_1 and l_2 containing the same members. Rather than stating this as an explicit axiom in Coq, we incorporate it into the action equality check defined in Definition 6.1.

Definition 6.1. For a pair of SISO trees W and W′, we define action equality as follows:

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

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:

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

Recall that the symbol $\in$ denotes the inductive stream membership check, `coseqIn`, whereas `In` refers to the standard list membership check.

Lemma 6.2. *Given* $\mathrm{W} := \mathsf{p}!\ell_1(\mathrm{S}_1).\mathsf{p}?\ell_2(\mathrm{S}_2).\mathsf{q}!\ell_3(\mathrm{S}_3).\mathrm{W}$ *and* $l := \mathsf{p}? :: \mathsf{p}! :: \mathsf{q}! :: nil$, *we have* (1) $\mathtt{act}(\mathrm{W}) \in_C l$ *and* (2) $l \in_I \mathtt{act}(\mathrm{W})$.

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 $\mathtt{act}(\mathrm{W})$. 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 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, 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.
```

Remark 8. *In Coq, there is no general method to bridge the gap between the action equality checks given in Definitions 6.1 and 3.10. One possible strategy involves assuming that*

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

holds, where $\in_C^I$ *denotes the inductively defined counterpart of* $\in_C$. *This would allow us to deduce that the property in Definition 6.1 implies the one in Definition 3.10. However, adopting this implication would lead to inconsistency, as the inductive predicate* $\in_C^I$ *enforces finiteness on its first argument, whereas* $\in_C$ *may hold for infinite streams.*

Therefore, implication (1) cannot be treated as an instance of the coinductive extensionality *(*`cext`*) principle, an example of which is given in [1, Appendix B]. There,* `cext` *is used to derive Leibniz equality from the bisimulation* $=_{\mathbb{N}^\infty}$ *over conats. This works since conats modulo* $=_{\mathbb{N}^\infty}$ *are isomorphic to* $\mathbb{N} + 1$.

In our setting, such an isomorphism is not available, as we work with a non-structural notion of equality over streams of actions. To address this, we introduce a new refinement (and subtyping) relation, `refinement`, *which differs from* `refinement2` *only in how it performs action equality checks. Under the* `finiteness` *assumption—that is, "in a session with a potentially infinite number of interactions, there can only be finitely many distinct actions"—these relations effectively serve the same purpose.*

Note also that in the rest of the paper, we overload the symbol $\lesssim$ *to denote the refinement relation (*`refinement`*) based on the check given in Definition 6.1.*

It is important to note that this strategy, with the new action equality check, becomes cumbersome when attempting to prove the completeness of subtyping. A limitation arises because the approach is not well-suited for establishing general properties about tree shapes with prefixes, as discussed in § 3.3, particularly in Lemmata 3.7 and 3.8. However, it proves effective for verifying the subtyping examples presented in § 6.3 and 6.4, as it allows us to store the actions of specific trees in finite lists and reduce action equality to list membership.

Below, we introduce an updated `refinement` relation within the Coq development. The relevant code for the action equality check is located under the existential quantifiers ∃.

```
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.
```

We also revisit the original subtyping Definition 4.1 (see [22, Def. 3.13]), $\mathsf{T} \leqslant \mathsf{T}'$, by modifying it to rely on the direct decomposition of T and T' into SISO trees, as in Definition 3.2. The central idea is to introduce a list of SISO tree pairs $(\mathsf{W}_i, \mathsf{W}'_i)$, where each W_i is drawn from the SISO decomposition of T and each W'_i from that of T'. We then check whether each W_i refines the corresponding W'_i under the updated refinement relation.

Definition 6.3. Revisited asynchronous subtyping relation $\leqslant$ over session trees is defined as:

$$\frac{\exists\{(\mathsf{W}_i, \mathsf{W}'_i) \mid i \in I\} \qquad \mathsf{W}_i \lhd_{\text{SISO}} \mathsf{T} \qquad \mathsf{W}'_i \lhd_{\text{SISO}} \mathsf{T}' \qquad \mathsf{W}_i \lesssim \mathsf{W}'_i}{\mathsf{T} \leqslant \mathsf{T}'}$$

for some finite set of indices I.

This approach has a drawback: one might fail to provide all the necessary SISO decompositions required to complete subtyping proofs. However, the existentially quantified list of SISO tree pairs can always be extended with additional pairs as needed to complete the proof. Notably, the crux of the method lies in the underlying refinement relation and the correctness of the associated checks.

```
Definition subtype (T T': st): Prop ≜ ∃ (l: list (siso*siso)), decomposeL l T T' ∧ listSisoPRef l.
Definition subltype (T T': local) ≜ subtype (lt2st T) (lt2st T').
```

The function `decomposeL` takes a list `l` of SISO tree pairs and the trees `T` and `T'`, and outputs the proposition: "for each pair in the list, the first tree is obtained from the decomposition of T, and the second from the decomposition of T'", for verification purposes. The function `listSisoPRef` performs similarly, transforming a list `l` of SISO tree pairs into refinement checks (using the `refinement` relation) between the first and second components of each pair, with certain terms prefixed when necessary.

6.2 Transitivity of the Refinement Relation

Transitivity of the refinement relation is a crucial property that often arises when checking asynchronous session subtyping in Coq (cf. the example in § 6.4). In this section, we outline the core elements of the transitivity proof. As in § 3.4, we start by introducing a new refinement relation, `refinement3`, which differs from the original `refinement` relation only in its use of non-parametric prefix types—$\mathcal{A}$ (`Apf`) (and $\mathcal{B}$ (`Bpf`))—along with the associated glue functions, e.g., `merge_apf_contn`.

```
Inductive refinementR3 (R: st → st → Prop): st → st → Prop ≜
  | ref3_a  : ∀ w w' p l s s' a n, subsort s s' → isInA a p = false → seq w (merge_apf_contn 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_apf_contn a w' n)) ∧
```

```
                          (∀ x, List.In x L1 ↔ List.In x L2) ) →
            refinementR3 R (st_receive p [(l,s,w)]) (merge_apf_contn a (st_receive p [(l,s',w')]) n)
  | ...

Definition refinement3: st → st → Prop ≜ fun s1 s2 ⇒ paco2 refinementR3 bot2 s1 s2.
```

We show that `refinement` and `refinement3` are equivalent relations .

```
Lemma refEquiv13: ∀ w w', refinement w w' ↔ refinement3 w w'.
```

The relations `refinement` and `refinement3` are distinguished from one another in exactly the same manner as `refinement2` and `refinement4`, as summarised in Table 1. In particular, the difference between each pair lies solely in the treatment of action prefixing and the associated reasoning principles, while their underlying refinement semantics coincide. Accordingly, we do not provide a separate table contrasting `refinement` and `refinement3`, as it would be identical to Table 1.

By contrast, the distinction between `refinement` and `refinement2` concerns the way action equality is checked: `refinement` employs an explicit action equality relation under a finiteness assumption, whereas `refinement2` relies on a more abstract treatment of action comparison. Given this finiteness assumption—which is reasonable in the domain of MPST—the relations `refinement` and `refinement2` coincide.

Modifying the underlying action equality check in the refinement relation, according to Definition 6.1, requires only minimal changes to the Coq proof of Lemma 3.18: obviously at points where action equalities between terms must be established. More generally, this modification necessitates reproving Lemma 3.16, which can be achieved using the following properties: (1) if x is a member of a colist s and every member of s is a member of some list l, then x is in l; (2) if x is a member of a list l and every member of l is a member of some colist s, then x is in s.

Lemma 6.4. (,)

(1) $\forall x\ l\ s, x \in s$ *and* $s \in_C l$ *implies that* x `In` l
(2) $\forall x\ l\ s, x$ `In` l *and* $l \in_I s$ *implies that* $x \in s$

The proofs of Lemmas 3.15 and 3.17 remain unchanged, except that we must employ the monotonicity of the generating relation `refinementR3` instead of that of `refinementR4` wherever necessary.

Note 9. *In what follows,*

- *we continue to refer to the aforementioned lemmas, with the understanding that their proofs have been adjusted accordingly;*
- *we show that the transitivity of refinement holds for any session tree by definition; we do not restrict ourselves to SISO trees.*

Theorem 6.5 (transitivity). $\forall W_1\ W_2\ W_3, \quad W_1 \lesssim W_2$ *and* $W_2 \lesssim W_3$ *implies that* $W_1 \lesssim W_3$.

```
Lemma Ref_Trans: ∀ w1 w2 w3: st, refinement w1 w2 → refinement w2 w3 → refinement w1 w3.
```

In a similar way to the proof of Theorem 3.25, the proof relies on the equivalent relation `refinement3`. We coinductively show that the statement holds, recording the initial state as the coinduction hypothesis. We then reshape the terms in the goal to allow the application of constructors of `refinement3`. Finally, we invoke the coinduction hypothesis to close the corresponding case.

Proof. The proof begins by introducing the coinduction hypothesis

```
CIH : ∀ x y z : st, refinement3 x y → refinement3 y z → r x z
```

into the goal context for a relation `r: st → st → Prop`, which serves as a knowledge accumulator (over session trees). Now, the goal is to establish `paco2 refinementR3 r x z`.

In principle, we mutually invert the refinement assumptions (`refinement3 x y` and `refinement3 y z`), resulting in nine cases to consider—three for each constructor (`ref3_a`, `ref3_b`, and `ref3_end`). Here, we focus on the "`ref3_b`-`ref3_b`" case, where we have following assumptions:

```
H1: refinement3 w (merge_bpf_cont (BpnB3 b n) w'),
H2: refinement3 w0 (merge_bpf_cont (BpnB3 b0 n0) w'0),
H3: act_eq w (merge_bpf_cont (BpnB3 b n) w'),
H4: act_eq w0 (merge_bpf_cont (BpnB3 b0 n0) w'0),
H5: p0 ! [|(l0, s0, w0)|] = merge_bpf_cont (BpnB3 b n) (p ! [|(l, s', w')|]),
H6: subsort s s' and subsort s0 s'0, and
H7: isInB b p = false and isInB b0 p0 = false.
```

And, the goal takes below form:

```
paco2 refinementR3 r (p ! [|(l, s, w)|]) (merge_bpf_cont (BpnB3 b0 n0) (p0 ! [|(l0, s'0, w'0)|])).
```

The function `BpnB3` 🐓 takes a prefix `b: Bpf` along with a natural number `n`, and concatenates `b` with itself `n` times.

We proceed with a case distinction over whether `p` equals `p0`:

(1) `p = p0`. Thanks to Lemma 3.11 and hypothesis `H5`, we obtain `l0 = l`, `s0 = s'`, and `w0 = w'`, along with `(BpnB3 b n)` being an end prefix. We then rewrite these equalities in the goal and apply the constructor `ref3_b`, which results in three subgoals:

```
_____________________________________(1)
subsort s s'0
_____________________________________(2)
paco2 refinementR3 r w (merge_bpf_cont (BpnB3 b0 n0) w'0) ∨ r w (merge_bpf_cont (BpnB3 b0 n0) w'0)
_____________________________________(3)
act_eq w (act (merge_bpf_cont (BpnB3 b0 n0) w'0)).
```

The first subgoal is resolved using the transitivity of the `subsort` relation and hypothesis `H6`. For the second (coinductive) goal, we refrain from folding the coinductive relation further and instead apply the hypothesis `CIH` (to the right side of disjunction) with `w2` ≜ `w'`.
The third goal concerns action equalities, which is somewhat tedious. We detail it here for reference, omitting similar proofs in other cases. We extract the following list of premises

```
H3a: coseqInLC (act w') l1,
H3b: coseqInLC (act (merge_bpf_cont (BpnB3 b0 n0) w'0)) l2,
H3c: coseqInR l1 (act w'),
H3d: coseqInR l2 (act (merge_bpf_cont (BpnB3 b0 n0) w'0)),
H3e: ∀ x : string * dir, In x l1 ↔ In x l2,
H4a: coseqInLC (act w) l3,
H4b: coseqInLC (act w') l4,
H4c: coseqInR l3 (act w),
H4d: coseqInR l4 (act w'),
H4e: ∀ x : string * dir, In x l3 ↔ In x l4,
```

from the unfolding and proper destruction of hypotheses H3 and H4. The goal in this case takes the below form:

```
∃ L1 L2 : seq.seq (string * dir),
  coseqInLC (act w) L1 ∧
  coseqInLC (act (merge_bpf_cont (BpnB3 b0 n0) w'0)) L2 ∧
  coseqInR L1 (act w) ∧
  coseqInR L2 (act (merge_bpf_cont (BpnB3 b0 n0) w'0)) ∧ (∀ x : string * dir, In x L1 ↔ In x L2).
```

We plug l3 and l2 into the goal as existential arguments, which allows us to trivially close the first four components of the goal using the premises H4a, H3b, H4c, and H3d, respectively. To solve the left-to-right direction of the remaining goal

```
∀ x : participant * dir, In x l3 ↔ In x l2
```

we assume `H8: In x l4` (using H4e) and try to close `In x l1` (thanks to H3e) as follows: From H8, H4d, and Lemma 6.4(2), we obtain `H9: coseqIn x (act w')`. Then, using H9 and H3a, the goal is resolved by applying Lemma 6.4(1). The right-to-left direction follows a similar line of reasoning.

(2) `p ≠ p0`. We apply Lemma 3.14(2) to H7 and obtain:

$$\begin{cases} \texttt{H8: w0 = merge\_bpf\_cont b1 (p ! [|(l, s', w')|])}, \\ \texttt{BpnB3 b n = bpf\_send p0 l0 s0 b1} \text{ and } \texttt{isInB b1 p = false}. \end{cases}$$

Next, applying Lemma 3.15 to H2 after rewriting w0 leads to a case split:

(a) Given

$$\begin{cases} \texttt{H2a: BpnB3 b0 n0 = Bpf\_merge b2 (bpf\_send p l s2 b3)}, \\ \texttt{subsort s' s2} \text{ and } \texttt{isInB b2 p = false}. \end{cases}$$

> bpf_send is one of the constructors of the Bpf type. The function Bpf_merge concatenates two prefixes of type Bpf.

We rewrite `BpnB3 b0 n0` (due to H2a) in the goal and shape it into the following form:

```
paco2 refinementR3 r (p ! [|(l, s, w)|])
            (merge_bpf_cont b2 (p ! [|(l, s2, merge_bpf_cont b3 (p0 ! [|(l0, s'0, w'0)|]))|]))
```

We then apply the constructor `ref3_b` to the goal, resulting in the following subgoal:

```
paco2 refinementR3 r w (Bpf_merge b2 b3) (p0 ! [|(l0, s'0, w'0)|]) ∨
r w (Bpf_merge b2 b3) (p0 ! [|(l0, s'0, w'0)|])
```

omitting those considering subsorting and action equalities. We do not fold the coinductive relation in the goal, instead apply CIH (to the right side of disjunction) with `w2 ≜ (merge_bpf_cont (BpnB3 b n) w')`. This leads to two further subgoals:

```
_____________________________________(1)
refinement3 w (merge_bpf_cont (BpnB3 b n) w')
_____________________________________(2)
refinement3 (merge_bpf_cont (BpnB3 b n) w')
            (merge_bpf_cont (Bpf_merge b2 b3) (p0 ! [|(l0, s'0, w'0)|]))
```

The first one trivially follows from H1. We now focus on the second subgoal. We begin by rewriting BpnB3 b n (using H8) inside the goal and again apply the constructor ref3_b. After dealing with subsorting and action equality goals, we are left with:

```
paco2 refinementR3 bot2 (merge_bpf_cont b1 w') (merge_bpf_cont (Bpf_merge b2 b3) w'0) ∨
bot2 (merge_bpf_cont b1 w') (merge_bpf_cont (Bpf_merge b2 b3) w'0)
```

The left side of disjunction is closed by rewriting w0 (using H8) and BpnB3 b0 n0 (using H2a) into H2 and then applying Lemma 3.18(2) to it.

(b) Given

```
H2a: w'0 = merge_bpf_cont b2 (p ! [|(l, s2, w2)|]),
     subsort s' s2 and isInB (BpnB3 b0 n0) p = false.
```

We start by rewriting w'0 (using H2a) inside the goal, and reshape it into the following:

```
paco2 refinementR3 r (p ! [|(l, s, w)|])
     (merge_bpf_cont (Bpf_merge (BpnB3 b0 n0) (bpf_send p0 l0 s'0 b2)) (p ! [|(l, s2, w2)|]))
```

Applying the constructor over the goal, transforms it into

```
paco2 refinementR3 r w (merge_bpf_cont (Bpf_merge (BpnB3 b0 n0) (bpf_send p0 l0 s'0 b2)) w2) ∨
r w (merge_bpf_cont (Bpf_merge (BpnB3 b0 n0) (bpf_send p0 l0 s'0 b2)) w2)
```

At this stage, we employ CIH (at the right side of disjunction) with w2 ≜ (merge_bpf_cont (BpnB3 b n) w') which leads to two further subgoals:

```
_____________________________________(1)
refinement3 w (merge_bpf_cont (BpnB3 b n) w')
_____________________________________(2)
refinement3 (merge_bpf_cont (BpnB3 b n) w')
  (merge_bpf_cont (Bpf_merge (BpnB3 b0 n0) (bpf_send p0 l0 s'0 b2)) w2).
```

The first subgoal is closed by H1. Now, we focus on the second subgoal. We start by rewriting BpnB3 b n (using H8) within the goal and then apply the constructor ref3_b once again. After addressing the subsorting and action equality cases, we are supposed to show:

```
paco2 refinementR3 bot2 (merge_bpf_cont b1 w')
                        (merge_bpf_cont (BpnB3 b0 n0) (merge_bpf_cont b2 w2)) ∨
bot2 (merge_bpf_cont b1 w') (merge_bpf_cont (BpnB3 b0 n0) (merge_bpf_cont b2 w2)).
```

The left side of the disjunction is closed by first rewriting w0 and w'0 inside H2, using H8 and H2a, respectively, and then applying Lemma 3.18(2) to it.

The remaining eight cases either follow similar lines of reasoning or result in contradictory assumptions, and thus hold vacuously. By virtue of the equivalence refEquiv13, the property extends to refinement as well. □

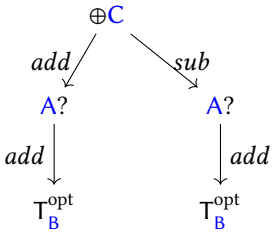
6.3 Ring Choice Example

This subsection presents the details of the ring choice example introduced in Figure 1, where the main objective is to prove and verify—using the accompanying Coq library—that $\mathbb{T}_B^{\text{opt}} \leqslant \mathbb{T}_B$ holds. The complete proof can be found in the file `ring_choice.v` .

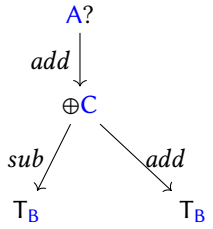
```
Definition TyBOp ≜ lt_mu (lt_send "C" [("add", I, lt_receive "A" [("add", I, (lt_var 0))]);
                                       ("sub", I, lt_receive "A" [("add", I, (lt_var 0))])]).
Definition TyB ≜ lt_mu (lt_receive "A" [("add", I, lt_send "C" [("add", I, (lt_var 0));
                                                                ("sub", I, (lt_var 0))])]).
```

We begin by translating these types into their corresponding trees, $\mathbb{T}_B^{\text{opt}} \xrightarrow{\mathcal{T}} \mathsf{T}_B^{\text{opt}}$ and $\mathbb{T}_B \xrightarrow{\mathcal{T}} \mathsf{T}_B$:

$$\mathsf{T}_B^{\text{opt}} = \bigoplus \mathsf{C}! \begin{cases} add(\text{i32}).\mathsf{A}?add(\text{i32}).\mathsf{T}_B^{\text{opt}} \\ sub(\text{i32}).\mathsf{A}?add(\text{i32}).\mathsf{T}_B^{\text{opt}} \end{cases} \qquad \mathsf{T}_B = \mathsf{A}?add(\text{i32}).\bigoplus \mathsf{C}! \begin{cases} add(\text{i32}).\mathsf{T}_B \\ sub(\text{i32}).\mathsf{T}_B \end{cases}$$



(a) tree representation of $\mathbb{T}_B^{\text{opt}}$ (sorts skipped)



(b) tree representation of $\mathbb{T}_B$ (sorts skipped)

Recall that this translation is carried out by the corecursive function `lt2st` (Definition 3.1), which implements the relation $\xrightarrow{\mathcal{T}}$ (`lt2st TyBOp` and `lt2st TyB` in the Coq development). We then decompose $\mathsf{T}_B^{\text{opt}}$ and T_B into streams of actions—represented as SISO trees—and present them in the table below:

Decompositions

$P_{AL} = \mathsf{C}!add(\text{i32}).\mathsf{A}?\text{add(i32)}$ $P_{SL} = \mathsf{C}!sub(\text{i32}).\mathsf{A}?\text{add(i32)}$	$P_{AR} = \mathsf{A}?add(\text{i32}).\mathsf{C}!\text{add(i32)}$ $P_{SR} = \mathsf{A}?add(\text{i32}).\mathsf{C}!\text{sub(i32)}$
$\mathsf{W}_2 = P_{AL}.\mathsf{W}_2 \lhd_{\text{SO}} \mathsf{T}_B^{\text{opt}}$ $\mathsf{W}_4 = P_{SL}.\mathsf{W}_4 \lhd_{\text{SO}} \mathsf{T}_B^{\text{opt}}$ $\mathsf{W}_6 = (P_{AL} + P_{SL})^n.\mathsf{W}_2 \lhd_{\text{SO}} \mathsf{T}_B^{\text{opt}} \quad \forall n \geqslant 1$ $\mathsf{W}_8 = (P_{AL} + P_{SL})^n.\mathsf{W}_4 \lhd_{\text{SO}} \mathsf{T}_B^{\text{opt}} \quad \forall n \geqslant 1$ $\mathsf{W}_2 \lhd_{\text{SI}} \mathsf{W}_2$ $\mathsf{W}_4 \lhd_{\text{SI}} \mathsf{W}_4$ $\mathsf{W}_6 \lhd_{\text{SI}} \mathsf{W}_6$ $\mathsf{W}_8 \lhd_{\text{SI}} \mathsf{W}_8$	$\mathsf{T}_B \lhd_{\text{SI}} \mathsf{T}_B$ $\mathsf{W}_1 = P_{AR}.\mathsf{W}_1 \lhd_{\text{SO}} \mathsf{T}_B$ $\mathsf{W}_3 = P_{SR}.\mathsf{W}_3 \lhd_{\text{SO}} \mathsf{T}_B$ $\mathsf{W}_5 = (P_{AR} + P_{SR})^n.\mathsf{W}_1 \lhd_{\text{SO}} \mathsf{T}_B \quad \forall n \geqslant 1$ $\mathsf{W}_7 = (P_{AR} + P_{SR})^n.\mathsf{W}_3 \lhd_{\text{SO}} \mathsf{T}_B \quad \forall n \geqslant 1$

In the notation $(P + Q)^n$, the operator + denotes a non-deterministic choice between the action prefixes P and Q, and the superscript indicates that the actions are concatenated n times.

According to Definition 4.1, for every member W in the set $\{\mathsf{W}_2, \mathsf{W}_4, \mathsf{W}_6, \mathsf{W}_8\}$, if we can find an element W′ from the set $\{\mathsf{W}_1, \mathsf{W}_3, \mathsf{W}_5, \mathsf{W}_7\}$ such that $\mathsf{W} \lesssim \mathsf{W}'$, we can then conclude that $\mathsf{T}_B^{\text{opt}} \leqslant \mathsf{T}_B$. We now show that

$$\text{(i) } W_2 \lesssim W_1 \quad \text{(ii) } W_4 \lesssim W_3 \quad \text{(iii) } W_6 \lesssim W_5 \quad \text{(iv) } W_8 \lesssim W_7$$

hold, in a Coq implementation.

The refinement in (i) captures possible reorderings when the first choices in the selections of T_B^{opt} and T_B are taken infinitely often. Refinement (ii) covers reorderings arising when the second choices are selected infinitely often. Refinement (iii) handles cases where the selections alternate non-deterministically between first and second choices for a finite number of steps, followed by infinitely many selections of the first choices. Refinement (iv) mirrors (iii), but instead considers infinitely many selections of the second choices after finitely many alternations. Cases (iii) and (iv) were unintentionally left unproven in the conference version [17] of this work.

Terms $W_1 \cdots W_4$ can be implemented straightforwardly:

```
CoFixpoint w1 ≜ "A" & [|("add", I, "C" ! [|("add", I, w1)|])|].
CoFixpoint w2 ≜ "C" ! [|("add", I, "A" & [|("add", I, w2)|])|].
CoFixpoint w3 ≜ "A" & [|("add", I, "C" ! [|("sub", I, w3)|])|].
CoFixpoint w4 ≜ "C" ! [|("sub", I, "A" & [|("add", I, w4)|])|].
```

To obtain the terms $W_5 \cdots W_8$ in Coq, we represent the non-deterministic choice among the prefixes using Boolean lists, since a finite number of binary choices is sufficient.

```
Fixpoint rShape (l: list bool): Dpf ≜
  match l with
    | nil   ⇒ dpf_end
    | x::xs ⇒ if x then dpf_receive "A" "add" (I) (dpf_send "C" "add" (I) (rShape xs))
              else dpf_receive "A" "add" (I) (dpf_send "C" "sub" (I) (rShape xs))
  end.
```

where `Dpf` is a finite prefix type, such as `Ap`, `Apf` and `Bpf`, (with constructors `dpf_receive`, `dpf_send` and `dpf_end`) however it does not impose any restriction on the actions involved.

We also define a function `sShape` as a variant of `rShape`, where in the `cons` case it returns:

$$\begin{cases} \texttt{dpf\_send "C" "add" (I) (dpf\_receive "A" "add" (I) (sShape xs))} & \texttt{x = true} \\ \texttt{dpf\_send "C" "sub" (I) (dpf\_receive "A" "add" (I) (sShape xs))} & \text{otherwise.} \end{cases}$$

We now formalise the terms $W_5 \cdots W_8$:

```
Definition w5 l k ≜ (merge_dpf_contn (rShape l) w1 k).
Definition w6 l k ≜ (merge_dpf_contn (sShape l) w2 k).
Definition w7 l k ≜ (merge_dpf_contn (rShape l) w3 k).
Definition w8 l k ≜ (merge_dpf_contn (sShape l) w4 k).
```

The function `merge_dpf_contn` takes a prefix `d` of type `Dpf`, a term `w`, and a natural number `k`; it concatenates `d` with itself `k` times and then prefixes the resulting sequence to `w`.

Lemma 6.6. *For all terms* `w w'`*, boolean lists* `l`*, given that* `refinement w w'`*, prefixing* `sShape l` *to* `w` *and* `rShape l` *to* `w'` *preserves the refinement:*

```
refinement w w' → refinement (merge_dpf_cont (sShape l) w) (merge_dpf_cont (rShape l) w').
```

Proof. Proceeds by case split over the structure of the list `l`.

(1) `nil`. Trivially closed by the hypothesis as the goal reduces into `refinement w w'`.

(2) `(a :: l)`. When `a = true`, the goal takes the form:

```
refinement (merge_dpf_cont (dpf_send "C" "add" (I) (dpf_receive "A" "add" (I) (sShape l))) w)
           (merge_dpf_cont (dpf_receive "A" "add" (I) (dpf_send "C" "add" (I) (rShape l))) w').
```

We discharge this by applying the constructors `ref_b` followed by `ref_a`, and then invoking the hypothesis. Action equality checks are omitted, as they are straightforward. The case `a = false` is handled analogously. □

We provide the list `[(w2,w1);(w4,w3);(w6,w5);(w8,w7)]` as the existential argument to the Coq implementation `subtype` of Definition 6.3. We first demonstrate that the terms `w2`, `w4`, `w6`, and `w8` are streamlines of actions (SISO trees) obtained by decomposing the tree `lt2st TyBOp`, while `w1`, `w3`, `w5`, and `w7` result from decomposing `lt2st TyB`. This decomposition is handled by the relation `st2siso`, which we omit here as it consists solely of constructor applications. We then prove:

(i) `refinement w2 w1` (ii) `refinement w4 w3`
(iii) `refinement (w6 l k) (w5 l k)` (iv) `refinement (w8 l k) (w7 l k)`,
for all boolean lists `l`, and natural numbers `k`.

Proofs of (i) and (ii). (🐓, 🐓) Save the initial state as the coinduction hypothesis, apply the constructors `ref_b` followed by `ref_a`, and then invoke the coinduction hypothesis to close the goal. Action equality checks are omitted here, as they are routine and lengthy. □

Proof of (iii). 🐓 Proceeds by induction over the structure of the natural number `k`.

(1) `0`. The goal reduces to `refinement w2 w1`, which is trivially discharged using the proof of (i).
(2) `S k`. The goal is now of the following shape:

```
refinement (merge_dpf_cont (sShape l) (merge_dpf_cont (DpnD3 (sShape l) k) w2))
           (merge_dpf_cont (rShape l) (merge_dpf_cont (DpnD3 (rShape l) k) w1))
```

> The function `DpnD3` takes a prefix `d` of type `Dpf` and a natural number `k`, and returns the result of concatenating `d` with itself `k` times.

It follows by applying Lemma 6.6 together with the induction hypothesis. □

Proof of (iv). 🐓 It follows a similar structure to that of (iii): it uses the proof of (ii) in the base case and replicates the exact same steps in the inductive case. □

All together, this verifies that $\mathbb{T}_B^{\text{opt}} \leqslant \mathbb{T}_B$; in Coq, `subltype (lt2st TyBOp) (lt2st TyB)` 🐓.

6.4 Example by Bravetti et al. [5]

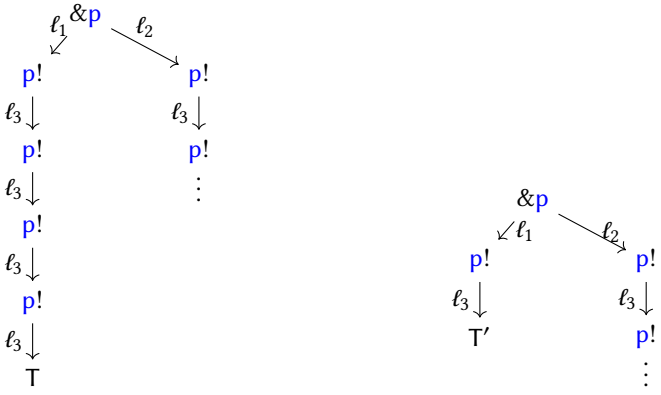
Another instance of optimisation enabled by asynchronous subtyping appears in the protocol for distributed batch processing (Example 4.14 in [22]). We have formalised the corresponding subtyping proof in Coq 🐓. Rather than presenting this proof, we focus on a different optimisation scenario (Example 3.19 in [22]), where subtyping proof is more intricate due to its coinductive yet non-cyclic derivations. Browse the entire proof in the file `Example3_19.v`.

Consider the following session types:

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

```
Definition lTS: local ≜ lt_mu (lt_send "p" [("l3",sint,(lt_var 0))]).
Definition T: local ≜ lt_mu (lt_receive "p" [("l1",sint,lt_send "p" [("l3",sint,
  lt_send "p" [("l3",sint,lt_send "p" [("l3",sint,(lt_var 0))])])]); ("l2",sint,lTS)]).
Definition T': local ≜ lt_mu (lt_receive "p" [("l1",sint,lt_send "p" [("l3",sint,(lt_var 0))]);
                                                ("l2",sint,lTS)]).
```

We aim to demonstrate that $\mathbb{T} \leqslant \mathbb{T}'$. To achieve this, we start with translating the types $\mathbb{T}$ and $\mathbb{T}'$ into their corresponding session trees T and T′:



The translation is carried out by the cofixpoint `lt2st` in the Coq development (namely, `lt2st T` and `lt2st T'`). We then decompose T and T′ into streams of actions, represented as SISO trees, and present them in the table below:

Decompositions

$\pi_3 = \mathrm{p}?\ell_1(\mathrm{S}).\mathrm{p}!\ell_3(\mathrm{S}).\mathrm{p}!\ell_3(\mathrm{S}).\mathrm{p}!\ell_3(\mathrm{S})$	$\pi_1 = \mathrm{p}?\ell_1(\mathrm{S}).\mathrm{p}!\ell_3(\mathrm{S})$
$\mathrm{T} \vartriangleleft_{\mathrm{SO}} \mathrm{T}$ $\mathrm{WB} = \pi_3.\mathrm{WB} \vartriangleleft_{\mathrm{SI}} \mathrm{T}$ $\mathrm{WA} = \mathrm{p}?\ell_2(\mathrm{S}).\mathrm{WD} \vartriangleleft_{\mathrm{SI}} \mathrm{T}$ where $\mathrm{WD} = \mathrm{p}!\ell_3(\mathrm{S}).\mathrm{WD}$ $\pi_3^n.\mathrm{WA} \vartriangleleft_{\mathrm{SI}} \mathrm{T} \quad \forall n \geqslant 1$	$\mathrm{WB}' = \pi_1.\mathrm{WB}' \vartriangleleft_{\mathrm{SI}} \mathrm{T}'$ $\mathrm{WA} \vartriangleleft_{\mathrm{SI}} \mathrm{T}'$ $\pi_1^n.\mathrm{WA} \vartriangleleft_{\mathrm{SI}} \mathrm{T}' \quad \forall n \geqslant 1$ $\mathrm{WB}' \vartriangleleft_{\mathrm{SO}} \mathrm{WB}'$ $\mathrm{WA} \vartriangleleft_{\mathrm{SO}} \mathrm{WA}$ $\pi_1^n.\mathrm{WA} \vartriangleleft_{\mathrm{SO}} \pi_1^n.\mathrm{WA}$

Before proceeding with the proof steps, we introduce these tree terms and prefixes in Coq for later use within the proof.

```
CoFixpoint WB:  st ≜ "p" & [|("l1",I,"p" ! [|("l3",I,"p" ! [|("l3",I,"p" ! [|("l3",I,WB)|])|])|])|].
CoFixpoint WD:  st ≜ "p" ! [|("l3",I,WD)|].
Definition WA:  st ≜ "p" & [|("l2",I,WD)|].
CoFixpoint WB': st ≜ "p" & [|("l1",I,"p" ! [|("l3",I,WB')|])|].
Definition pi1: Dpf ≜ dpf_receive "p" "l1" (I) (dpf_send "p" "l3" (I) dpf_end).
Definition pi3: Dpf ≜ dpf_receive "p" "l1" (I) (dpf_send "p" "l3" (I) (dpf_send "p" "l3" (I)
                  (dpf_send "p" "l3" (I) dpf_end))).
```

Employing Definition 4.1, we are supposed to demonstrate for every member W′ of the set $\{\mathrm{WB}', \mathrm{WA}, \pi_1^n.\mathrm{WA}\}$ that there exists a member W in $\{\mathrm{WB}, \mathrm{WA}, \pi_3^n.\mathrm{WA}\}$ such that $\mathrm{W} \lesssim \mathrm{W}'$.

We achieve this by proving

(i) $\mathtt{WB} \lesssim \mathtt{WB'}$ (ii) $\mathtt{WA} \lesssim \mathtt{WA}$ (iii) $\pi_3^n.\mathtt{WA} \lesssim \pi_1^n.\mathtt{WA}$ $\forall$ natural numbers n.

To formalise this in Coq, we adopt the approach outlined in Definition 6.3 and provide the list `[(WB,WB');(WA,WA);((merge_dpf_contn pi3 WA n),(merge_dpf_contn pi1 WA n))]` as the existential argument, and proceed as follows.

(1) Demonstrate that

`(WB,(lt2st T))` `(WB',(lt2st T'))` `(WA,(lt2st T))` `(WA,(lt2st T'))`
`((merge_dpf_contn pi3 WA n),(lt2st T))` `((merge_dpf_contn pi1 WA n),(lt2st T'))`

are in `st2sisoC`;

(2) Prove above refinements (i), (ii) and (iii).

The complexity in the refinements described in item (2) stems from the intricate corecursive nature of the terms `(lt2st T)` and `(lt2st T')`. Intuitively, case (i) captures the action reordering across infinitely many unfoldings of the outer corecursive structures, while case (ii) targets the infinitely many unfoldings of the inner structures. Case (iii), in turn, addresses a combination: finitely many unfoldings of the outer structures and infinitely many of the inner ones.

Proof of (1). We begin by proving that pairs `(WB,(lt2st T))` and `(WB',(lt2st T'))` 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. We omit the last four cases of item (1) since they follow the same steps. □

Proof of (2)(i). Showing that `WB` $\lesssim$ `WB'` holds however presents a more intriguing scenario. For that, a pen-and-paper proof is structured in Figure 7 (read: bottom left $\longrightarrow$ top left $\xrightarrow{\text{generalised by}}$ bottom right $\longrightarrow$ top right). Steps on the left are straightforward refinement rule applications, where

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{WB} \lesssim \mathtt{p}?\ell_1(\mathtt{S}).\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{WB'}}{\mathtt{p}!\ell_3(\mathtt{S}).\mathtt{WB} \lesssim \mathtt{p}?\ell_1(\mathtt{S}).\mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{(\mathtt{p}!\ell_3(\mathtt{S}))^2.\mathtt{WB} \lesssim \mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{(\mathtt{p}!\ell_3(\mathtt{S}))^3.\mathtt{WB} \lesssim \mathtt{p}!\ell_3(\mathtt{S}).\mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{\mathtt{WB} \lesssim \mathtt{WB'}}\,[\text{REF-}\mathcal{A}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{WB} \lesssim (\mathtt{p}?\ell_1(\mathtt{S}_1).\mathtt{p}?\ell_1(\mathtt{S}))^{n+1}.\mathtt{WB'}}{\mathtt{p}!\ell_3(\mathtt{S}).\mathtt{WB} \lesssim (\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{p}?\ell_1(\mathtt{S}_1))^{n}.\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{(\mathtt{p}!\ell_3(\mathtt{S}))^2.\mathtt{WB} \lesssim (\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{p}?\ell_1(\mathtt{S}))^{n}.\mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{(\mathtt{p}!\ell_3(\mathtt{S}))^3.\mathtt{WB} \lesssim (\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{p}?\ell_1(\mathtt{S}))^{n}.\mathtt{p}!\ell_3(\mathtt{S}).\mathtt{WB'}}\,[\text{REF-}\mathcal{B}]}{\mathtt{WB} \lesssim (\mathtt{p}?\ell_1(\mathtt{S}).\mathtt{p}?\ell_1(\mathtt{S}))^{n}.\mathtt{WB'}}\,[\text{REF-}\mathcal{A}]$$

Fig. 7. Proof steps of $\mathtt{WB} \lesssim \mathtt{WB'}$. (*Source:* [22, Example 3.19])

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 $\mathtt{p}?\ell_1(\mathtt{S})$ and ensure that this happens an even number of times. Below lemma aligns with the bottommost line of the helper steps in Figure 7 .

```
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 $\mathtt{r}?\ell(\mathtt{S}).\mathcal{B}^{(\mathtt{p})}$ constructor of $\mathcal{B}^{(\mathtt{p})}$, 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 ≲ 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 10. *To center the attention on actions and continuations, we will no longer use colist notation, labels, or sorts in the rest of the proof text. This is because both sides of ≲ are made of streamline of actions (nested singleton colists), where all elements (labels and sorts) align. We use a dot to separate prefixes from the infinite terms. The notation* `p?` *(resp.* `p!`*) indicates a single receive (resp. send) action from (resp. to) participant* `p`.

Unfolding `WB` and applying the rule `ref_a` transforms the goal into `p!p!p!.WB ≲ (p?)^{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 7. 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 ≲ (p?)^{n+2}p!.WB'`). We then apply `ref_b` with `n ≜ n+2`, resulting in `p!p!.WB ≲ (p?)^{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 ≲ (p?)^{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 6.1. □

Proof of (2)(ii). ❦ We begin by applying the rule [ref-$\mathcal{A}$], which reduces the goal to WD ≲ WD. This is straightforward: we record the initial state as the coinduction hypothesis *CIH*, apply the [ref-$\mathcal{B}$] constructor, and discharge the goal by invoking *CIH*. □

Lemma 6.7. ❦ $\forall n, \quad \pi_3^{n+1}.\mathrm{WA} \lesssim \pi_1.\pi_3^n.\mathrm{WA}$.

Proof. We save the initial state as the coinduction hypothesis $CIH : \forall n, \quad \pi_3^{n+1}.\mathrm{WA} \lesssim \pi_1.\pi_3^n.\mathrm{WA}$, and proceed by induction on the structure of n.

The proof steps rely on the term definitions $\mathrm{WA} = \mathrm{p}?\ell_2(\mathrm{S}).\mathrm{WD} \triangleleft_{\mathrm{SI}} \mathrm{T}$, where $\mathrm{WD} = \mathrm{p}!\ell_3(\mathrm{S}).\mathrm{WD}$. We unfold these definitions as needed throughout the derivation. For brevity, we omit the action equality checks that are generated after each constructor application.

(1) In the base case, when n is zero, we build the following derivation tree, highlighting the matching terms:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{WD} \lesssim \mathrm{WD}}}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} \lesssim \mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} = \mathrm{WD}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p?}\ell_2(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} \lesssim \mathrm{p?}\ell_2(\mathrm{S}).\mathrm{WD}}\,[\text{REF-}\mathcal{A}]}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p?}\ell_2(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} \lesssim \mathrm{p?}\ell_2(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} = \mathrm{p?}\ell_2(\mathrm{S}).\mathrm{WD}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WA} = \mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p?}\ell_2(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} \lesssim \mathrm{p?}\ell_2(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WD} = \mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WA} \lesssim \mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WA} \lesssim \mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{WA}}\,[\text{REF-}\mathcal{A}]$$

The proof of $\mathrm{WD} \lesssim \mathrm{WD}$ is trivial and follows from the argument given in case (2)(ii) above.

(2) In the step case, where n is non-zero, we apply the following constructors:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\pi_3^{n+1}.\mathrm{WA} \lesssim \mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n}.\mathrm{WA} = \pi_1.\pi_3^{n}.\mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA} \lesssim \mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n}.\mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA} \lesssim \mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n}.\mathrm{WA} = \pi_3^{n+1}.\mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA} \lesssim \mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA}}\,[\text{REF-}\mathcal{B}]}{\mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA} \lesssim \mathrm{p?}\ell_1(\mathrm{S}).\mathrm{p!}\ell_3(\mathrm{S}).\pi_3^{n+1}.\mathrm{WA}}\,[\text{REF-}\mathcal{A}]$$

At the end of the derivation, the goal is discharged by the coinduction hypothesis *CIH*.

□

Lemma 6.8. *$\forall n\ k, \quad n - k \geqslant 1$ implies that $\pi_1^k.\pi_3^{n-k}.\mathrm{WA} \lesssim \pi_1^{k+1}.\pi_3^{n-k-1}.\mathrm{WA}$.*

Proof. Proceeds with induction over the structure of k.

(1) In the base case, when k is zero, it follows from Lemma 6.7 with $n := n - 1$.
(2) In the step case, we apply the [REF-$\mathcal{A}$] followed by the [REF-$\mathcal{B}$] over the goal and then employ the induction hypothesis with $n := n - 1$.

□

Lemma 6.9. *$\forall k\ n\ u, \quad n \geqslant u + k$ implies that $\pi_1^u.\pi_3^{n-u}.\mathrm{WA} \lesssim \pi_1^{u+k}.\pi_3^{n-u-k}.\mathrm{WA}$.*

Proof. Proceeds with induction over the structure of k.

(1) In the base case, when k is zero, the goal takes the form $\pi_1^u.\pi_3^{n-u}.\mathrm{WA} \lesssim \pi_1^u.\pi_3^{n-u}.\mathrm{WA}$. Reflexivity follows directly from standard rule applications. Similar to the proof of case 2(ii) above.
(2) In the step case, we obtain $H_1 : \pi_1^{u+k}.\pi_3^{n-u-k}.\mathrm{WA} \lesssim \pi_1^{u+k+1}.\pi_3^{n-k-2}.\mathrm{WA}$ from Lemma 6.8. In addition, induction hypothesis reveals $H_2 : \pi_1^u.\pi_3^{n-u}.\mathrm{WA} \lesssim \pi_1^{u+k}.\pi_3^{n-u-k}.\mathrm{WA}$. By transitivity, Theorem 6.5, we conclude $\pi_1^u.\pi_3^{n-u}.\mathrm{WA} \lesssim \pi_1^{u+k+1}.\pi_3^{n-k-2}.\mathrm{WA}$ which matches, and discharges the goal in this case.

□

Proof of (2)(iii). Thanks to Lemma 6.9, with $k := n$, $n := n$ and $u := 0$, we get an hypothesis

```
H: m ≥ 0 + m → refinement (merge_dpf_contn pi1 (merge_dpf_contn pi3 WA (n - 0)) 0)
                          (merge_dpf_contn pi1 (merge_dpf_contn pi3 WA (n - 0 - n)) (0 + n))
```

Performing simple natural number manipulations within `H` results in

```
H: refinement (merge_dpf_contn pi3 WA n) (merge_dpf_contn pi1 WA n)
```

which closes the goal.

□

All together, this verifies that $\mathbb{T} \leqslant \mathbb{T}'$; in Coq, `subltype (lt2st T) (lt2st T')` 🐓.

7 RELATED WORK AND CONCLUSION

Asynchronous session subtyping was first introduced to achieve message optimisation in session-based high-performance computing platforms, i.e., multicore C programming [29, 52] and MPI-C [39, 40]. Then, numerous theoretical and practical advancements have emerged.

In theory, Chen et al. [11] introduced and proved *preciseness* of synchronous [15, 18] 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 [38]. This provoked active studies on (1) identifying a set of binary session types where asynchronous subtyping is decidable [7, 38]; and (2) proposing *sound* algorithms extending the formalism to *binary* communicating automata [4] in [2, 5] (also to fair refinement in [8]). In the multiparty setting, Ghilezan et al. [20–22] 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 8) 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 8 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]	✗	✗	✗	✓	✓

Fig. 8. Examples and related work

Mechanisation recently emerges as a pivotal facet in the concurrent communication models.

Tirore et al. [50] introduced a novel computable projection function that maps global types to local types. This function is formally verified in Coq for both soundness and completeness with respect to its coinductive tree semantics.

Castro-Perez et al. [9] introduced Zooid, a domain-specific language embedded in Coq for certified multiparty communication. Zooid provides mechanised soundness and completeness guarantees by establishing trace equivalences between the labelled transition systems of local and global types, thereby preserving key properties such as deadlock-freedom and protocol compliance.

Tassarotti et al. [46] developed a compiler for a functional language with binary session types, based on a simplified version of the GV system [19], and formally verified its correctness in Coq.

Jacobs et al. [32] extended this work with MPGV, which enriches a linear lambda calculus with multiparty session types, supporting features like participant redirection and dynamic thread spawning. Their type system includes both global and local types, where local types manage linear data. Deadlock freedom is guaranteed by encoding cyclic communication patterns as acyclic graphs, removing the need for central coordination. The proof of preservation and progress [32, Theorem 5.7] employs separation logic and configuration invariants to ensure that well-typed configurations do not get stuck.

Hinrichsen et al. [23, 24, 26] developed Actris, a verification tool that integrates separation logic with asynchronous session types (including subtyping), built on the Coq Iris program logic [34–37]. Jacobs et al. [33] extended this into LinearActris, incorporating linear logic to ensure both deadlock-freedom and memory leak-freedom. However, both Actris and LinearActris are limited to binary session types.

More recently, Hinrichsen et al. [25] proposed the Multris framework, which combines separation logic for verifying functional correctness with multiparty message-passing and shared-memory concurrency. Their approach formally proves protocol consistency within the Coq Iris environment, and is inspired by the bottom-up MPST methodology of [42], which centres on local types. As a result, properties that rely on global type structures are not captured or proven within Multris.

Tirore [49] formalises subject reduction in Coq for the multiparty session π-calculus of [30], including session initialisation and delegation. The type system employs channel-explicit global and local types, with projections based on [50]. Channel-explicit types require additional linearity checks to ensure projectability of global types, which complicates the formalisation and makes it less adaptable to other systems. In contrast, most session type systems—including ours—use channel-implicit types, offering greater flexibility and ease of integration.

Brady [3] devised secure communication protocols for binary sessions in Idris. Thiemann et al. [48] formalised the progress and preservation properties for binary session types in Agda.

The choreographic programming paradigm allows distributed programs to be implemented as single programs, ensuring coherence between send and receive operations by consolidating them into a unified construct. Deadlock freedom is inherent in the design.

Cruz-Filipe et al. [12] formalised the theory of choreographic programming in Coq. In their work, Hirsch and Garg introduced Pirouette [27], a choreographic language designed with formal guarantees, which are rigorously verified in Coq. Similarly, Pohjola et al. [41] presented Kalas, a compiler for a choreographic language whose correctness has been verified using the HOL4 theorem prover.

Conclusion. In this paper, we present the first formalisation framework for asynchronous subtyping in MPST. Our approach: (1) translates session types into session trees (recursion unfolding); (2) decomposes arbitrary session trees into SISO trees, which are free of branching and selection; and (3) defines the subtyping relation via a notion of refinement between these trees. We demonstrate the utility of our framework by certifying four representative protocol optimisation examples from the literature (see Figure 8).

In our development, we redefined the negation of the refinement relation, significantly reducing the number of rules from eighteen [22] to eight, and proved that subtyping is complete with respect to this revised negation. To precisely characterise refinement and its negation, we introduced a new form of term prefixing. We also proved the correctness and transitivity of the refinement relation.

Future Directions. Our future work includes developing a complete type system for asynchronous MPST, together with languages for processes and sessions, typing rules incorporating subtyping, and environment reductions, with the ultimate goal of turning the present library into a full mechanisation of asynchronous communication.

As part of this effort, we aim to establish a Coq proof of liveness [22, Lemma 4.10], a behavioural property of typing environments ensuring that every pending send eventually enqueues a message and every pending reception is eventually executed, even in the presence of environment reductions modulo subtyping. Building on this result, we plan to combine liveness with inversion and substitution lemmas [22, Lemma B.14, Lemma B.17] to prove subject reduction [22, Theorem 4.11]. The main technical challenge in this programme lies in decomposing session trees into SI, SO, and SISO trees (Figure 2). While the current development employs a relational encoding of these decompositions, exploring alternative representations may further streamline the mechanisation and proof structure.

Axiomatic Base and Mechanisation Effort. In the accompanying library, we rely on classical reasoning to perform case analysis over coinductively defined predicates, for instance over `act_eq`, in order to distinguish whether two sets of actions are equal or distinct. Classical reasoning is also required when relating the coinductively defined refinement relation to its inductively defined negation. Rather than attempting to invert the negation of a coinductive predicate—which, when encoded in Coq's **`Prop`**, yields little useful information in the proof context—we proceed indirectly. We first establish that negating the inductive negation of refinement implies the coinductive refinement relation itself. We then apply *contraposition* (making use of Law of Excluded Middle) to derive that negating refinement implies the inductive negation of refinement (Lemma 5.1).

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 32K lines of code (excluding the other components of the library that formalise processes, environments and type-checking), containing 642 proven lemmata and 186 definitions, with 43 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.

ACKNOWLEDGMENTS

We are grateful to Dawit Tirore, Marco Giunti, and Mukesh Tiwari for their feedback on earlier versions of this paper. We also thank Jovanka Vanja Pantovic and Alceste Scalas for their in-depth discussions on the negation of the refinement relation. Finally, we appreciate the constructive comments provided by the anonymous referees of ITP 2024 and ACM-TOCL.

This work is partially supported by EPSRC EP/T006544/2 ("POST"), EP/T014709/2 ("STARDUST"), EP/Y005244/1 ("VSL-Q"), EP/V000462/1 ("AppContol"), EP/X015955/1 ("Morello-HAT"), EU Horizon 101093006 and UKRI 10066667 ("TaRDIS"), Advanced Research and Invention Agency (ARIA) Safeguarded AI, EP/Z533749/1 ("IDEAL"), and a grant from the Simons Foundation.

REFERENCES

[1] Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2023. Inductive Reasoning for Coinductive Types. *CoRR* abs/2301.09802 (2023). https://doi.org/10.48550/ARXIV.2301.09802 arXiv:2301.09802

[2] Laura Bocchi, Andy King, and Maurizio Murgia. 2024. Asynchronous Subtyping by Trace Relaxation. In *TACAS'24: 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Luxembourg City, Luxembourg, 6 - 11 April 2024*. Springer.

[3] Edwin C. Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Comput. Sci.* 18, 3 (2017). https://doi.org/10.7494/CSCI.2017.18.3.1413

[4] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342. https://doi.org/10.1145/322374.322380

[5] Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2021. A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science* Volume 17, Issue 1 (March 2021). https://doi.org/10.23638/LMCS-17(1:20)2021

[6] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2017. Undecidability of asynchronous session subtyping. *Inf. Comput.* 256 (2017), 300–320.

[7] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2018. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.* 722 (2018), 19–51. https://doi.org/10.1016/j.tcs.2018.02.010

[8] Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. 2021. Fair Refinement for Asynchronous Session Types. In *FoSSaCS (Lecture Notes in Computer Science)*.

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

[10] David Castro-Perez and Nobuko Yoshida. 2020. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 155:1–155:30. https://doi.org/10.1145/3428223

[11] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *LMCS* 13 (2017), 1–62. Issue 2.

[12] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2023. A Formal Theory of Choreographic Programming. *J. Autom. Reason.* 67, 2 (2023), 21. https://doi.org/10.1007/S10817-023-09665-3

[13] Zak Cutner and Nobuko Yoshida. 2021. Safe Session-Based Asynchronous Coordination in Rust. In *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 (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 80–89. https://doi.org/10.1007/978-3-030-78142-2_5

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

[15] Romain Demangeon and Kohei Honda. 2011. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *22nd International Conference on Concurrency Theory (LNCS, Vol. 6901)*. Springer, 280–296.

[16] Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7454)*, Maciej Koutny and Irek Ulidowski (Eds.). Springer, 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

[17] Burak Ekici and Nobuko Yoshida. 2024. Completeness of Asynchronous Session Tree Subtyping in Coq. In *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia (LIPIcs, Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:20. https://doi.org/10.4230/LIPICS.ITP.2024.13

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

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

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

[21] Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. 2021. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.* 5, Article 16 (jan 2021), 28 pages.

[22] Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. 2023. Precise Subtyping for Asynchronous Multiparty Sessions. *ACM Trans. Comput. Logic* 24, 2, Article 14 (Nov 2023), 73 pages. https://doi.org/10.1145/3568422

[23] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

[24] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/LMCS-18(2:16)2022

[25] Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Multris: Functional Verification of Multiparty Message Passing in Separation Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1446–1474. https://doi.org/10.1145/3689762

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

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

[28] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP 1998*. 122–138. https://doi.org/10.1007/BFb0053567

[29] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2009. Type-Directed Compilation for Multicore Programming. *ENTCS* 241 (2009), 101–111.

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

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

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

[33] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proc. ACM Program. Lang.* 8, POPL (2024), 1385–1417. https://doi.org/10.1145/3632889

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

[35] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[36] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[37] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[38] Julien Lange and Nobuko Yoshida. 2017. On the Undecidability of Asynchronous Session Subtyping. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 10203)*. 441–457.

[39] Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. 2015. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction (LNCS, Vol. 9031)*. Springer, 212–232.

[40] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *50th International Conference on Objects, Models, Components, Patterns (LNCS, Vol. 7304)*. Springer, 202–218.

[41] Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:18. https://doi.org/10.4230/LIPICS.ITP.2022.27

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

[43] Kathrin Stark. 2020. *Mechanising syntax with binders in Coq*. Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28822

[44] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 166–180. https://doi.org/10.1145/3293880.3294101

[45] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE 1994*. 398–413. https://doi.org/10.1007/3-540-58184-7_118

[46] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936.

https://doi.org/10.1007/978-3-662-54434-1_34

[47] The Coq Development Team. 2023. The Coq Reference Manual – Release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman.

[48] Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 19:1–19:15. https://doi.org/10.1145/3354166.3354184

[49] Dawit Legesse Tirore. 2024. *A Mechanisation of Multiparty Session Types.* Ph. D. Dissertation. ITU Copenhagen.

[50] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. 2023. A Sound and Complete Projection for Global Types. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:19. https://doi.org/10.4230/LIPICS.ITP.2023.28

[51] Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 11969)*, Dang Van Hung and Meenakshi D'Souza (Eds.). Springer, 73–93. https://doi.org/10.1007/978-3-030-36987-3_5

[52] Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. 2008. Session-Based Compilation Framework for Multicore Programming. In *FMCO 2008*. 226–246. https://doi.org/10.1007/978-3-642-04167-9_12

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