Precise Subtyping for Asynchronous Multiparty Sessions

SILVIA GHILEZAN, JOVANKA PANTOVIĆ, and IVAN PROKIĆ, Univerzitet u Novom Sadu, Serbia
ALCESTE SCALAS, Technical University of Denmark, DK and Aston University, UK
NOBUKO YOSHIDA, Imperial College London, UK

Session subtyping is a cornerstone of refinement of communicating processes: a process implementing a session type (i.e., a communication protocol) T can be safely used whenever a process implementing one of its supertypes T' is expected, in any context, without introducing deadlocks nor other communication errors. As a consequence, whenever $T \leqslant T'$ holds, it is safe to replace an implementation of T' with an implementation of the subtype T, which may allow for more optimised communication patterns.

We present the first formalisation of the *precise* subtyping relation for *asynchronous multiparty* sessions. We show that our subtyping relation is *sound* (i.e., guarantees safe process replacement, as outlined above) and also *complete*: any extension of the relation is unsound. To achieve our results, we develop a novel *session decomposition* technique, from *full* session types (including internal/external choices) into *single input/output session trees* (without choices).

Previous work studies precise subtyping for *binary* sessions (with just two participants), or multiparty sessions (with any number of participants) and *synchronous* interaction. Here, we cover *multiparty* sessions with *asynchronous* interaction, where messages are transmitted via FIFO queues (as in the TCP/IP protocol), and prove that our subtyping is both operationally and denotationally precise. In the asynchronous multiparty setting, finding the precise subtyping relation is a highly complex task: this is because, under some conditions, participants can permute the order of their inputs and outputs, by sending some messages earlier or receiving some later, without causing errors; the precise subtyping relation must capture *all* such valid permutations — and consequently, its formalisation, reasoning and proofs become challenging. Our session decomposition technique overcomes this complexity, expressing the subtyping relation as a composition of refinement relations between single input/output trees, and providing a simple reasoning principle for asynchronous message optimisations.

CCS Concepts: • **Theory of computation** → **Process calculi**; **Type structures**.

Additional Key Words and Phrases: session types, π-calculus, typing systems, subtyping, asynchronous multiparty session types, soundness, completeness

ACM Reference Format:
Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. 2021. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.* 5, POPL, Article 16 (January 2021), 28 pages. https://doi.org/10.1145/3434297

Authors' addresses: Silvia Ghilezan, gsilvia@uns.ac.rs; Jovanka Pantović, pantovic@uns.ac.rs; Ivan Prokić, prokic@uns.ac.rs, Univerzitet u Novom Sadu, Novi Sad, Serbia; Alceste Scalas, alcsc@dtu.dk, Technical University of Denmark, Richard Petersens Plads, Bygning 324, Kongens Lyngby, DK, 2800, Aston University, Birmingham, UK; Nobuko Yoshida, n.yoshida@imperial.ac.uk, Imperial College London, South Kensington Campus, London, UK, SW7 2AZ.

2475-1421/2021/1-ART16
https://doi.org/10.1145/3434297

1 INTRODUCTION

Modern software systems are routinely designed and developed as ensembles of concurrent and distributed components, interacting via message-passing according to pre-determined *communication protocols*. A key challenge lies in ensuring that each component abides by the desired protocol, thus avoiding run-time failures due to, e.g., communication errors and deadlocks. One of the most successful approaches to this problem are **session types** [Honda et al. 1998; Takeuchi et al. 1994]. In their original formulation, session types allow formalising two-party protocols (e.g., for client-server interaction), whose structure includes sequencing, choices, and recursion; they were later extended to *multiparty* protocols [Honda et al. 2008, 2016]. By describing *(multiparty) protocols as types*, session types provide a type-based methodology to statically verify whether a given process implements a desired protocol. Beyond the theoretical developments, multiparty session types have been implemented in mainstream programming languages such as Java, Python, Go, Scala, C, TypeScript, F♯, OCaml, Haskell, Erlang [Ancona et al. 2016; Gay and Ravara 2017].

Precise subtyping. The substitution principle of Liskov and Wing [1994] establishes a general notion of *subtyping*: if T is subtype of T', then an object of type T can always replace an object of type T', in any context. Similar notions arise in the realm of process calculi, since Pierce and Sangiorgi [1996] first introduced IO-subtyping for input and output channel capabilities in the π-calculus. As session types are protocols, the notion of *session subtyping* (first introduced by Gay and Hole [2005]) can be interpreted as ***protocol refinement***: given two types/protocols T and T', if T is a subtype (or refinement) of T', then a process that implements T can be used whenever a process implementing T' is needed. In general, when a type system is equipped with a subtyping (subsumption) rule, then we can enlarge the set of typable programs by enlarging its subtyping relation. On the one hand, a larger subtyping can be desirable, since it makes the type system more flexible, and the verification more powerful; however, a subtyping relation that is *too* large makes the type system unsound: e.g., if we consider `string` as a subtype of `real`, then expressions like $1 +$ "`foo`" become typable, and typed programs can crash at run-time. Finding the "right subtyping" (not too strict, nor too lax) leads to the problem of finding a canonical, *precise* subtyping for a given type system — i.e., a subtyping relation that is sound ("typed programs never go wrong") and cannot be further enlarged (otherwise, the type system would become unsound). The problem has been widely studied for the λ-calculus [Blackburn et al. 2012; Dezani-Ciancaglini et al. 1998; Dezani-Ciancaglini and Ghilezan 2014; Ligatti et al. 2017]; several papers have also tackled the problem in the realm of session types [Chen et al. 2017; Ghilezan et al. 2019]. A session subtyping relation $\leqslant$ is ***precise*** when it is both *sound* and *complete*:

soundness means that, if we have a context C expecting some process P' of type T', then $T \leqslant T'$ implies that any process P of type T can be placed into C without causing "bad behaviours" (e.g., communication errors or deadlocks);

completeness means that $\leqslant$ cannot be extended without becoming unsound. More accurately: if $T \nleqslant T'$, then we can find a process P of type T, and a context C expecting a process of type T', such that if we place P in C, it will cause "bad behaviours."

Asynchronous Multiparty Session Subtyping. This work tackles the problem of finding the precise subtyping relation $\leqslant$ for *multiparty asynchronous* session types. The starting point is a type system for processes that *(1)* implement types/protocols with 2 or more participants, and *(2)* communicate through a medium that buffers messages while preserving their order — as in TCP/IP sockets, and akin to the original papers on multiparty session types [Bettini et al. 2008; Honda et al. 2008].

For example, consider a scenario where participant r waits for the outcome of a computation from p, and notifies q on whether to continue the calculation (we omit part of the system "$\cdots$"):

$$\mathsf{r} \triangleleft P_{\mathsf{r}} \mid \mathsf{p} \triangleleft P_{\mathsf{p}} \mid \mathsf{q} \triangleleft P_{\mathsf{q}} \mid \cdots$$

$$\text{where } P_{\mathsf{r}} = \sum \begin{Bmatrix} \mathsf{p}?\mathit{success}(x).\mathsf{if}\ (x > 0)\ \mathsf{then}\ \mathsf{q}!\mathit{cont}\langle x\rangle.\mathbf{0}\ \mathsf{else}\ \mathsf{q}!\mathit{stop}\langle\rangle.\mathbf{0} \\ \mathsf{p}?\mathit{error}(\mathit{fatal}).\mathsf{if}\ (\neg\mathit{fatal})\ \mathsf{then}\ \mathsf{q}!\mathit{cont}\langle 42\rangle.\mathbf{0}\ \mathsf{else}\ \mathsf{q}!\mathit{stop}\langle\rangle.\mathbf{0} \end{Bmatrix}$$

Above, $\mathsf{r} \triangleleft P_{\mathsf{r}}$ denotes a process P_{r} executed by participant r, $\mathsf{p}?\ell(x)$ is an input of message ℓ with payload value x from participant p, and $\mathsf{q}!\ell\langle 5\rangle$ is an output of message ℓ with payload 5 to participant q. In the example, r waits to receive either *success* or *error* from p. In case of *success*, r checks whether the message payload x is greater than 0 (zero), and tells q to either *cont*inue (forwarding the payload x) or *stop*, then terminates ($\mathbf{0}$); in case of *error*, r checks whether the error is non-fatal, and then tells q to either *cont*inue (with a constant value 42), or *stop*. Note that r is blocked until a message is sent by p, and correspondingly, q is waiting for r, who is waiting for p. Yet, depending on the application, one might attempt to *locally optimise* r, by replacing P_{r} above with the process:

$$P'_{\mathsf{r}} = \mathsf{if}\ (\ldots)\ \mathsf{then}\ \mathsf{q}!\mathit{cont}\langle 42\rangle.\sum \begin{Bmatrix} \mathsf{p}?\mathit{success}(x).\mathbf{0} \\ \mathsf{p}?\mathit{error}(y).\mathbf{0} \end{Bmatrix}\ \mathsf{else}\ \mathsf{q}!\mathit{stop}\langle\rangle.\sum \begin{Bmatrix} \mathsf{p}?\mathit{success}(x).\mathbf{0} \\ \mathsf{p}?\mathit{error}(y).\mathbf{0} \end{Bmatrix}$$

Process P'_{r} internally decides (with an omitted condition "$\ldots$") whether to tell q to *cont*inue with a constant value 42, or *stop*. Then, r receives the *success*/*error* message from p, and does nothing with it. As a result, q can start its computation immediately, without waiting for p. Observe that this optimisation permutes the order of inputs and outputs in the process of r: is it "correct"? I.e., could this permutation introduce any deadlock or communication error in the system? Do we have enough information to determine it, or do we need to know the behaviour of P_{p} and P_{q}, and the omitted part ("$\cdots$") of the system? If this optimisation never causes deadlocks or communication errors, then a session subtyping relation should allow for it: i.e., if T' is the type of P'_{r}, and T is the type of P_{r}, we should have $T' \leqslant T$ — hence, the type system should let P'_{r} be used in place of P_{r}. (We illustrate such types later on, in Example 3.9.) Due to practical needs, similar program optimisations have been implemented for various programming languages [Castro-Perez and Yoshida 2020a,b; Hu 2017; Ng et al. 2015, 2012; Yoshida et al. 2008]. Yet, this optimisation is *not* allowed by *synchronous* multiparty session subtyping [Ghilezan et al. 2019]: in fact, under synchronous communication, there are cases where replacing P_{r} with P'_{r} would introduce deadlocks. However, most real-world distributed and concurrent systems use *asynchronous* communication: does asynchrony make the optimisation above *always* safe, and should the subtyping allow for it? If we prove that this optimisation, and others, are indeed sound under asynchrony, it would be possible to check them locally, at the type-level, for each individual participant in a multiparty session.

Contributions. We present the **first formalisation of the *precise* subtyping relation $\leqslant$ for multiparty asynchronous session types**. We introduce the relation (Section 3.2) and prove that it is **operationally precise** (Theorem 5.13), i.e., satisfies the notions of "soundness" and "completeness" outlined above. Then, we use this result as a stepping stone to prove that $\leqslant$ is also **denotationally precise** (Theorem 7.2), and **precise wrt. liveness** (Theorem 7.3).

A key element of our contribution is a novel approach based on **session decomposition**: given two session types T and T', we formalise the subtyping $T \leqslant T'$ as a composition of refinement relations $\lesssim$ over *single-input, single-output (SISO) trees* extracted from T and T'. (The idea and its motivation are explained in Section 3.1.)

We base our development on a recent advancement of the multiparty session types theory [Scalas and Yoshida 2019]: this way, we achieve not only a precise subtyping relation, but also a simpler

$$\mathsf{v} ::= \mathsf{i} \mid \mathsf{true} \mid \mathsf{false} \mid () \qquad \mathsf{i} ::= \mathtt{0} \mid \mathsf{n} \mid -\mathsf{n} \qquad \mathsf{n} ::= 1 \mid \mathsf{n}+1$$
$$\mathsf{e} ::= x \mid \mathsf{v} \mid \mathsf{succ}(\mathsf{e}) \mid \mathsf{inv}(\mathsf{e}) \mid \neg\mathsf{e} \mid \mathsf{e} > \mathtt{0} \mid \mathsf{e} \approx ()$$

$$\begin{array}{rcll}
\mathcal{M} & ::= & & \textbf{Sessions} \\
 & & \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h & \textit{individual participant} \\
 & \mid & \mathcal{M} \mid \mathcal{M} & \textit{parallel} \\
 & \mid & \mathsf{error} & \textit{error} \\
h & ::= & & \textbf{Message queues} \\
 & & \varnothing & \textit{empty queue} \\
 & \mid & (\mathsf{q}, \ell(\mathsf{v})) & \textit{message} \\
 & \mid & h \cdot h & \textit{concatenation}
\end{array}
\qquad
\begin{array}{rcll}
P, Q & ::= & \textbf{Processes} & \\
 & & \sum_{i \in I} \mathsf{p}?\ell_i(x_i).P_i & \textit{external choice} \\
 & \mid & \mathsf{p}!\ell\langle \mathsf{e} \rangle.P & \textit{output} \\
 & \mid & \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q & \textit{conditional} \\
 & \mid & X & \textit{variable} \\
 & \mid & \mu X.P & \textit{recursion} \\
 & \mid & \mathbf{0} & \textit{inaction}
\end{array}$$

Fig. 1. Syntax of values (v), expressions (e), sessions, processes, and queues.

formulation, and more general results than previous work on multiparty session subtyping [Chen et al. 2017; Ghilezan et al. 2019; Mostrous and Yoshida 2015], supporting the verification of a larger set of concurrent and distributed processes and protocols.

To demonstrate the tractability and generality of our subtyping relation, we discuss various examples — including one that is *not* supported by a sound algorithm for asynchronous *binary* session subtyping (Example 3.10), and another where we prove the correctness of a messaging optimisation (based on *double buffering* [Huang et al. 2002, Section 3.2], adapted from [Castro-Perez and Yoshida 2020a; Mostrous et al. 2009; Yoshida et al. 2008]) applied to a distributed data processing scenario (Section 6).

Overview. Section 2 formalises the asynchronous multiparty session calculus. Section 3 presents our asynchronous multiparty session subtyping relation, with its decomposition technique. Section 4 introduces the typing system, proving its soundness. Section 5 proves the completeness and preciseness of our subtyping. Section 6 applies our subtyping to prove the correctness of an asynchronous optimisation of a distributed system. Section 7 proves additional preciseness results: denotational preciseness, and preciseness wrt. liveness. Related work is in Section 8. Proofs and additional examples are available in a separate technical report [Ghilezan et al. 2020].

2 ASYNCHRONOUS MULTIPARTY SESSION CALCULUS

This section formalises the syntax and operational semantics of an asynchronous multiparty session calculus. Our formulation is a streamlined presentation of the session calculus by Bettini et al. [2008], omitting some elements (in particular, session creation and shared channels) to better focus on subtyping. The same design is adopted, e.g., by Ghilezan et al. [2019] — but here we include message queues, for asynchronous (FIFO-based) communication.

2.1 Syntax

The syntax of our calculus is defined in Figure 1. Values and expressions are standard: a **value** v can be an integer i (positive n, negative $-\mathsf{n}$, or zero $\mathtt{0}$) , a boolean true/false, or unit $()$ (that we will often omit, for brevity); an **expression** e can be a variable, a value, or a term built from expressions by applying the operators succ, inv, $\neg$, or the relations $>$, $\approx$.

Asynchronous multiparty sessions (ranged over by $\mathcal{M}, \mathcal{M}', \ldots$) are parallel compositions of individual **participants** (ranged over by $\mathsf{p}, \mathsf{q}, \ldots$) associated with their own **process** P and **message queue** h (notation: $\mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h$). In the processes syntax, the **external choice** $\sum_{i \in I} \mathsf{p}?\ell_i(x_i).P_i$ denotes the input from participant p of a message with label ℓ_i carrying value x_i, for any $i \in I$;

instead, $\mathsf{p}!\ell\langle \mathsf{e}\rangle.P$ denotes the **output** towards participant p of a message with label ℓ carrying the value returned by expression e. The **conditional** $\text{if e then } P \text{ else } Q$ is standard. The term $\mathsf{p} \triangleleft h$ states that h is the **output message queue** of participant p; if a message $(\mathsf{q}, \ell(\mathsf{v}))$ is in the queue of participant p, it means that p has sent $\ell(\mathsf{v})$ to q.[1] Messages are consumed by their recipients on a FIFO (first in, first out) basis. The rest of the syntax is standard [Ghilezan et al. 2019]. We assume that in recursive processes, recursion variables are guarded by external choices and/or outputs.

We also define the set $\mathtt{act}(P)$, containing the **input and output actions** of P; its elements have the form p? or p!, representing an input or an output from/to participant p, respectively:

$$\mathtt{act}(\mathbf{0}) = \emptyset \qquad \mathtt{act}(\mathsf{p}!\ell\langle \mathsf{e}\rangle.P) = \{\mathsf{p}!\} \cup \mathtt{act}(P) \qquad \mathtt{act}(\textstyle\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i) = \{\mathsf{p}?\} \cup \bigcup_{i\in I} \mathtt{act}(P_i)$$
$$\mathtt{act}(\mu X.P') = \mathtt{act}(P') \qquad \mathtt{act}(\text{if e then } P_1 \text{ else } P_2) = \mathtt{act}(P_1) \cup \mathtt{act}(P_2)$$

2.2 Reductions and Errors

First, we give the operational semantics of expressions. To this purpose, we define an **evaluation context** $\mathcal{E}$ as an expression (from Figure 1) with exactly one hole [ ], given by the grammar:

$$\mathcal{E} \ ::= \ [\,] \ \mid \ \mathsf{succ}(\mathcal{E}) \ \mid \ \mathsf{inv}(\mathcal{E}) \ \mid \ \neg\mathcal{E} \ \mid \ \mathcal{E} > 0$$

We write $\mathcal{E}(\mathsf{e})$ for the expression obtained from context $\mathcal{E}$ by filling its unique hole with e. The **value of an expression** is computed as defined in Figure 2: the notation $\mathsf{e} \downarrow \mathsf{v}$ means that expression e evaluates to v. Notice that the successor operation $\mathtt{succ}$ is defined on natural numbers (i.e. positive integers), the inverse operation $\mathtt{inv}$ is defined on integers, and negation $\neg$ is defined on boolean values; moreover, the evaluation of $\mathsf{e} > 0$ is defined only if e evaluates to some integer, while the evaluation of $\mathsf{e} \approx ()$ is defined only if e is exactly the unit value.

The operational semantics of our calculus is defined in Figure 3. By [r-send], a participant p sends $\ell\langle \mathsf{e}\rangle$ to a participant q, enqueuing the message $(\mathsf{q}, \ell(\mathsf{v}))$. Rule [r-rcv] lets participant p receive a message from q: if one of the input labels ℓ_k matches a queued message $(\mathsf{p}, \ell_k(\mathsf{v}))$ previously sent by q (for some $k \in I$), the message is dequeued, and the continuation P_k proceeds with value v substituting x_k. The rules for conditionals are standard. Rule [r-struct] defines the reduction modulo a standard **structural congruence** $\equiv$, defined in Figure 4.

Figure 3 also formalises **error reductions**, modelling the following scenarios: in [err-mism], a process tries to read a queued message with an unsupported label; in [err-orph], there is a queued message from q to p, but p's process does not contain any input from q, hence the message is orphan; in [err-strv], p is waiting for a message from q, but no such message is queued, and q's process does not contain any output for p, hence p will starve; in [err-eval], a condition does not evaluate to a boolean value; in [err-eval2], an expression like "$\mathsf{succ}(\mathsf{true})$" cannot reduce to any value; in [err-dlock], the session cannot reduce further, but at least one participant is expecting an input.

Example 2.1 (Reduction relation). We now describe the operational semantics using the example from the Introduction. Consider the session:

$$\mathsf{r} \triangleleft \sum \left\{ \begin{array}{l} \mathsf{p}?\mathit{success}(x).\text{if } (x > 0) \text{ then } \mathsf{q}!\mathit{cont}\langle x\rangle.\mathbf{0} \text{ else } \mathsf{q}!\mathit{stop}\langle\rangle.\mathbf{0} \\ \mathsf{p}?\mathit{error}(\mathit{fatal}).\text{if } (\neg\mathit{fatal}) \text{ then } \mathsf{q}!\mathit{cont}\langle 42\rangle.\mathbf{0} \text{ else } \mathsf{q}!\mathit{stop}\langle\rangle.\mathbf{0} \end{array} \right\} \ \mid \ \mathsf{r} \triangleleft \emptyset \ \mid \ \mathsf{p} \triangleleft P_{\mathsf{p}} \ \mid \ \mathsf{p} \triangleleft \emptyset \ \mid \ \mathsf{q} \triangleleft P_{\mathsf{q}} \ \mid \ \cdots$$

In this session, the process of r is blocked until a message is sent by p to r. If this cannot happen (e.g., because P_{p} is $\mathbf{0}$), the session will reduce to error by [err-starv]. Now, consider the session above

[1] Alternatively, we could formalise a calculus with actor-style *input* message queues, at the cost of some additional notation; the semantics would be equivalent [Demangeon and Yoshida 2015] and would not influence subtyping.

$$\mathsf{succ}(\mathsf{n}) \downarrow (\mathsf{n}+1) \qquad \mathsf{inv}(\mathsf{n}) \downarrow -\mathsf{n} \qquad \mathsf{inv}(-\mathsf{n}) \downarrow \mathsf{n} \qquad \mathsf{inv}(0) \downarrow 0$$
$$\neg\mathsf{true} \downarrow \mathsf{false} \qquad \neg\mathsf{false} \downarrow \mathsf{true} \qquad (() \approx ()) \downarrow \mathsf{true} \qquad \mathsf{v} \downarrow \mathsf{v}$$
$$\frac{\mathsf{e}_1 \downarrow \mathsf{n}}{(\mathsf{e}_1 > 0) \downarrow \mathsf{true}} \qquad \frac{\mathsf{e}_1 \downarrow -\mathsf{n}}{(\mathsf{e}_1 > 0) \downarrow \mathsf{false}} \qquad \frac{\mathsf{e}_1 \downarrow 0}{(\mathsf{e}_1 > 0) \downarrow \mathsf{false}} \qquad \frac{\mathsf{e} \downarrow \mathsf{v} \quad \mathcal{E}(\mathsf{v}) \downarrow \mathsf{v}'}{\mathcal{E}(\mathsf{e}) \downarrow \mathsf{v}'}$$

Fig. 2. Evaluation rules for expressions.

[R-SEND]	$\mathsf{p} \triangleleft \mathsf{q}!\ell\langle\mathsf{e}\rangle.P \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathcal{M} \longrightarrow \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \cdot (\mathsf{q}, \ell(\mathsf{v})) \mid \mathcal{M}$	$(\mathsf{e} \downarrow \mathsf{v})$
[R-RCV]	$\mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell_k(\mathsf{v})) \cdot h \mid \mathcal{M}$ $\longrightarrow \mathsf{p} \triangleleft P_k\{\mathsf{v}/x_k\} \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft h \mid \mathcal{M}$	$(k \in I)$
[R-COND-T]	$\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \longrightarrow \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h \mid \mathcal{M}$	$(\mathsf{e} \downarrow \mathsf{true})$
[R-COND-F]	$\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \longrightarrow \mathsf{p} \triangleleft Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M}$	$(\mathsf{e} \downarrow \mathsf{false})$
[R-STRUCT]	$\mathcal{M}_1 \equiv \mathcal{M}_1'$ and $\mathcal{M}_1' \longrightarrow \mathcal{M}_2'$ and $\mathcal{M}_2' \equiv \mathcal{M}_2 \implies \mathcal{M}_1 \longrightarrow \mathcal{M}_2$	
[ERR-MISM]	$\mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell(\mathsf{v})) \cdot h \mid \mathcal{M} \longrightarrow \mathsf{error}$	$(\forall i \in I. \ell_i \neq \ell)$
[ERR-OPHN]	$\mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell(\mathsf{v})) \cdot h \mid \mathcal{M} \longrightarrow \mathsf{error}$	$(\mathsf{q}? \notin \mathsf{act}(P))$
[ERR-STRV]	$\mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft h_{\mathsf{q}} \mid \mathcal{M} \longrightarrow \mathsf{error}$	$(\mathsf{p}! \notin \mathsf{act}(Q), h_{\mathsf{q}} \not\equiv (\mathsf{p}, -(-)) \cdot h_{\mathsf{q}}')$
[ERR-EVAL]	$\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \longrightarrow \mathsf{error}$	$(\mathsf{e} \not\downarrow \mathsf{true}$ and $\mathsf{e} \not\downarrow \mathsf{false})$
[ERR-EVAL2]	$\mathsf{p} \triangleleft \mathsf{q}!\ell\langle\mathsf{e}\rangle.P \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \longrightarrow \mathsf{error}$	$(\nexists \mathsf{v} : \mathsf{e} \downarrow \mathsf{v})$
[ERR-DLOCK]	$\prod_{j\in J} \left(\mathsf{p}_j \triangleleft \sum_{i_j\in I_j} \mathsf{q}_j?\ell_{i_j}(x_{i_j}).P_{i_j} \mid \mathsf{p}_j \triangleleft h_{\mathsf{p}_j}\right) \mid \prod_{j\in J'} \left(\mathbf{0} \mid \mathsf{p}_j \triangleleft h_{\mathsf{p}_j}\right)$ $\longrightarrow \mathsf{error}$	$(\forall j \in J \neq \emptyset : \forall i \in J \cup J' :$ $h_{\mathsf{p}_i} \not\equiv (\mathsf{p}_j, -(-)) \cdot h_{\mathsf{p}_i}')$

Fig. 3. Reduction relation on sessions.

$$h_1 \cdot (\mathsf{q}_1, \ell_1(\mathsf{v}_1)) \cdot (\mathsf{q}_2, \ell_2(\mathsf{v}_2)) \cdot h_2 \equiv h_1 \cdot (\mathsf{q}_2, \ell_2(\mathsf{v}_2)) \cdot (\mathsf{q}_1, \ell_1(\mathsf{v}_1)) \cdot h_2 \quad (\text{if } \mathsf{q}_1 \neq \mathsf{q}_2)$$
$$\varnothing \cdot h \equiv h \qquad h \cdot \varnothing \equiv h \qquad h_1 \cdot (h_2 \cdot h_3) \equiv (h_1 \cdot h_2) \cdot h_3 \qquad \mu X.P \equiv P\{\mu X.P/X\}$$
$$\mathsf{p} \triangleleft \mathbf{0} \mid \mathsf{p} \triangleleft \varnothing \mid \mathcal{M} \equiv \mathcal{M} \qquad \mathcal{M}_1 \mid \mathcal{M}_2 \equiv \mathcal{M}_2 \mid \mathcal{M}_1 \qquad (\mathcal{M}_1 \mid \mathcal{M}_2) \mid \mathcal{M}_3 \equiv \mathcal{M}_1 \mid (\mathcal{M}_2 \mid \mathcal{M}_3)$$
$$P \equiv Q \text{ and } h_1 \equiv h_2 \implies \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h_1 \mid \mathcal{M} \equiv \mathsf{p} \triangleleft Q \mid \mathsf{p} \triangleleft h_2 \mid \mathcal{M}$$

Fig. 4. Structural congruence rules for queues, processes, and sessions.

optimised with the process:

$$\mathsf{r} \triangleleft \mathsf{if}\ (\mathsf{e})\ \mathsf{then}\ \mathsf{q}!cont\langle 42\rangle. \sum \begin{Bmatrix} \mathsf{p}?success(x).\mathbf{0} \\ \mathsf{p}?error(y).\mathbf{0} \end{Bmatrix}\ \mathsf{else}\ \mathsf{q}!stop\langle\rangle. \sum \begin{Bmatrix} \mathsf{p}?success(x).\mathbf{0} \\ \mathsf{p}?error(y).\mathbf{0} \end{Bmatrix}$$

This session could reduce to error if e does *not* evaluate to true or false (by [ERR-EVAL]). Instead, if $\mathsf{e} \downarrow \mathsf{true}$, then r can fire [R-COND-T] and [R-SEND], reducing the session to:

$$\mathsf{r} \triangleleft \sum \begin{Bmatrix} \mathsf{p}?success(x).\mathbf{0} \\ \mathsf{p}?error(y).\mathbf{0} \end{Bmatrix} \mid \mathsf{r} \triangleleft (\mathsf{q}, cont(42)) \mid \mathsf{p} \triangleleft P_{\mathsf{p}} \mid \mathsf{p} \triangleleft \varnothing \mid \mathsf{q} \triangleleft P_{\mathsf{q}} \mid \cdots$$

Otherwise, if $\mathsf{e} \downarrow \mathsf{false}$, then r can fire [R-COND-F] and [R-SEND], reducing the session to:

$$\mathsf{r} \triangleleft \sum \begin{Bmatrix} \mathsf{p}?success(x).\mathbf{0} \\ \mathsf{p}?error(y).\mathbf{0} \end{Bmatrix} \mid \mathsf{r} \triangleleft (\mathsf{q}, stop()) \mid \mathsf{p} \triangleleft P_{\mathsf{p}} \mid \mathsf{p} \triangleleft \varnothing \mid \mathsf{q} \triangleleft P_{\mathsf{q}} \mid \cdots$$

In both cases, r reduces to **0** (by [R-RCV]) if it receives a *success*/*error* message from p; meanwhile, if q is ready to receive an input from p, then q can continue by consuming a message from p's output queue. This kind of optimisation will be verified by means of subtyping in the following section.

3 MULTIPARTY SESSION TYPES AND ASYNCHRONOUS SUBTYPING

This section formalises multiparty session types, and introduces our asynchronous session subtyping relation. We begin with the standard definition of (local) session types.

Definition 3.1. The *sorts* S and *session types* $\mathbb{T}$ are defined as follows:

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

with $I \neq \emptyset$, and $\forall i, j \in I$: $i \neq j \implies \ell_i \neq \ell_j$. We assume guarded recursion. We define $\equiv$ as the least congruence such that $\mu\mathbf{t}.\mathbb{T} \equiv \mathbb{T}\{\mu\mathbf{t}.\mathbb{T}/\mathbf{t}\}$. We define $\mathtt{pt}(\mathbb{T})$ as the set of participants occurring in $\mathbb{T}$.

Sorts are the types of values (naturals, integers, booleans, ...). A session type $\mathbb{T}$ describes the behaviour of a participant in a multiparty session. The **branching type** (or **external choice**) $\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i$ denotes waiting for a message from participant p, where (for some $i \in I$) the message has label ℓ_i and carries a payload value of sort S_i; then, the interaction continues by following $\mathbb{T}_i$. The **selection type** (or **internal choice**) $\bigoplus_{i\in I} \mathsf{p}!\ell_i(\mathsf{S}_i).\mathbb{T}_i$ denotes an output toward participant p of a message with label ℓ_i and payload of sort S_i, after which the interaction follows $\mathbb{T}_i$ (for some $i \in I$). Type $\mu\mathbf{t}.\mathbb{T}$ provides **recursion**, binding the *recursion variable* $\mathbf{t}$ in $\mathbb{T}$; the guarded recursion assumption means: in $\mu\mathbf{t}.\mathbb{T}$, we have $\mathbb{T} \neq \mathbf{t}'$ for any $\mathbf{t}'$ (which ensures *contractiveness*). Type end denotes that the participant has concluded its interactions. For brevity, we often omit branch/selection symbols for singleton inputs/outputs, payloads of sort unit, unnecessary parentheses, and trailing ends.

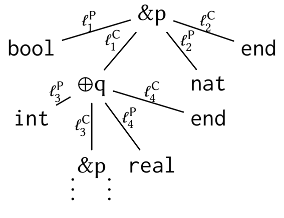
3.1 Session Trees and Their Refinement

Our subtyping is defined in two phases:

(1) we introduce a *refinement relation* $\lesssim$ for *session trees* having only singleton choices in all branchings and selections, called *single-input-single-output (SISO) trees* (definition below);
(2) then, we consider trees that have only singleton choices in branchings (called *single-input (SI) trees*), or in selections (*single-output (SO) trees*), and we define the session subtyping $\leqslant$ over all session types by considering their decomposition into SI, SO, and SISO trees.

This two-phases approach is crucial to capture all input/output reorderings needed by the *precise* subtyping relation, while taming the technical complexity of its formulation. In essence, our session decomposition is a "divide and conquer" technique to separately tackle the main sources of complications in the definition of the subtyping, and in the proofs of preciseness:

- on the one hand, the SISO trees refinement $\lesssim$ focuses on capturing safe permutations and alterations of input/output messages, that never cause deadlocks or communication errors under asynchrony;
- on the other hand, the subtyping relation $\leqslant$ focuses on reconciling the SISO tree refinement $\lesssim$ with the branching structures (i.e., the choices) occurring in session types.



Session trees. To define our subtyping relation, we use (possibly infinite) *session trees* with the standard formulation of [Ghilezan et al. 2019, Appendix A.2], based on Pierce [2002]. The diagram on the right depicts a session tree: its internal nodes represent branching (&p) or selection (⊕q) from/to a participant; leaf nodes are either payload sorts or end; edge annotations are either ℓ^{P} or ℓ^{C}, respectively linking an internal node to the payload or continuation for message ℓ. A type $\mathbb{T}$ yields a tree $\mathcal{T}(\mathbb{T})$: the diagram above shows the (infinite) tree of the type $\mu\mathbf{t}.\,\&\,\big\{\mathsf{p}?\ell_1(\mathtt{bool}).\bigoplus\{\mathsf{q}!\ell_3(\mathtt{int}).\mathbf{t}, \mathsf{q}!\ell_4(\mathtt{real}).\mathsf{end}\}\;,\;\mathsf{p}?\ell_2(\mathtt{nat}).\mathsf{end}\big\}$. Notably, the tree of a recursive type $\mu\mathbf{t}.\mathbb{T}$ coincides with the tree of its unfolding $\mathbb{T}\{\mu\mathbf{t}.\mathbb{T}/\mathbf{t}\}$. We will write T to denote a session tree, and we will represent it using the *coinductive* syntax:

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

The above coinductive definition means that T can be an infinite term, generated by infinite applications of the productions: this approach (previously adopted e.g. by Castagna et al. [2009b]) provides a compact way to represent (possibly infinite) trees.

SISO trees. A SISO tree W only has singleton choices (i.e., one pair of payload+continuation edges) in all its branchings and selections. We represent W with the *coinductive* syntax:

$$\mathsf{W} \quad ::= \quad \mathsf{end} \;\;\mathbf{|}\;\; \mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \;\;\mathbf{|}\;\; \mathsf{p}!\ell(\mathsf{S}).\mathsf{W}$$

We will write $\mathbb{W}$ to denote a SISO session type (i.e., having singleton choice in all its branchings and selections), such that $\mathcal{T}(\mathbb{W})$ yields a SISO tree. We coinductively define the set $\mathtt{act}(\mathsf{W})$ over a tree W as the set of participant names together with actions ? (input) or ! (output), as:

$$\mathtt{act}(\mathsf{end}) = \emptyset \qquad \mathtt{act}(\mathsf{p}?\ell(\mathsf{S}).\mathsf{W}') = \{\mathsf{p}?\} \cup \{\mathtt{act}(\mathsf{W}')\} \qquad \mathtt{act}(\mathsf{p}!\ell(\mathsf{S}).\mathsf{W}') = \{\mathsf{p}!\} \cup \{\mathtt{act}(\mathsf{W}')\}$$

By extension, we also define $\mathtt{act}(\mathbb{W}) = \mathtt{act}(\mathcal{T}(\mathbb{W}))$.

SISO trees refinement. As discussed in Section 1, the asynchronous subtyping should support the reordering the input/output actions of a session type: the intuition is that, under certain conditions, the subtype could *anticipate* some input/output actions occurring in the supertype, by performing them earlier than prescribed. This is crucial to achieve the most flexible and *precise* subtyping. More in detail, such reorderings can have two forms:

R1. anticipating a branching from participant p before a finite number of branchings which are *not* from p;
R2. anticipating a selection toward participant p before a finite number of branchings (from any participant), and *also* before other selections which are *not* toward participant p,

To characterise such reorderings of actions we define two kinds of *finite* sequences of inputs/outputs:

- $\mathcal{A}^{(\mathsf{p})}$, containing only inputs from participants distinct from p (we will use it to formalise reordering R1);
- $\mathcal{B}^{(\mathsf{p})}$, containing inputs from any participant and/or outputs to participants distinct from p (we will use it to formalise reordering R2).

Such sequences are inductively defined by the following productions:

$$\mathcal{A}^{(\mathsf{p})} ::= \mathsf{q}?\ell(\mathsf{S}) \;\mathbf{|}\; \mathsf{q}?\ell(\mathsf{S}).\mathcal{A}^{(\mathsf{p})} \qquad \mathcal{B}^{(\mathsf{p})} ::= \mathsf{r}?\ell(\mathsf{S}) \;\mathbf{|}\; \mathsf{q}!\ell(\mathsf{S}) \;\mathbf{|}\; \mathsf{r}?\ell(\mathsf{S}).\mathcal{B}^{(\mathsf{p})} \;\mathbf{|}\; \mathsf{q}!\ell(\mathsf{S}).\mathcal{B}^{(\mathsf{p})} \quad (\mathsf{q}\neq\mathsf{p})$$

We will use the sequences $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ as prefixes for SISO trees; notice that the base cases require the sequences to have at least one element.

Definition 3.2. We define *subsorting* $\leq:$ as the least reflexive binary relation on sorts (Definition 3.1) such that $\mathtt{nat} \leq: \mathtt{int}$. The *SISO tree refinement relation* $\lesssim$ is coinductively defined as:

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

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

$$\frac{}{\mathsf{end} \lesssim \mathsf{end}}\;[\text{REF-END}]$$

Definition 3.2 above formalises a simulation between SISO trees, as the largest relation closed backward under the given rules [Sangiorgi 2011]. Rule [REF-IN] relates trees beginning with inputs from a same participant, and having equal message labels; the subtyping between carried sorts must be contravariant, and the continuation trees must be related. Rule [REF-OUT] relates trees beginning with outputs to the same participant, and having equal message labels; the subtyping between carried sorts must be covariant, and the continuation trees must be related. Rule [REF-$\mathcal{A}$] captures reordering R1: it enables anticipating an input from participant p before a finite number of inputs from any other participant; the two payload sorts and the rest of the trees satisfy the same conditions as in rule [REF-IN], while "$\mathtt{act}(\mathsf{W}) = \mathtt{act}(\mathcal{A}^{(\mathsf{p})}.\mathsf{W}')$" ensures soundness: without such a condition, the tree refinement relation could "forget" some inputs. (See Example 3.5 below.) Rule [REF-$\mathcal{B}$] captures reordering R2: it enables anticipating an output to participant p before a finite number of inputs from any participant and/or outputs to any other participant; the payload types and the rest of the two trees are related similarly to rule [REF-OUT], while "$\mathtt{act}(\mathsf{W}) = \mathtt{act}(\mathcal{B}^{(\mathsf{p})}.\mathsf{W}')$" ensures that inputs or outputs are not "forgotten," similarly to rule [REF-$\mathcal{A}$]. (See Example 3.5 below.)

LEMMA 3.3. *The refinement relation $\lesssim$ over SISO trees is reflexive and transitive.*

Example 3.4 (SISO tree refinement). Consider the session types:

$$\mathbb{W}_1 = \mu\mathbf{t}.\mathsf{p}?\ell(\mathsf{S}).\mathsf{q}?\ell'(\mathsf{S}').\mathbf{t} \qquad\qquad \mathbb{W}_2 = \mu\mathbf{t}.\mathsf{q}?\ell'(\mathsf{S}').\mathsf{p}?\ell(\mathsf{S}).\mathbf{t}$$

Their trees are related by the following infinite coinductive derivation (notice that the coinductive premise and conclusion coincide); we highlight the inputs matched by rule [REF-$\mathcal{A}$]:

$$\dfrac{\dfrac{\mathcal{T}(\mathbb{W}_1) \lesssim \mathcal{T}(\mathbb{W}_2)}{\mathsf{q}?\ell'(\mathsf{S}').\mathcal{T}(\mathbb{W}_1) \lesssim \mathsf{q}?\ell'(\mathsf{S}').\mathcal{T}(\mathbb{W}_2)}\;[\text{REF-IN}]}{\mathcal{T}(\mathbb{W}_1) = \mathsf{p}?\ell(\mathsf{S}).\mathsf{q}?\ell'(\mathsf{S}').\mathcal{T}(\mathbb{W}_1) \lesssim \mathsf{q}?\ell'(\mathsf{S}').\mathsf{p}?\ell(\mathsf{S}).\mathcal{T}(\mathbb{W}_2) = \mathcal{T}(\mathbb{W}_2)}\;[\text{REF-}\mathcal{A}] \text{ with } \mathcal{A}^{(\mathsf{p})} = \mathsf{q}?\ell'(\mathsf{S}')$$

An example using rule [REF-$\mathcal{B}$] is available later on (Example 3.9).

Example 3.5. We now illustrate why we need the clauses on $\mathtt{act}(...)$ in rules [REF-$\mathcal{A}$] and [REF-$\mathcal{B}$] (Definition 3.2). Consider the following session types:

$$\mathbb{T} = \mu\mathbf{t}.\mathsf{p}?\ell(\mathsf{S}).\mathbf{t} \qquad\qquad \mathbb{T}' = \mathsf{q}?\ell'(\mathsf{S}').\mathbb{T} = \mathsf{q}?\ell'(\mathsf{S}').\mu\mathbf{t}.\mathsf{p}?\ell(\mathsf{S}).\mathbf{t}$$

Observe that $\mathbb{T}$ is "forgetting" to perform the input $\mathsf{q}?\ell'(\mathsf{S}')$ occurring in $\mathbb{T}'$.

If we omit the clause on $\mathtt{act}(...)$ in rule [REF-$\mathcal{A}$], we can construct the following infinite coinductive derivation (notice that the coinductive premise and conclusion coincide; we highlight the inputs matched by rule [REF-$\mathcal{A}$]):

$$\dfrac{\mathcal{T}(\mathbb{T}) \lesssim \mathsf{q}?\ell'(\mathsf{S}').\mathcal{T}(\mathbb{T}) = \mathcal{T}(\mathbb{T}')}{\mathcal{T}(\mathbb{T}) = \mathsf{p}?\ell(\mathsf{S}).\mathcal{T}(\mathbb{T}) \lesssim \mathsf{q}?\ell'(\mathsf{S}').\mathsf{p}?\ell(\mathsf{S}).\mathcal{T}(\mathbb{T}) = \mathcal{T}(\mathbb{T}')}\;[\text{REF-}\mathcal{A}] \text{ with } \mathcal{A}^{(\mathsf{p})} = \mathsf{q}?\ell'(\mathsf{S}')$$

Were we to admit $\mathcal{T}(\mathbb{T}) \lesssim \mathcal{T}(\mathbb{T}')$, then later on (Definition 3.7) we would consider $\mathbb{T}$ a subtype of $\mathbb{T}'$, which would be unsound: in fact, this would allow our type system (Section 4) to type-check processes that "forget" to perform inputs and cause orphan message errors (rule [ERR-OPHN] in Figure 3).

A similar example can be constructed for rule [REF-$\mathcal{B}$], with the following session types:

$$\mathbb{T} = \mu\mathbf{t}.\mathsf{p}!\ell(\mathsf{S}).\mathbf{t} \qquad\qquad \mathbb{T}' = \mathsf{q}!\ell'(\mathsf{S}').\mathbb{T} = \mathsf{q}!\ell'(\mathsf{S}').\mu\mathbf{t}.\mathsf{p}!\ell(\mathsf{S}).\mathbf{t}$$

Now, $\mathbb{T}$ is "forgetting" to perform the output $\mathsf{q}!\ell'(\mathsf{S}')$. If we omit the clause on $\mathtt{act}(...)$ in rule [REF-$\mathcal{B}$], we could derive $\mathcal{T}(\mathbb{T}) \lesssim \mathcal{T}(\mathbb{T}')$ through an infinite sequence of instances of [REF-$\mathcal{B}$] with $\mathcal{B}^{(\mathsf{p})} = \mathsf{q}!\ell'(\mathsf{S}')$; were we to admit this, we would later be able to type-check processes that "forget" to perform outputs and cause starvation errors (rule [ERR-STRV] in Figure 3).

Remark 3.6 (Prefixes vs. n-hole contexts). The binary asynchronous subtyping by Chen et al. [2017, 2014] uses the n-hole branching type context $\mathcal{A} ::= [\ ]^n \mid \&_{i\in I} ?\ell_i(\mathsf{S}_i).\mathcal{A}_i$, which complicates the rules and reasoning (see Fig.2, Fig.3 in Chen et al. [2017]). Our $\mathcal{A}^{(\mathsf{p})}$ and $\mathcal{B}^{(\mathsf{p})}$ have a similar purpose, but they are simpler (just sequences of inputs or outputs), and cater for multiple participants.

SO trees and SI trees. To formalise our subtyping, we need two more kinds of session trees: *single-output (SO) trees*, denoted U, have only singleton choices in their selections; dually, *single-input (SI) trees*, denoted V, have only singleton branchings. We represent them with a *coinductive* syntax:

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

We will write $\mathbb{U}$ (resp. $\mathbb{V}$) to denote a SO (resp. SI) session type, i.e., with only singleton selections (resp. branchings), such that $\mathcal{T}(\mathbb{U})$ (resp. $\mathcal{T}(\mathbb{V})$) yields a SO (resp. SI) tree.

We decompose session trees into their SO/SI subtrees, with the functions $[\![\cdot]\!]_{\mathsf{SO}}$ / $[\![\cdot]\!]_{\mathsf{SI}}$:

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

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

Hence, when $[\![\cdot]\!]_{\mathsf{SO}}$ is applied to a session tree T, it gives the set of all SO trees obtained by taking only a single choice from each selection in T (i.e., we take a single continuation edge and the corresponding payload edge starting in a selection node). The function $[\![\cdot]\!]_{\mathsf{SI}}$ is dual: it takes a single selection from each branching in T. Notice that for any SO tree U, and SI tree V, both $[\![\mathsf{U}]\!]_{\mathsf{SI}}$ and $[\![\mathsf{V}]\!]_{\mathsf{SO}}$ yield SISO trees. We will provide an example shortly (see Example 3.9 below).

3.2 Asynchronous Multiparty Session Subtyping

We can now define our asynchronous session subtyping $\leqslant$: it relates two session types by decomposing them into their SI, SO, and SISO trees, and checking their refinements.

Definition 3.7. The *asynchronous subtyping relation* $\leqslant$ over session trees is defined as:

$$\frac{\forall \mathsf{U} \in [\![\mathsf{T}]\!]_{\mathsf{SO}} \quad \forall \mathsf{V}' \in [\![\mathsf{T}']\!]_{\mathsf{SI}} \quad \exists \mathsf{W} \in [\![\mathsf{U}]\!]_{\mathsf{SI}} \quad \exists \mathsf{W}' \in [\![\mathsf{V}']\!]_{\mathsf{SO}} \qquad \mathsf{W} \lesssim \mathsf{W}'}{\mathsf{T} \leqslant \mathsf{T}'}$$

The subtyping relation for session types is defined as $\mathbb{T} \leqslant \mathbb{T}'$ iff $\mathcal{T}(\mathbb{T}) \leqslant \mathcal{T}(\mathbb{T}')$.

Definition 3.7 says that a session tree T is subtype of T' if, for all SO decompositions of T and all SI decompositions of T', there are paths (i.e., SISO decompositions) related by $\lesssim$.

Lemma 3.8. *The asynchronous subtyping relation $\leqslant$ is reflexive and transitive.*

We now illustrate the relation with two examples: we reprise the scenario in the introduction (Example 3.9), and we discuss a case from [Bravetti et al. 2019a,b] (Example 3.10).

Example 3.9. Consider the opening example in Section 1. The following types describe the interactions of P'_{r} and P_{r}, respectively:

$$\mathbb{T}' = \bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\ \& \ \mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\mathsf{end} \\ \mathit{error}(\mathtt{bool}).\mathsf{end} \end{cases} \\ \mathit{stop}(\mathtt{unit}).\ \& \ \mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\mathsf{end} \\ \mathit{error}(\mathtt{bool}).\mathsf{end} \end{cases} \end{cases} \qquad \mathbb{T} = \& \ \mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\ \bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\mathsf{end} \\ \mathit{stop}(\mathtt{unit}).\mathsf{end} \end{cases} \\ \mathit{error}(\mathtt{bool}).\ \bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\mathsf{end} \\ \mathit{stop}(\mathtt{unit}).\mathsf{end} \end{cases} \end{cases}$$

In order to derive $\mathbb{T}' \leqslant \mathbb{T}$, we first show the two SO trees such that $[\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SO}} = \{\mathsf{U}_1, \mathsf{U}_2\}$:

$$\mathsf{U}_1 = \mathsf{q}!\mathit{cont}(\mathtt{int}).\,\&\,\mathsf{p}?\begin{cases}\mathit{success}(\mathtt{int}).\mathsf{end}\\ \mathit{error}(\mathtt{bool}).\mathsf{end}\end{cases} \qquad \mathsf{U}_2 = \mathsf{q}!\mathit{stop}(\mathtt{unit}).\,\&\,\mathsf{p}?\begin{cases}\mathit{success}(\mathtt{int}).\mathsf{end}\\ \mathit{error}(\mathtt{bool}).\mathsf{end}\end{cases}$$

and these are the two SI trees such that $[\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SI}} = \{\mathsf{V}_1, \mathsf{V}_2\}$:

$$\mathsf{V}_1 = \mathsf{p}?\mathit{success}(\mathtt{int}).\bigoplus \mathsf{q}!\begin{cases}\mathit{cont}(\mathtt{int}).\mathsf{end}\\ \mathit{stop}(\mathtt{unit}).\mathsf{end}\end{cases} \qquad \mathsf{V}_2 = \mathsf{p}?\mathit{error}(\mathtt{bool}).\bigoplus \mathsf{q}!\begin{cases}\mathit{cont}(\mathtt{int}).\mathsf{end}\\ \mathit{stop}(\mathtt{unit}).\mathsf{end}\end{cases}$$

Therefore, for all $i, j \in \{1, 2\}$, we can find $\mathsf{W}' \in [\![\mathsf{U}_i]\!]_{\mathsf{SI}}$ and $\mathsf{W} \in [\![\mathsf{V}_j]\!]_{\mathsf{SO}}$ such that $\mathsf{W}' \lesssim \mathsf{W}$ can be derived using [REF-$\mathcal{B}$]. For instance, since we have:

$$[\![\mathsf{U}_1]\!]_{\mathsf{SI}} = \big\{\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end},\ \mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{error}(\mathtt{bool}).\mathsf{end}\big\}$$
$$[\![\mathsf{V}_1]\!]_{\mathsf{SO}} = \big\{\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{end},\ \mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{stop}(\mathtt{unit}).\mathsf{end}\big\}$$

we can show that $\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end} \lesssim \mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{end}$, by the following coinductive derivation: (we highlight the outputs matched by rule [REF-$\mathcal{B}$])

$$\dfrac{\dfrac{\dfrac{}{\mathsf{end} \lesssim \mathsf{end}}\,[\text{REF-END}]}{\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end} \lesssim \mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end}}\,[\text{REF-IN}]}{\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end} \lesssim \mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{end}}\,[\text{REF-}\mathcal{B}],\ \mathcal{B}^{(\mathsf{q})} = \mathsf{p}?\mathit{success}(\mathtt{int})$$

Hence, by Definition 3.7, we conclude that $\mathbb{T}' \leqslant \mathbb{T}$ holds.

Example 3.10 (Example 3.21 by Bravetti et al. [2019b]). This example demonstrates how our subtyping, and its underlying session decomposition approach, apply to a complex case, where the subtyping proof requires infinite, non-cyclic derivations. Consider the following session types:

$$\mathbb{T} = \mu\mathbf{t}_1.\,\&\,\mathsf{p}?\begin{cases}\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t}_1\\ \ell_2(\mathsf{S}_2).\mu\mathbf{t}_2.\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t}_2\end{cases} \qquad \mathbb{T}' = \mu\mathbf{t}_1.\,\&\,\mathsf{p}?\begin{cases}\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t}_1\\ \ell_2(\mathsf{S}_2).\mu\mathbf{t}_2.\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t}_2\end{cases}$$

We now prove that $\mathbb{T} \leqslant \mathbb{T}'$. Notably, if we omit the participant p, we obtain binary session types that are related under the *binary* asynchronous subtyping relation by Chen et al. [2017] — but due to its complexity, the relation cannot be proved using the binary asynchronous subtyping algorithm of Bravetti et al. [2019a,b].

Letting $\mathsf{T} = \mathcal{T}(\mathbb{T})$ and $\mathsf{T}' = \mathcal{T}(\mathbb{T}')$, by Definition 3.7 we need to show that:

$$\forall \mathsf{U} \in [\![\mathsf{T}]\!]_{\mathsf{SO}} \quad \forall \mathsf{V}' \in [\![\mathsf{T}']\!]_{\mathsf{SI}} \quad \exists \mathsf{W} \in [\![\mathsf{U}]\!]_{\mathsf{SI}} \quad \exists \mathsf{W}' \in [\![\mathsf{V}']\!]_{\mathsf{SO}} \qquad \mathsf{W} \lesssim \mathsf{W}' \tag{1}$$

Observe that both T and T′ are SO trees. Therefore, we have $[\![\mathsf{T}]\!]_{\mathsf{SO}} = \{\mathsf{T}\}$; moreover, all $\mathsf{V}' \in [\![\mathsf{T}']\!]_{\mathsf{SI}}$ are SISO trees, which means that, in (1), for all such V' we have $[\![\mathsf{V}']\!]_{\mathsf{SO}} = \{\mathsf{V}'\}$. These singleton sets allow for simplifying the quantifications in (1), as follows:

$$\forall \mathsf{W}' \in [\![\mathsf{T}']\!]_{\mathsf{SI}} \quad \exists \mathsf{W} \in [\![\mathsf{T}]\!]_{\mathsf{SI}} \qquad \mathsf{W} \lesssim \mathsf{W}' \tag{2}$$

and to prove $\mathbb{T} \leqslant \mathbb{T}'$, it is enough to prove (2). Before proceeding, we introduce the following abbreviations (where W_i are SISO trees, and π_i are sequences of inputs and outputs, used to prefix a SISO tree):

$$\mathsf{W}_1 = \mathcal{T}(\mu\mathbf{t}.\mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t}) \qquad \mathsf{W}_2 = \mathsf{p}?\ell_2(\mathsf{S}_2).\,\mathcal{T}(\mu\mathbf{t}.\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t})$$
$$\mathsf{W}_3 = \mathcal{T}(\mu\mathbf{t}.\mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathbf{t})$$
$$\pi_1 = \mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3) \qquad \pi_3 = \mathsf{p}?\ell_1(\mathsf{S}_1).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3).\mathsf{p}!\ell_3(\mathsf{S}_3) \qquad \pi_i^n = \overbrace{\pi_i.\ldots.\pi_i}^{n\ \text{times}}$$

Then, we have:

$$\begin{array}{rcl}[\![\mathsf{T}]\!]_{\mathsf{SI}} & = & \{\mathsf{W}_3,\ \mathsf{W}_2,\ \pi_3.\mathsf{W}_2,\ \pi_3^2.\mathsf{W}_2,\ \ldots,\ \pi_3^n.\mathsf{W}_2,\ \ldots\}\\ [\![\mathsf{T}']\!]_{\mathsf{SI}} & = & \{\mathsf{W}_1,\ \mathsf{W}_2,\ \pi_1.\mathsf{W}_2,\ \pi_1^2.\mathsf{W}_2,\ \ldots,\ \pi_1^n.\mathsf{W}_2,\ \ldots\}\end{array}$$

and we can prove (2) by showing that:

(i) $W_3 \lesssim W_1$ and

(ii) for all $n \geq 1$, $\pi_3^n.W_2 \lesssim \pi_1^n.W_2$

Here we develop item *(i)*. The relation $W_3 \lesssim W_1$ is proved by the following coinductive derivation:

$$
\begin{array}{cl}
W_3 \lesssim p?\ell_1(S_1).p?\ell_1(S_1).W_1 & \\
\hline\hline
p!\ell_3(S_3).W_3 \lesssim p?\ell_1(S_1).W_1 & [\text{REF-}\mathcal{B}],\ \mathcal{B}^{(p)} = p?\ell_1(S_1).p?\ell_1(S_1) \\
\hline\hline
(p!\ell_3(S_3))^2.W_3 \lesssim W_1 & [\text{REF-}\mathcal{B}],\ \mathcal{B}^{(p)} = p?\ell_1(S_1) \\
\hline\hline
(p!\ell_3(S_3))^3.W_3 \lesssim p!\ell_3(S_3).W_1 & [\text{REF-OUT}] \\
\hline\hline
W_3 \lesssim W_1 & [\text{REF-IN}]
\end{array}
$$

where the topmost relation holds by the following infinite coinductive derivation, for all $n \geq 1$:

$$
\begin{array}{cl}
W_3 \lesssim (p?\ell_1(S_1).p?\ell_1(S_1))^{n+1}.W_1 & \\
\hline\hline
p!\ell_3(S_3).W_3 \lesssim (p?\ell_1(S_1).p?\ell_1(S_1))^n.p?\ell_1(S_1).W_1 & [\text{REF-}\mathcal{B}],\ \mathcal{B}^{(p)} = (p?\ell_1(S_1).p?\ell_1(S_1))^{n+1} \\
\hline\hline
(p!\ell_3(S_3))^2.W_3 \lesssim (p?\ell_1(S_1).p?\ell_1(S_1))^n.W_1 & [\text{REF-}\mathcal{B}],\ \mathcal{B}^{(p)} = (p?\ell_1(S_1).p?\ell_1(S_1))^n.p?\ell_1(S_1) \\
\hline\hline
(p!\ell_3(S_3))^3.W_3 \lesssim (p?\ell_1(S_1).p?\ell_1(S_1))^n.p!\ell_3(S_3).W_1 & [\text{REF-}\mathcal{B}],\ \mathcal{B}^{(p)} = (p?\ell_1(S_1).p?\ell_1(S_1))^n \\
\hline\hline
W_3 \lesssim (p?\ell_1(S_1).p?\ell_1(S_1))^n.W_1 & [\text{REF-IN}]
\end{array}
$$

4 TYPING SYSTEM AND TYPE SAFETY

Our multiparty session typing system blends the one by Ghilezan et al. [2019] with the one in [Scalas and Yoshida 2019, Section 7]: like the latter, we type multiparty sessions without need for global types, thus simplifying our formalism and generalising our results. The key differences are our asynchronous subtyping (Definition 3.7) and our choice of *typing environment liveness* (Definition 4.4): their interplay yields our preciseness results.

4.1 Typing System

Before proceeding, we need to formalise **queue types** for message queues, extending Def. 3.1:

$$\sigma \quad ::= \quad \epsilon \ \mid\ p!\ell(S) \ \mid\ \sigma \cdot \sigma$$

Type ϵ denotes an empty queue; $p!\ell(S)$ denotes a queued message with recipient p, label ℓ, and payload of sort S; they are concatenated as $\sigma \cdot \sigma'$.

Definition 4.1 (Typing system). The type system uses 4 judgments:

- for expressions: $\Theta \vdash e : S$
- for processes: $\Theta \vdash P : \mathbb{T}$
- for queues: $\vdash h : \sigma$
- for sessions: $\Gamma \vdash \mathcal{M}$

where the typing environments Γ and Θ are defined as:

$$\Gamma \ ::= \ \emptyset \mid \Gamma, p : (\sigma, \mathbb{T}) \qquad\qquad \Theta \ ::= \ \emptyset \mid \Theta, X : \mathbb{T} \mid \Theta, x : S$$

The typing system is inductively defined by the rules in Figure 5.

The judgment for expressions is standard: $\Theta \vdash e : S$ means that, given the variables and sorts in environment Θ, expression e is of the sort S. The judgment for queues means that queue h has queue type σ. The judgment for processes states that, given the types of the variables in Θ, process P behaves as prescribed by $\mathbb{T}$. The judgment for sessions states that multiple participants and queues behave as prescribed by Γ, which maps each participant p to the pairing of a queue type (for p's message queue) and a session type (for p's process). If $\Theta = \emptyset$ we write $\vdash e : S$ and $\vdash P : \mathbb{T}$.

We now comment the rules for processes and sessions (other rules are self-explanatory). Rule [T-0] types a terminated process. Rule [T-VAR] types a process variable with the assumption in the environment. By [T-REC], a recursive process is typed with $\mathbb{T}$ if the process variable X and the body

$$\frac{}{\Theta \vdash \mathsf{n} : \mathtt{nat}} \quad \frac{}{\Theta \vdash (-\mathsf{n}) : \mathtt{int}} \quad \frac{}{\Theta \vdash \mathtt{0} : \mathtt{int}} \quad \frac{}{\Theta \vdash \mathsf{true} : \mathtt{bool}} \quad \frac{}{\Theta \vdash \mathsf{false} : \mathtt{bool}}$$

$$\frac{}{\Theta \vdash () : \mathtt{unit}} \quad \frac{}{\Theta, x : \mathsf{S} \vdash x : \mathsf{S}} \quad \frac{\Theta \vdash \mathsf{e} : \mathtt{nat}}{\Theta \vdash \mathsf{succ}(\mathsf{e}) : \mathtt{nat}} \quad \frac{\Theta \vdash \mathsf{e} : \mathtt{int}}{\Theta \vdash \mathsf{inv}(\mathsf{e}) : \mathtt{int}}$$

$$\frac{\Theta \vdash \mathsf{e} : \mathtt{bool}}{\Theta \vdash \neg \mathsf{e} : \mathtt{bool}} \quad \frac{\Theta \vdash \mathsf{e} : \mathtt{int}}{\Theta \vdash \mathsf{e} > \mathtt{0} : \mathtt{bool}} \quad \frac{\Theta \vdash \mathsf{e} : \mathtt{unit}}{\Theta \vdash \mathsf{e} \approx () : \mathtt{bool}} \quad \frac{\Theta \vdash \mathsf{e} : \mathsf{S} \quad \mathsf{S} \leq: \mathsf{S}'}{\Theta \vdash \mathsf{e} : \mathsf{S}'}$$

$$\frac{}{\vdash \varnothing : \epsilon} \quad \frac{\vdash \mathsf{v} : \mathsf{S}}{\vdash (\mathsf{q}, \ell(\mathsf{v})) : \mathsf{q}!\ell(\mathsf{S})} \quad \frac{\vdash h_1 : \sigma_1 \quad \vdash h_2 : \sigma_2}{\vdash h_1 \cdot h_2 : \sigma_1 \cdot \sigma_2}$$

$$\frac{}{\Theta \vdash \mathbf{0} : \mathsf{end}}\,[\text{T-}\mathbf{0}] \quad \frac{}{\Theta, X : \mathbb{T} \vdash X : \mathbb{T}}\,[\text{T-VAR}] \quad \frac{\Theta, X : \mathbb{T} \vdash P : \mathbb{T}}{\Theta \vdash \mu X.P : \mathbb{T}}\,[\text{T-REC}] \quad \frac{\Theta \vdash \mathsf{e} : \mathsf{S} \quad \Theta \vdash P : \mathbb{T}}{\Theta \vdash \mathsf{q}!\ell\langle \mathsf{e}\rangle.P : \mathsf{q}!\ell(\mathsf{S}).\mathbb{T}}\,[\text{T-OUT}]$$

$$\frac{\forall i \in I \quad \Theta, x_i : \mathsf{S}_i \vdash P_i : \mathbb{T}_i}{\Theta \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : \&_{i \in I} \mathsf{q}?\ell_i(\mathsf{S}_i).\mathbb{T}_i}\,[\text{T-EXT}] \quad \frac{\Theta \vdash \mathsf{e} : \mathtt{bool} \quad \Theta \vdash P_i : \mathbb{T}\ (i=1,2)}{\Theta \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \mathbb{T}}\,[\text{T-COND}]$$

$$\frac{\Theta \vdash P : \mathbb{T} \quad \mathbb{T} \leqslant \mathbb{T}'}{\Theta \vdash P : \mathbb{T}'}\,[\text{T-SUB}] \quad \frac{\Gamma = \{\mathsf{p}_i : (\sigma_i, \mathbb{T}_i) \mid i \in I\} \quad \forall i \in I \quad \vdash P_i : \mathbb{T}_i \quad \vdash h_i : \sigma_i}{\Gamma \vdash \prod_{i \in I} (\mathsf{p}_i \triangleleft P_i \mid \mathsf{p}_i \triangleleft h_i)}\,[\text{T-SESS}]$$

Fig. 5. Typing rules for expressions and queues (top 4 rows), and for processes and sessions (bottom 3 rows).

P have the same type $\mathbb{T}$. By [T-OUT], an output process is typed with a singleton selection type, if the message being sent is of the correct sort, and the process continuation has the continuation type. By [T-EXT], a process external choice is typed as a branching type with matching participant p and labels ℓ_i (for all $i \in I$); in the rule premise, each continuation process P_i must be typed with the corresponding continuation type $\mathbb{T}_i$, assuming the bound variable x_i is of sort S_i (for all $i \in I$). By [T-COND], a conditional has type $\mathbb{T}$ if its expression has sort bool, and its "then" and "else" branches have type $\mathbb{T}$. Rule [T-SUB] is the *subsumption rule*: it states that a process of type $\mathbb{T}$ is also typed by any supertype of $\mathbb{T}$ (and since $\leqslant$ relates types up-to unfolding, this rule makes our type system *equi-recursive* [Pierce 2002]). By [T-SESS], a session $\mathcal{M}$ is typed by environment Γ if all the participants in $\mathcal{M}$ have processes and queues typed by the session and queue types pairs in Γ.

Example 4.2. The processes P_{r} and its optimised version P'_{r} in Section 1 are typable using the rules in Def. 4.1, and the types $\mathbb{T}, \mathbb{T}'$ in Example 3.9. In particular, P_{r} has type $\mathbb{T}$, while P'_{r} has type $\mathbb{T}'$. Moreover, since $\mathbb{T}' \leqslant \mathbb{T}$, by rule [T-SUB] the optimised process P'_{r} has *also* type $\mathbb{T}$: hence, our type system allows using P'_{r} whenever a process of type $\mathbb{T}$ (such as P_{r}) is expected.

4.2 Typing Environment Reductions and Liveness

To formulate the soundness result for our type system, we define typing environment reductions. The reductions rely on a standard structural congruence relation $\equiv$ over queue types, allowing to reorder messages with different recipients. Formally, $\equiv$ is the least congruence satisfying:

$$\sigma \cdot \epsilon \equiv \epsilon \cdot \sigma \equiv \sigma \qquad \sigma_1 \cdot (\sigma_2 \cdot \sigma_3) \equiv (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$$
$$\sigma \cdot \mathsf{p}_1!\ell_1(\mathsf{S}_1) \cdot \mathsf{p}_2!\ell_2(\mathsf{S}_2) \cdot \sigma' \equiv \sigma \cdot \mathsf{p}_2!\ell_2(\mathsf{S}_2) \cdot \mathsf{p}_1!\ell_1(\mathsf{S}_1) \cdot \sigma' \quad (\text{if } \mathsf{p}_1 \neq \mathsf{p}_2)$$

For pairs of queue/session types, we define structural congruence as $(\sigma_1, \mathbb{T}_1) \equiv (\sigma_2, \mathbb{T}_2)$ iff $\sigma_1 \equiv \sigma_2$ and $\mathbb{T}_1 \equiv \mathbb{T}_2$, and subtyping as $(\sigma_1, \mathbb{T}_1) \leqslant (\sigma_2, \mathbb{T}_2)$ iff $\sigma_1 \equiv \sigma_2$ and $\mathbb{T}_1 \leqslant \mathbb{T}_2$. We extend subtyping and congruence to typing environments, by requiring all corresponding entries to be related, and allowing additional unrelated entries of type (ϵ, end). More formally, we write

$\Gamma \equiv \Gamma'$ (resp. $\Gamma \leqslant \Gamma'$) iff $\forall \mathsf{p} \in dom(\Gamma) \cap dom(\Gamma')$: $\Gamma(\mathsf{p}) \equiv \Gamma'(\mathsf{p})$ (resp. $\Gamma(\mathsf{p}) \leqslant \Gamma'(\mathsf{p})$) and $\forall \mathsf{p} \in dom(\Gamma) \setminus dom(\Gamma'), \mathsf{q} \in dom(\Gamma') \setminus dom(\Gamma)$: $\Gamma(\mathsf{p}) \equiv (\epsilon, \mathsf{end}) \equiv \Gamma'(\mathsf{q})$.

Definition 4.3 (Typing environment reduction). The reduction $\xrightarrow{\alpha}$ of asynchronous session typing environments is inductively defined as follows, with α being either $\mathsf{p}{:}\mathsf{q}?\ell$ or $\mathsf{p}{:}\mathsf{q}!\ell$ (for some $\mathsf{p}, \mathsf{q}, \ell$):

$$\text{[E-RCV]}\quad \mathsf{p}{:}\, (\mathsf{q}!\ell_k(\mathsf{S}'_k){\cdot}\sigma, \mathbb{T}_\mathsf{p}), \mathsf{q}{:}\, (\sigma_\mathsf{q}, \&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{T}_i), \Gamma \xrightarrow{\mathsf{q}:\mathsf{p}?\ell_k} \mathsf{p}{:}\, (\sigma, \mathbb{T}_\mathsf{p}), \mathsf{q}{:}\, (\sigma_\mathsf{q}, \mathbb{T}_k), \Gamma \qquad (k \in I, \mathsf{S}'_k \leq{:} \mathsf{S}_k)$$

$$\text{[E-SEND]}\quad \mathsf{p} : (\sigma, \textstyle\bigoplus_{i\in I} \mathsf{q}!\ell_i(\mathsf{S}_i).\mathbb{T}_i), \Gamma \xrightarrow{\mathsf{p}:\mathsf{q}!\ell_k} \mathsf{p} : (\sigma{\cdot}\mathsf{q}!\ell_k(\mathsf{S}_k){\cdot}\epsilon, \mathbb{T}_k), \Gamma \qquad (k \in I)$$

$$\text{[E-STRUCT]}\quad \Gamma \equiv \Gamma_1 \xrightarrow{\alpha} \Gamma'_1 \equiv \Gamma' \implies \Gamma \xrightarrow{\alpha} \Gamma'$$

We often write $\Gamma \longrightarrow \Gamma'$ instead of $\Gamma \xrightarrow{\alpha} \Gamma'$ when α is not important. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

Rule [E-RCV] says that an environment can take a reduction step if participant p has a message toward q with label ℓ_k and payload sort S'_k at the head of its queue, while q's type is a branching from p including label ℓ_k and a corresponding sort S_k supertype of S'_k; the environment evolves with a reduction labelled $\mathsf{q}{:}\mathsf{p}?\ell_k$, by consuming p's message and activating the continuation $\mathbb{T}_k$ in q's type. In rule [E-SEND] the environment evolves by letting participant p (having a selection type) send a message toward q; the reduction is labelled $\mathsf{p}{:}\mathsf{q}!\ell_k$ (with ℓ_k being a selection label), and places the message at the end of p's queue. Rule [E-STRUCT] closes the reduction under structural congruence.

Similarly to Scalas and Yoshida [2019], we define a behavioral property of typing environments (and their reductions) called *liveness*:[2] we will use it as a precondition for typing, to ensure that typed processes cannot reduce to any error in Figure 3.

Definition 4.4 (Live typing environment). A *typing environment path* is possibly infinite sequence of typing environments $(\Gamma_i)_{i\in I}$, where $I = \{0, 1, 2, \ldots\}$ is a set of consecutive natural numbers, and, $\forall i \in I, \Gamma_i \longrightarrow \Gamma_{i+1}$. We say that a path $(\Gamma_i)_{i\in I}$ is *fair* iff, $\forall i \in I$:

(F1) whenever $\Gamma_i \xrightarrow{\mathsf{p}:\mathsf{q}!\ell} \Gamma'$, then $\exists k, \ell'$ such that $I \ni k{+}1 > i$, and $\Gamma_k \xrightarrow{\mathsf{p}:\mathsf{q}!\ell'} \Gamma_{k+1}$

(F2) whenever $\Gamma_i \xrightarrow{\mathsf{p}:\mathsf{q}?\ell} \Gamma'$, then $\exists k$ such that $I \ni k{+}1 > i$, and $\Gamma_k \xrightarrow{\mathsf{p}:\mathsf{q}?\ell} \Gamma_{k+1}$

We say that a path $(\Gamma_i)_{i\in I}$ is *live* iff, $\forall i \in I$:

(L1) if $\Gamma_i(\mathsf{p}) \equiv (\mathsf{q}!\ell(\mathsf{S}) \cdot \sigma\,,\ \mathbb{T})$, then $\exists k$: $I \ni k{+}1 > i$ and $\Gamma_k \xrightarrow{\mathsf{q}:\mathsf{p}?\ell} \Gamma_{k+1}$

(L2) if $\Gamma_i(\mathsf{p}) \equiv \left(\sigma_\mathsf{p}\,,\ \&_{j\in J}\, \mathsf{q}?\ell_j(\mathsf{S}_j).\mathbb{T}_j\right)$, then $\exists k, \ell'$: $I \ni k{+}1 > i$ and $\Gamma_k \xrightarrow{\mathsf{p}:\mathsf{q}?\ell'} \Gamma_{k+1}$

We say that a typing environment Γ is *live* iff all fair paths beginning with Γ are live.

By Def. 4.4, a path is a (possibly infinite) sequence of reductions of a typing environment. Intuitively, a fair path represents a "fair scheduling:" along its reductions, every pending internal choice eventually enqueues a message (**F1**), and every pending message reception is eventually performed (**F2**). A path is live if, along its reductions, every queued message is eventually consumed (**L1**), and every waiting external choice eventually consumes a queued message (**L2**). A typing environment is live if it always yields a live path when it is fairly scheduled.

Example 4.5 (Fairness and liveness). Consider the typing environment:

$$\Gamma \ = \ \mathsf{p}{:}(\epsilon, \mu\mathbf{t}.\mathsf{q}!\ell(\mathsf{S}).\mathbf{t}),\ \mathsf{q}{:}(\epsilon, \mu\mathbf{t}.\mathsf{p}?\ell(\mathsf{S}).\mathbf{t})$$

The typing environment Γ has an infinite path where p keeps enqueuing outputs, while q never fires a reduction to receive them: such a path is unfair, because q's message reception is always enabled,

[2] Notably, our definition of liveness is stronger than the "liveness" in [Scalas and Yoshida 2019, Fig. 5], and is closer to "liveness$^+$" therein: we adopt it because a weaker "liveness" would not allow to achieve Theorem 7.3 later on.

but never performed. Instead, in all fair paths of Γ, the messages enqueued by p are eventually consumed by q, and the inputs of q are eventually fired: hence, Γ is live. Now consider:

$$\Gamma' = \mathsf{p}\colon \left(\epsilon, \mu\mathbf{t}. \bigoplus\left\{\mathsf{q}!\ell(\mathsf{S}).\mathbf{t},\ \mathsf{q}!\ell'(\mathsf{S}).\mathsf{r}!\ell'(\mathsf{S}).\mathbf{t}\right\}\right),\ \mathsf{q}\colon \left(\epsilon, \mu\mathbf{t}. \&\left\{\mathsf{p}?\ell(\mathsf{S}).\mathbf{t}, \mathsf{p}?\ell'(\mathsf{S}).\mathbf{t}\right\}\right),\ \mathsf{r}\colon (\epsilon, \mu\mathbf{t}.\mathsf{p}?\ell'(\mathsf{S}).\mathbf{t})$$

The environment Γ' has fair and live paths where p chooses to send ℓ' to q and then to r. However, there is also a fair path where p always chooses to send ℓ to q: in this case, q always eventually receives a message, but r forever waits for an input that will never arrive. Therefore, such a path is fair but not live, hence Γ' is not live. ■

Liveness is preserved by environment reductions and subtyping, as formalised in Proposition 4.6 and Lemma 4.7: these properties are crucial for proving subject reduction later on.

Proposition 4.6. *If Γ is live and $\Gamma \longrightarrow \Gamma'$, then Γ' is live.*

Lemma 4.7. *If Γ is live and $\Gamma' \leqslant \Gamma$, then Γ' is live.*

We can now state our subject reduction result (Theorem 4.8 below): if session $\mathcal{M}$ is typed by a live Γ, and $\mathcal{M} \longrightarrow \mathcal{M}'$, then $\mathcal{M}$ might anticipate some inputs/outputs prescribed by Γ, as allowed by subtyping $\leqslant$. Hence, $\mathcal{M}$ reduces by following some $\Gamma'' \leqslant \Gamma$, which evolves to Γ' that types $\mathcal{M}'$.

Theorem 4.8 (Subject Reduction). *Assume $\Gamma \vdash \mathcal{M}$ with Γ live. If $\mathcal{M} \longrightarrow \mathcal{M}'$, then there are live type environments Γ', Γ'' such that $\Gamma'' \leqslant \Gamma$, $\Gamma'' \longrightarrow^* \Gamma'$ and $\Gamma' \vdash \mathcal{M}'$.*

Corollary 4.9 (Type Safety and Progress). *Let $\Gamma \vdash \mathcal{M}$ with Γ live. Then, $\mathcal{M} \longrightarrow^* \mathcal{M}'$ implies $\mathcal{M}' \neq \mathsf{error}$; also, either $\mathcal{M}' \equiv \mathsf{p} \triangleleft \mathbf{0} \mid \mathsf{p} \triangleleft \varnothing$, or $\exists \mathcal{M}''$ such that $\mathcal{M}' \longrightarrow \mathcal{M}'' \neq \mathsf{error}$.*

Notably, since our errors (Figure 3) include orphan messages, deadlocks, and starvation, Corollary 4.9 implies *session liveness*: a typed session will never deadlock, all its external choices will be eventually activated, all its queued messages will be eventually consumed.[3]

5 PRECISENESS OF ASYNCHRONOUS MULTIPARTY SESSION SUBTYPING

We now present our main result. A subtyping relation $\leqslant$ is *sound* if it satisfies the Liskov and Wing [1994] substitution principle: if $\mathbb{T} \leqslant \mathbb{T}'$, then a process of type $\mathbb{T}'$ engaged in a well-typed session may be safely replaced with a process of type $\mathbb{T}$. The reversed implication is called *completeness*: if it is always safe to replace a process of type $\mathbb{T}'$ with a process of type $\mathbb{T}$, then we should have $\mathbb{T} \leqslant \mathbb{T}'$. If a subtyping $\leqslant$ is both sound and complete, then $\leqslant$ is *precise*. This is formalised in Definition 5.1 below (where we use the contrapositive of the completeness implication).

Definition 5.1 (Preciseness). Let $\trianglelefteq$ be a preorder over session types. We say that $\trianglelefteq$ is:

(1) a **sound subtyping** if $\mathbb{T} \trianglelefteq \mathbb{T}'$ implies that, for all $\mathsf{r} \notin \mathtt{pt}(\mathbb{T}'), \mathcal{M}, P$, the following holds:
  (a) if $\left(\forall Q : \ \vdash Q : \mathbb{T}' \implies \Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M} \text{ for some live } \Gamma\right)$ then $\left(\vdash P : \mathbb{T} \implies (\mathsf{r} \triangleleft P \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M} \longrightarrow^* \mathcal{M}' \text{ implies } \mathcal{M}' \neq \mathsf{error})\right)$
(2) a **complete subtyping** if $\mathbb{T} \ntrianglelefteq \mathbb{T}'$ implies that there are $\mathsf{r} \notin \mathtt{pt}(\mathbb{T}'), \mathcal{M}, P$ such that:
  (a) $\forall Q : \ \vdash Q : \mathbb{T}' \implies \Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}$ for some live Γ
  (b) $\vdash P : \mathbb{T}$
  (c) $\mathsf{r} \triangleleft P \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M} \longrightarrow^* \mathsf{error}$.
(3) a **precise subtyping** if it is both sound and complete.

As customary, our subtyping relation is embedded in the type system via a subsumption rule, giving soundness as an immediate consequence of the subject reduction property.

Theorem 5.2 (Soundness). *The asynchronous multiparty session subtyping $\leqslant$ is sound.*

[3]This assumes that a session is fairly scheduled, similarly to the notion of "fair path" in Definition 4.4.

$$\frac{\mathsf{p}! \notin \mathsf{act}(\mathsf{W}')}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{W}'}\,[\textsc{n-out}] \quad \frac{\mathsf{p}? \notin \mathsf{act}(\mathsf{W}')}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{W}'}\,[\textsc{n-inp}] \quad \frac{\mathsf{p}! \notin \mathsf{act}(\mathsf{W})}{\mathsf{W} \not\lesssim \mathsf{p}!\ell(\mathsf{S}).\mathsf{W}'}\,[\textsc{n-out-R}] \quad \frac{\mathsf{p}? \notin \mathsf{act}(\mathsf{W})}{\mathsf{W} \not\lesssim \mathsf{p}?\ell(\mathsf{S}).\mathsf{W}'}\,[\textsc{n-inp-R}]$$

$$\frac{\ell \neq \ell'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}?\ell'(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-inp-}\ell] \quad \frac{\mathsf{S}' \not\leq: \mathsf{S}}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-inp-S}] \quad \frac{\mathsf{S}' \leq: \mathsf{S} \quad \mathsf{W} \not\lesssim \mathsf{W}'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-inp-W}]$$

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

$$\frac{\mathsf{S}' \leq: \mathsf{S} \quad \mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{W}'}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-}\mathcal{A}\textsc{-W}] \quad \frac{}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-i-o-1}] \quad \frac{}{\mathsf{p}?\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathcal{A}^{(\mathsf{p})}.\mathsf{q}!\ell'(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-i-o-2}]$$

$$\frac{\ell \neq \ell'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}!\ell'(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-out-}\ell] \quad \frac{\mathsf{S} \not\leq: \mathsf{S}'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-out-S}] \quad \frac{\mathsf{S} \leq: \mathsf{S}' \quad \mathsf{W} \not\lesssim \mathsf{W}'}{\mathsf{p}!\ell(\mathsf{S}).\mathsf{W} \not\lesssim \mathsf{p}!\ell(\mathsf{S}').\mathsf{W}'}\,[\textsc{n-out-W}]$$

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

Fig. 6. The relation $\not\lesssim$ between SISO trees.

Proof. Take any $\mathbb{T}, \mathbb{T}'$ such that $\mathbb{T} \leqslant \mathbb{T}'$, and $\mathsf{r}, \mathcal{M}$ satisfying the following condition:

$$\forall Q : \ \vdash Q : \mathbb{T}' \implies \Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M} \ \text{ for some live } \Gamma \tag{3}$$

If $\vdash P : \mathbb{T}$, we derive by [t-sub] that $\vdash P : \mathbb{T}'$ holds. By (3), $\Gamma \vdash \mathsf{r} \triangleleft P \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}$ for some live Γ. Hence, by Corollary 4.9, $\mathsf{r} \triangleleft P \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M} \longrightarrow^* \mathcal{M}'$ implies $\mathcal{M}' \neq \text{error}$. □

The proof of completeness of $\leqslant$ is much more involved. We show that $\leqslant$ satisfies item *(2)* of Definition 5.1 in 4 steps, that we develop in the next sections:

[Step 1] We define the *negation $\not\lesssim$ of the SISO trees refinement relation* by an inductive definition, thus getting a clear characterisation of the complement $\not\lesssim$ of the subtyping relation, that is necessary for Step 2. In addition, for every pair $\mathbb{T}, \mathbb{T}'$ such that $\mathbb{T} \not\leqslant \mathbb{T}'$, we choose a pair $\mathbb{U}, \mathbb{V}'$ satisfying $\mathbb{U} \not\lesssim \mathbb{V}'$ and $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}}$ and $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SI}}$.

[Step 2] We define for every $\mathbb{U}$ a *characteristic process* $\mathcal{P}(\mathbb{U})$. Moreover, we prove that if $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}}$ then we have $\vdash \mathcal{P}(\mathbb{U}) : \mathbb{T}$.

[Step 3] For every $\mathbb{V}'$ with $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SI}}$, and for every participant $\mathsf{r} \notin \mathsf{pt}(\mathbb{V}')$, we define a *characteristic session* $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$, which is typable if composed with a process Q of type $\mathbb{T}'$:

$$\forall Q : \ \vdash Q : \mathbb{T}' \implies \Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'} \ \text{ for some live } \Gamma.$$

[Step 4] Finally, we show that for all $\mathbb{U}, \mathbb{V}'$ such that $\mathbb{U} \not\lesssim \mathbb{V}'$, the characteristic session $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$ (Step 3) reduces to error if composed with the characteristic process of $\mathbb{U}$ (Step 2):

$$\mathsf{r} \triangleleft \mathcal{P}(\mathbb{U}) \ \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'} \ \longrightarrow^* \ \text{error}.$$

Hence, we prove the completeness of $\leqslant$ by showing that, for all $\mathbb{T}, \mathbb{T}'$ such that $\mathbb{T} \not\leqslant \mathbb{T}'$, we can find $\mathsf{r} \notin \mathsf{pt}(\mathbb{T}')$, $P = \mathcal{P}(\mathbb{U})$ (Steps 1,2), and $\mathcal{M} = \mathcal{M}_{\mathsf{r},\mathbb{V}'}$ (Step 3) satisfying Def. 5.1*(2)* (Step 4). We now illustrate each step in more detail.

5.1 Step 1: Subtyping Negation

In Figure 6 we *inductively* define the relation $\not\lesssim$ over SISO trees: it contains all pairs of SISO trees that are *not* related by $\lesssim$, as stated in Lemma 5.3 below. This step is necessary because, in Step 2 (Section 5.2), we will need the shape of the types related by $\not\lesssim$ in order to generate some corresponding processes.

The first category of rules checks whether two SISO trees have a direct syntactic mismatch: whether their sets of actions are disjoint ([N-OUT], [N-INP], [N-OUT-R], [N-INP-R]); the label of the LHS is not equal to the label of the RHS ([N-INP-ℓ], [N-OUT-ℓ]); or matching labels are followed by mismatching sorts or continuations ([N-INP-S], [N-OUT-S], [N-INP-W], [N-OUT-W]).

The second category checks more subtle cases related to asynchronous permutations; rule [N-$\mathcal{A}$-ℓ] checks a label mismatch when the input on the RHS is preceded by a finite number of inputs from other participant; similarly, rules [N-$\mathcal{A}$-S] and [N-$\mathcal{A}$-W] check mismatching sorts or continuations. Rules [N-I-O-1] and [N-I-O-2] formulate the cases such that the top prefix on the LHS is input and the top sequence of prefixes on the RHS consists of a finite number of inputs from other participants and/or outputs. Finally, rules [N-$\mathcal{B}$-ℓ], [N-$\mathcal{B}$-S] and [N-$\mathcal{B}$-W] check the cases of label mismatch, or matching labels followed by mismatching sorts, or continuations of the two types with output prefixes targeting a same participant, where the RHS is prefixed by a finite number of outputs (to other participants) and/or inputs (to any participant).

LEMMA 5.3. *Take any pair of SISO trees* W *and* W'*. Then,* $\mathsf{W} \lesssim \mathsf{W}'$ *is* not *derivable* if and only if $\mathsf{W} \not\lesssim \mathsf{W}'$ *is derivable with the rules in Figure 6.*

PROOF. We adopt an approach inspired by Blackburn et al. [2012] and Ligatti et al. [2017].

For the "$\Rightarrow$" direction of the statement, assume that $\mathsf{W} \lesssim \mathsf{W}'$ is *not* derivable: this means that, if we attempt to build a derivation by applying the rules in Definition 3.2, starting with "$\mathsf{W} \lesssim \mathsf{W}'$" and moving upwards, we obtain a *failing derivation* that, after a finite number n of rule applications, reaches a (wrong) judgement "$\mathsf{W}_n \lesssim \mathsf{W}'_n$" on which no rule of Definition 3.2 can be further applied. Then, by induction on n, we transform such a failing derivation into an *actual* derivation based on the rules for $\not\lesssim$ in Figure 6, which proves $\mathsf{W} \not\lesssim \mathsf{W}'$.

For the "$\Leftarrow$" direction, assume that $\mathsf{W} \not\lesssim \mathsf{W}'$ holds, by the rules in Figure 6: from its derivation, we construct a failing derivation that starts with "$\mathsf{W} \lesssim \mathsf{W}'$" and is based on the rules of Definition 3.2 — which implies that $\mathsf{W} \lesssim \mathsf{W}'$ is *not* derivable. □

It is immediate from Definition 3.7 and Lemma 5.3 that $\mathbb{T}$ is *not* a subtype of $\mathbb{T}'$, written $\mathbb{T} \not\leqslant \mathbb{T}'$, if and only if:

$$\exists \mathsf{U} \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}} \;\; \exists \mathsf{V}' \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SI}} \;\; \forall \mathsf{W} \in [\![\mathsf{U}]\!]_{\mathsf{SI}} \;\; \forall \mathsf{W}' \in [\![\mathsf{V}']\!]_{\mathsf{SO}} \;\; \mathsf{W} \not\lesssim \mathsf{W}' \tag{4}$$

Moreover, we prove that whenever $\mathbb{T} \not\leqslant \mathbb{T}'$, we can find *regular, syntax-derived* SO/SI trees usable as the witnesses U, V' in (4). Thus, from this result and by (4), $\mathbb{T} \not\leqslant \mathbb{T}'$ implies:

$$\exists \mathbb{U}, \mathbb{V}' : \; \mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}} \;\; \mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SI}} \;\; \forall \mathsf{W} \in [\![\mathcal{T}(\mathbb{U})]\!]_{\mathsf{SI}} \;\; \forall \mathsf{W}' \in [\![\mathcal{T}(\mathbb{V}')]\!]_{\mathsf{SO}} \;\; \mathsf{W} \not\lesssim \mathsf{W}' \tag{5}$$

Example 5.4. Consider the example in Section 1, and its types $\mathbb{T}'$ and $\mathbb{T}$ in Example 3.9:

$$\mathbb{T}' = \bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\,\&\,\mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\mathtt{end} \\ \mathit{error}(\mathtt{bool}).\mathtt{end} \end{cases} \\ \mathit{stop}(\mathtt{unit}).\,\&\,\mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\mathtt{end} \\ \mathit{error}(\mathtt{bool}).\mathtt{end} \end{cases} \end{cases} \qquad \mathbb{T} = \&\,\mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\mathtt{end} \\ \mathit{stop}(\mathtt{unit}).\mathtt{end} \end{cases} \\ \mathit{error}(\mathtt{bool}).\bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\mathtt{end} \\ \mathit{stop}(\mathtt{unit}).\mathtt{end} \end{cases} \end{cases}$$

We have seen that $\mathbb{T}' \leqslant \mathbb{T}$ holds (Example 3.9), and thus, by subsumption, our type system allows to use the optimised process P'_{r} in place of P_{r} (Example 4.2). We now show that the inverse relation does *not* hold, i.e., $\mathbb{T} \not\leqslant \mathbb{T}'$, hence the inverse process replacement is disallowed. Take, e.g., $\mathbb{U}, \mathbb{V}'$ as follows, noticing that $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}}$ and $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathsf{SI}}$:

$$\mathbb{U} = \&\,\mathsf{p}? \begin{cases} \mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathtt{end} \\ \mathit{error}(\mathtt{bool}).\mathsf{q}!\mathit{stop}(\mathtt{unit}).\mathtt{end} \end{cases} \qquad \mathbb{V}' = \bigoplus \mathsf{q}! \begin{cases} \mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{success}(\mathtt{int}).\mathtt{end} \\ \mathit{stop}(\mathtt{unit}).\mathsf{p}?\mathit{error}(\mathtt{bool}).\mathtt{end} \end{cases}$$

For all $\mathsf{W} \in [\![\mathcal{T}(\mathbb{U})]\!]_{\mathsf{SI}} = \{\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{end},\ \mathsf{p}?\mathit{error}(\mathtt{bool}).\mathsf{q}!\mathit{stop}(\mathtt{unit}).\mathsf{end}\}$ and all $\mathsf{W}' \in [\![\mathcal{T}(\mathbb{V}')]\!]_{\mathsf{SO}} = \{\mathsf{q}!\mathit{cont}(\mathtt{int}).\mathsf{p}?\mathit{success}(\mathtt{int}).\mathsf{end},\ \mathsf{q}!\mathit{stop}(\mathtt{unit}).\mathsf{p}?\mathit{error}(\mathtt{bool}).\mathsf{end}\}$ we get by [N-I-O-1] that $\mathsf{W} \not\lesssim \mathsf{W}'$. Therefore, we conclude $\mathbb{T} \not\leqslant \mathbb{T}'$.

5.2 Step 2: Characteristic Processes

For any SO type $\mathbb{U}$, we define a characteristic process $\mathcal{P}(\mathbb{U})$ (Def. 5.5): intuitively, it is a process constructed to communicate as prescribed by $\mathbb{U}$, and to be typable by $\mathbb{U}$.

Definition 5.5. The characteristic process $\mathcal{P}(\mathbb{U})$ of type $\mathbb{U}$ is defined inductively as follows:

$$\mathcal{P}(\mathsf{end}) = \mathbf{0} \qquad \mathcal{P}(\mathbf{t}) = X_{\mathbf{t}} \qquad \mathcal{P}(\mu\mathbf{t}.\mathbb{U}) = \mu X_{\mathbf{t}}.\mathcal{P}(\mathbb{U}) \qquad \mathcal{P}(\mathsf{p}!\ell(\mathsf{S}).\mathbb{U}) = \mathsf{p}!\ell\langle \underline{\mathbf{val}(\!|\mathsf{S}|\!)}\rangle.\mathcal{P}(\mathbb{U})$$

$$\mathcal{P}\big(\textstyle\&_{i\in I}\, \mathsf{p}?\ell_i(\mathsf{S}_i).\mathbb{U}_i\big) \ = \ \sum_{i\in I} \mathsf{p}?\ell_i(x_i).\mathsf{if}\ \underline{\mathbf{expr}(\!|x_i,\mathsf{S}_i|\!)}\ \mathsf{then}\ \mathcal{P}(\mathbb{U}_i)\ \mathsf{else}\ \mathcal{P}(\mathbb{U}_i)$$

where:

$$\underline{\mathbf{val}(\!|\mathtt{nat}|\!)} = 1 \quad \underline{\mathbf{val}(\!|\mathtt{int}|\!)} = -1 \quad \underline{\mathbf{val}(\!|\mathtt{bool}|\!)} = \mathsf{true} \quad \underline{\mathbf{val}(\!|\mathtt{unit}|\!)} = () \quad \underline{\mathbf{expr}(\!|x,\mathtt{unit}|\!)} = (x \approx ())$$

$$\underline{\mathbf{expr}(\!|x,\mathtt{bool}|\!)} = (\neg x) \quad \underline{\mathbf{expr}(\!|x,\mathtt{nat}|\!)} = (\mathsf{succ}(x) > 0) \quad \underline{\mathbf{expr}(\!|x,\mathtt{int}|\!)} = (\mathsf{inv}(x) > 0)$$

By Definition 5.5, for every output in $\mathbb{U}$, the characteristic $\mathcal{P}(\mathbb{U})$ sends a value $\underline{\mathbf{val}(\!|\mathsf{S}|\!)}$ of the expected sort S; and for every external choice in $\mathbb{U}$, $\mathcal{P}(\mathbb{U})$ performs a branching, and uses any received value x_i of sort S_i in a boolean expression $\underline{\mathbf{expr}(\!|x_i,\mathsf{S}_i|\!)}$: the expression will cause an error if the value of x_i is not of sort S_i.

Crucially, for all $\mathbb{T}$ and $\mathbb{U}$ such that $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}}$ (e.g., from (5) above), we have $\mathbb{U} \leqslant \mathbb{T}$: therefore, $\mathcal{P}(\mathbb{U})$ is also typable by $\mathbb{T}$, as per Proposition 5.6 below.

PROPOSITION 5.6. *For all closed types $\mathbb{T}$ and $\mathbb{U}$, if $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathsf{SO}}$ then $\vdash \mathcal{P}(\mathbb{U}) : \mathbb{T}$.*

5.3 Step 3: Characteristic Session

The next step to prove completeness is to define for each session type $\mathbb{V}'$ and participant $\mathsf{r} \notin \mathsf{pt}(\mathbb{V}')$ a *characteristic session* $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$, that is well typed (with a live typing environment) when composed with participant r associated with a process of type $\mathbb{V}'$ and empty queue.

For a SI type $\mathbb{V}'$ and $\mathsf{r} \notin \mathsf{pt}(\mathbb{V}') = \{\mathsf{p}_1,\ldots,\mathsf{p}_m\}$, we define m *characteristic SO session types* where participants $\mathsf{p}_1,\ldots,\mathsf{p}_m$ are engaged in a live multiparty interaction with r, and with each other. Definition 5.7 ensures that after each communication between r and some $\mathsf{p} \in \mathsf{pt}(\mathbb{V}')$, there is a cyclic sequence of communications starting with p, involving *all* other $\mathsf{q} \in \mathsf{pt}(\mathbb{V}')$, and ending with p — with each participant acting both as receiver, and as sender.

Definition 5.7. Let $\mathbb{V}'$ be a SI session type and $\mathsf{r} \notin \mathsf{pt}(\mathbb{V}') = \{\mathsf{p}_1,\ldots,\mathsf{p}_m\}$. For every $k \in \{1,\ldots,m\}$, if $m \geq 2$ we define a *characteristic SO session type* $\mathsf{cyclic}(\mathbb{V}',\mathsf{p}_k,\mathsf{r})$ as follows:

$$\begin{array}{lll}
\mathsf{cyclic}(\mathsf{end},\mathsf{p}_k,\mathsf{r}) & = \mathsf{end} & \\
\mathsf{cyclic}(\mathbf{t},\mathsf{p}_k,\mathsf{r}) & = \mathbf{t} & \\
\mathsf{cyclic}(\mu\mathbf{t}.\mathbb{V}_1'',\mathsf{p}_k,\mathsf{r}) & = \mu\mathbf{t}.\mathsf{cyclic}(\mathbb{V}_1'',\mathsf{p}_k,\mathsf{r}) & \\
\mathsf{cyclic}(\mathsf{p}_k?\ell(\mathsf{S}).\mathbb{V}',\mathsf{p}_k,\mathsf{r}) & = \mathsf{r}!\ell(\mathsf{S}).\mathsf{p}_{k+1}!\ell(\mathtt{bool}).\mathsf{p}_{k-1}?\ell(\mathtt{bool}).\mathsf{cyclic}(\mathbb{V}',\mathsf{p}_k,\mathsf{r}) & \\
\mathsf{cyclic}(\mathsf{q}?\ell(\mathsf{S}).\mathbb{V}',\mathsf{p}_k,\mathsf{r}) & = \mathsf{p}_{k-1}?\ell(\mathtt{bool}).\mathsf{p}_{k+1}!\ell(\mathtt{bool}).\mathsf{cyclic}(\mathbb{V}',\mathsf{p}_k,\mathsf{r}) & (\text{if } \mathsf{q} \neq \mathsf{p}_k) \\
\mathsf{cyclic}(\bigoplus_{j\in J}\mathsf{p}_k!\ell_j(\mathsf{S}_j).\mathbb{V}_j',\mathsf{p}_k,\mathsf{r}) & = \&_{j\in J}\,\mathsf{r}?\ell_j(\mathsf{S}_j).\mathsf{p}_{k+1}!\ell_j(\mathtt{bool}).\mathsf{p}_{k-1}?\ell_j(\mathtt{bool}).\mathsf{cyclic}(\mathbb{V}_j',\mathsf{p}_k,\mathsf{r}) & \\
\mathsf{cyclic}(\bigoplus_{j\in J}\mathsf{q}!\ell_j(\mathsf{S}_j).\mathbb{V}_j',\mathsf{p}_k,\mathsf{r}) & = \&_{j\in J}\,\mathsf{p}_{k-1}?\ell_j(\mathtt{bool}).\mathsf{p}_{k+1}!\ell_j(\mathtt{bool}).\mathsf{cyclic}(\mathbb{V}_j',\mathsf{p}_k,\mathsf{r}) & (\text{if } \mathsf{q} \neq \mathsf{p}_k)
\end{array}$$

If $m=1$ (i.e., if there is only one participant in $\mathbb{V}'$) we define $\mathsf{cyclic}(\mathbb{V}',\mathsf{p}_1,\mathsf{r})$ as above, but we omit the (highlighted) cyclic communications, and the cases with $\mathsf{q} \neq \mathsf{p}_k$ do not apply.

Example 5.8 (Characteristic session types). Consider the following SI type:

$$\mathbb{V}' \ = \ \mu\mathbf{t}.\bigoplus\{\mathsf{q}!\ell_2(\mathtt{nat}).\mathsf{p}?\ell_1(\mathtt{nat}).\mathbf{t},\ \mathsf{q}!\ell_3(\mathtt{nat}).\mathsf{p}?\ell_4(\mathtt{nat}).\mathbf{t}\}$$

Let $\mathsf{r} \notin \mathtt{pt}(\mathbb{V}')$. The characteristic session types for participants $\mathsf{p}, \mathsf{q} \in \mathtt{pt}(\mathbb{V}')$ are:

$$\mathtt{cyclic}(\mathbb{V}', \mathsf{p}, \mathsf{r}) = \mu\mathbf{t}. \& \begin{cases} \mathsf{q}?\ell_2(\mathtt{bool}).\mathsf{q}!\ell_2(\mathtt{bool}).\mathsf{r}!\ell_1(\mathtt{nat}).\mathsf{q}!\ell_1(\mathtt{bool}).\mathsf{q}?\ell_1(\mathtt{bool}).\mathbf{t} \\ \mathsf{q}?\ell_3(\mathtt{bool}).\mathsf{q}!\ell_3(\mathtt{bool}).\mathsf{r}!\ell_4(\mathtt{nat}).\mathsf{q}!\ell_4(\mathtt{bool}).\mathsf{q}?\ell_4(\mathtt{bool}).\mathbf{t} \end{cases}$$

$$\mathtt{cyclic}(\mathbb{V}', \mathsf{q}, \mathsf{r}) = \mu\mathbf{t}. \& \begin{cases} \mathsf{r}?\ell_2(\mathtt{nat}).\mathsf{p}!\ell_2(\mathtt{bool}).\mathsf{p}?\ell_2(\mathtt{bool}).\mathsf{p}?\ell_1(\mathtt{bool}).\mathsf{p}!\ell_1(\mathtt{bool}).\mathbf{t} \\ \mathsf{r}?\ell_3(\mathtt{nat}).\mathsf{p}!\ell_3(\mathtt{bool}).\mathsf{p}?\ell_3(\mathtt{bool}).\mathsf{p}?\ell_4(\mathtt{bool}).\mathsf{p}!\ell_4(\mathtt{bool}).\mathbf{t} \end{cases}$$

Note that if r follows type $\mathbb{V}'$, then it must select and send to q one message between ℓ_2 and ℓ_3; correspondingly, the characteristic session type for q receives the message (with a branching), and propagates it to p, who sends it back to q (cyclic communication). Then, r waits for a message from p (either ℓ_1 or ℓ_4, depending on the previous selection): correspondingly, the characteristic session type for p will send such a message, and also propagate it to q with a cyclic communication. ■

Given an SI type $\mathbb{V}'$, we can use Def. 5.7 to construct the following typing environment:

$$\Gamma = \{\mathsf{r}\colon(\epsilon, \mathbb{V}')\} \cup \{\mathsf{p}\colon(\epsilon, \mathbb{U}_\mathsf{p}) \mid \mathsf{p} \in \mathtt{pt}(\mathbb{V}')\} \quad \text{where } \forall \mathsf{p} \in \mathtt{pt}(\mathbb{V}')\colon \mathbb{U}_\mathsf{p} = \mathtt{cyclic}(\mathbb{V}', \mathsf{p}, \mathsf{r}) \qquad (6)$$

i.e., we compose $\mathbb{V}'$ with the characteristic session types of all its participants. The cyclic communications of Def. 5.7 ensure that Γ is live. We can use Γ to type the composition of a process for r, of type $\mathbb{V}'$, together with the characteristic processes of the characteristic session types of each participants in $\mathbb{V}'$: we call such processes the *characteristic session* $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$. This is formalised in Def. 5.9 and Prop. 5.10 below.

Definition 5.9. For any SI type $\mathbb{V}'$ and $\mathsf{r} \notin \mathtt{pt}(\mathbb{V}')$, we define the *characteristic session*:

$$\mathcal{M}_{\mathsf{r},\mathbb{V}'} = \prod_{\mathsf{p} \in \mathtt{pt}(\mathbb{V}')} \left(\mathsf{p} \triangleleft \mathcal{P}(\mathbb{U}_\mathsf{p}) \mid \mathsf{p} \triangleleft \varnothing\right) \qquad \text{where } \forall \mathsf{p} \in \mathtt{pt}(\mathbb{V}') : \mathbb{U}_\mathsf{p} = \mathtt{cyclic}(\mathbb{V}', \mathsf{p}, \mathsf{r})$$

Proposition 5.10. *Let $\mathbb{V}'$ be a SI type and $\mathsf{r} \notin \mathtt{pt}(\mathbb{V}')$. Let Q be a process such that $\vdash Q : \mathbb{V}'$. Then, there is a live typing environment Γ (see (6)) such that $\Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'}$.*

Crucially, for all $\mathbb{T}'$ and $\mathbb{V}'$ such that $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathrm{SI}}$ (e.g., from (5) above), we have $\mathbb{T}' \leqslant \mathbb{V}'$. Thus, by subsumption, $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$ is also typable with a process of type $\mathbb{T}'$ (Proposition 5.11).

Proposition 5.11. *Take any $\mathbb{T}'$, $\mathsf{r} \notin \mathtt{pt}(\mathbb{T}')$, SI type $\mathbb{V}'$ such that $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathrm{SI}}$, and Q such that $\vdash Q : \mathbb{T}'$. Then, there is a live Γ (see (6)) such that $\Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'}$.*

5.4 Step 4: Completeness

This final step of our completeness proof encompasses all elements introduced thus far.

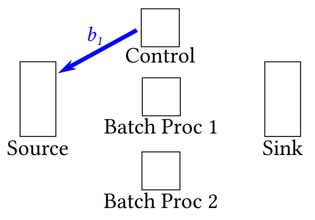
Proposition 5.12. *Let $\mathbb{T}$ and $\mathbb{T}'$ be session types such that $\mathbb{T} \nleqslant \mathbb{T}'$. Take any $\mathsf{r} \notin \mathtt{pt}(\mathbb{T}')$. Then, there are $\mathbb{U}$ and $\mathbb{V}'$ with $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathrm{SO}}$ and $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathrm{SI}}$ and $\mathbb{U} \nleqslant \mathbb{V}'$ such that:*

(1) $\forall Q : \vdash Q : \mathbb{T}' \implies \Gamma \vdash \mathsf{r} \triangleleft Q \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'}$ for some live Γ; (by (6) and Prop. 5.11)
(2) $\vdash \mathcal{P}(\mathbb{U}) : \mathbb{T}$; (by Prop. 5.6)
(3) $\mathsf{r} \triangleleft \mathcal{P}(\mathbb{U}) \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'} \longrightarrow^* \mathtt{error}$.

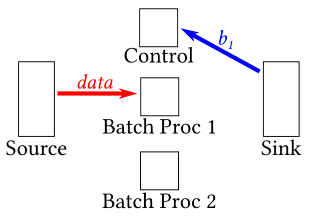
Intuitively, we obtain item 3 of Proposition 5.12 because the characteristic session $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$ expects to interact with a process of type $\mathbb{V}'$ (or a subtype, like $\mathbb{T}'$); however, when a process that behaves like $\mathbb{U}$ is inserted, the cyclic communications and/or the expressions of $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$ (given by Def. 5.5 and 5.7) are disrupted: this is because $\mathbb{U} \nleqslant \mathbb{V}'$, and the (incorrect) message reorderings and mutations allowed by $\nleqslant$ (Figure 6) cause the errors in Figure 3.

We now conclude with our main results.

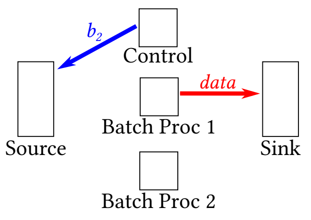
Theorem 5.13. *The asynchronous multiparty session subtyping $\leqslant$ is complete.*



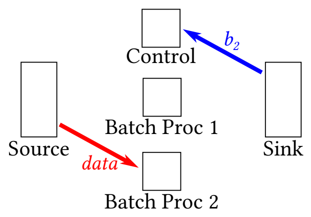
(a) Control tells Source it should send the next batch of data to Processor 1.



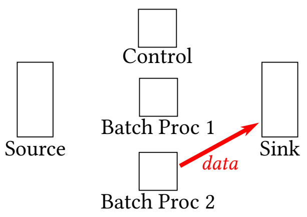
(b) Source starts sending data to Processor 1. Meanwhile, Sink asynchronously tells Control it is ready to receive data from Processor 1.



(c) Processor 1 finishes processing its data and sends the result to Sink; meanwhile, Control tells Source that it should send the next data to Processor 2.



(d) Processor 1 has sent all its data to Sink; Source starts sending data to Processor 2, while Sink asynchronously tells Control it is ready to receive data from Processor 2.



(e) Processor 2 is sending its resulting data to Sink. Processor 2 completes the transmission, we go back to step (a).

Fig. 7. An execution of the basic unoptimised distributed batch processing protocol.

Proof. Direct consequence of Proposition 5.12: by taking r and letting $\mathcal{M} = \mathcal{M}_{r,\mathbb{V}'}$ and $P = \mathcal{P}(\mathbb{U})$ from its statement, we satisfy item *(2)* of Def. 5.1. □

Theorem 5.14. *The asynchronous multiparty session subtyping $\leqslant$ is precise.*

Proof. Direct consequence of Theorems 5.2 and 5.13, which satisfy item *(3)* of Def. 5.1. □

6 EXAMPLE: DISTRIBUTED BATCH PROCESSING

In this section we illustrate our subtyping relation by showing the correctness of a messaging optimisation in a distributed processing scenario. We adapt a multiparty protocol from the *double-buffering algorithm* [Castro-Perez and Yoshida 2020a; Huang et al. 2002; Mostrous et al. 2009; Yoshida et al. 2008], that is widely used, e.g., in streaming media applications and computer graphics, to regulate and speed up asynchronous communication and data processing.

6.1 Basic Unoptimised Protocol

In a distributed processing system, two *Batch Processors* are always ready to receive data from a Source, perform some computation, and send the results to a Sink. A Control process regulates their interactions, by telling the Source/Sink where to send/receive new data; consequently, the Control can influence when the Batch Processors are running (e.g., all at once, or one a time), and can ensure that the Source is not sending too much data (thus overwhelming the Batch Processors and the Sink), and the Sink is expecting data from an active Processor.

Figure 7 provides an overview of the communications between the five parties (Source, Sink, Control, Processor 1 and Processor 2) involved in the system. Their communication protocols can be formalised as multiparty session types. The types $\mathbb{T}_{bp1}$ and T_{bp2} below provide the specification of the two batch processors. They are very simple: they recursively read data (of sort S) from the

Source (src) and send the processing result to the Sink (sk).

$$\mathbb{T}_{\mathsf{bp1}} = \mathbb{T}_{\mathsf{bp2}} = \mu\mathbf{t}.\mathsf{src}?\mathit{data}(\mathsf{S}).\mathsf{sk}!\mathit{data}(\mathsf{S}).\mathbf{t}$$

We now present the types $\mathbb{T}_{\mathsf{src}}$, $\mathbb{T}_{\mathsf{ctl}}$, and $\mathbb{T}_{\mathsf{sk}}$, which provide the specifications of Source, Control and Sink, respectively:

$$\begin{array}{lcl}\mathbb{T}_{\mathsf{src}} & = & \mu\mathbf{t}.\,\&\,\{\mathsf{ctl}?b_1().\mathsf{ctl}!b_1().\mathsf{bp}_1!\mathit{data}(\mathsf{S}).\mathbf{t}\,,\ \mathsf{ctl}?b_2().\mathsf{ctl}!b_2().\mathsf{bp}_2!\mathit{data}(\mathsf{S}).\mathbf{t}\}\\ \mathbb{T}_{\mathsf{sk}} & = & \mu\mathbf{t}.\mathsf{ctl}!b_1().\mathsf{ctl}?b_1().\mathsf{bp}_1?\mathit{data}(\mathsf{S}).\mathsf{ctl}!b_2().\mathsf{ctl}?b_2().\mathsf{bp}_2?\mathit{data}(\mathsf{S}).\mathbf{t}\\ \mathbb{T}_{\mathsf{ctl}} & = & \mu\mathbf{t}.\mathsf{src}!b_1().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathbf{t}\end{array}$$

The above type specifications are summarised as follows:

- the Source (denoted as participant src) expects to be told by the Control (ctl) where to send the next data — either to the Processor 1 or Processor 2 (denoted with message labels b_1 and b_2, respectively). Then, the Source acknowledges (by replying b_1 or b_2 to the Control) sends the data to the corresponding Processor (bp_1 or bp_2), and loops;
- instead, the Sink (denoted as participant sk) notifies the Control that it is willing to read the output from Processor 1 (message b_1), expects an acknowledgement from Control (with the same message b_1), and proceeds with reading the data. Then, it performs a similar sequence of interactions to notify Control and read data from Processor 2, and loops;
- finally, the Control (denoted as ctl) regulates the interactions of Source, Sink, Processor 1 and Processor 2, in a loop where it first tells Source to send to Processor 1, and then waits for Sink to be ready to receive from Processor 2, and then repeats the above for Processor 2.

6.2 Optimised Protocol

The type specifications illustrated in Section 6.1 produce a correct (live) interaction. However, the interaction is also rather sequential:

- after Processor 1 (bp_1) receives its data, Processor 2 (bp_2) is idle, waiting for data that is only made available after Sink sends a message b_1 to Control;
- later on, while bp_2 is processing, bp_1 becomes idle, waiting for data that is only made available after Sink sends a message b_2 to Control.

When implementing the system, it can be appropriate to optimise it by leveraging asynchronous communications: we can aim at a higher degree of parallelism for bp_1 and bp_2 by making their input data available earlier — without compromising the overall correctness of the system.

One way to achieve such an optimisation is to take inspiration from the *double-buffering algorithm* (a version implemented in C is found in [Huang et al. 2002, Section 3.2]), and implement the following alternative type $\mathbb{T}^*_{\mathsf{ctl}}$ for the Control that changes the order of some of the actions of $\mathbb{T}_{\mathsf{ctl}}$:

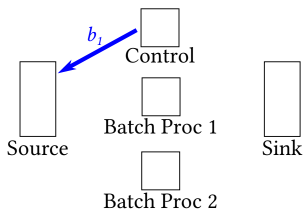
$$\mathbb{T}^*_{\mathsf{ctl}} = \mathsf{src}!b_1().\mathsf{src}!b_2().\mu\mathbf{t}.\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_1().\mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}!b_2().\mathbf{t}$$

According to type $\mathbb{T}^*_{\mathsf{ctl}}$, Control behaves as follows:

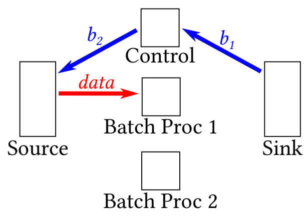
- first, it signals Source that it should send data to bp_1, then to bp_2, and then enters the loop;
- inside the loop, Control receives the notifications from Source and Sink, on their intention to send/receive data to/from bp_1; then, the Control *immediately notifies Source to send again at* bp_1. This refinement allows Source to start sending data earlier, and activate bp_1 immediately.

A resulting execution of the protocol is illustrated in Figure 8.

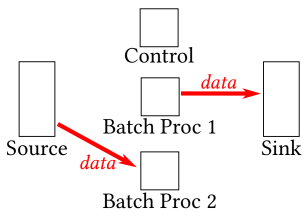
We now want to ensure that $\mathbb{T}^*_{\mathsf{ctl}}$ represents a correct optimisation of $\mathbb{T}_{\mathsf{ctl}}$ — i.e., if a process implementing the former is used in a system where a process implementing the latter is expected, then the optimisation does not introduce deadlocks, orphan messages, or starvation errors. To



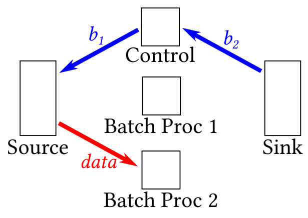
(a) Control tells Source it should send the next batch of data to Processor 1.



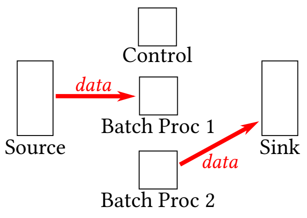
(b) Source starts sending data to Processor 1; asynchronously, Control tells Source it should *also* send data to Processor 2. Meanwhile, Sink asynchronously tells Control it is ready to receive data from Processor 1.



(c) Processor 1 finishes processing its data and sends the result to Sink; meanwhile, Source is sending data to Processor 2.



(d) Processor 1 has sent its data to Sink; Source is still sending to Processor 2; Control asynchronously tells Source to send data to Processor 1; Sink asynchronously tells Control it can now receive from Processor 2.



(e) Source sends data to Processor 1, while Processor 2 is sending its resulting data to Sink. After Processor 2 completes the transmission, we go back to step (b).

Fig. 8. An execution of the optimised distributed batch processing protocol. Note that in steps (c) and (e) the two batch processors are active at the same time, and they can perform their computations in parallel (depending on the time required by the computation, and the time it takes for the source to send its data.)

this purpose, we show that $\mathbb{T}^*_{\mathsf{ctl}}$ is an asynchronous subtype of $\mathbb{T}_{\mathsf{ctl}}$, by Definition 3.7. We take the session trees $\mathsf{T}^*_{\mathsf{ctl}} = \mathcal{T}\left(\mathbb{T}^*_{\mathsf{ctl}}\right)$ and $\mathsf{T}_{\mathsf{ctl}} = \mathcal{T}(\mathbb{T}_{\mathsf{ctl}})$, and we define:

$$\mathsf{T}_R = \mathcal{T}(\mathbb{T}_R) \quad \text{where} \quad \mathbb{T}_R = \mu\mathbf{t}.\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_1().\mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}!b_2().\mathbf{t}$$

Hence, we have $\mathsf{T}^*_{\mathsf{ctl}} = \mathsf{src}!b_1().\mathsf{src}!b_2().\mathsf{T}_R$. We can now prove $\mathbb{T}^*_{\mathsf{ctl}} \leqslant \mathbb{T}_{\mathsf{ctl}}$: since they are both SISO session types, by Definition 3.7 it is enough to build a coinductive derivation showing $\mathsf{T}^*_{\mathsf{ctl}} \lesssim \mathsf{T}_{\mathsf{ctl}}$, with the SISO refinement rules in Definition 3.2. By using the abbreviation $\pi = \mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2()$, and highlighting matching pairs of prefixes, we have the following infinite coinductive derivation: (notice that the topmost refinement matches the second from the bottom)

$$\begin{array}{cl}
\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & \\
\hline\hline
\mathsf{sk}!b_2().\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{sk}!b_2().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-OUT}] \\
\hline\hline
\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-IN}] \\
\hline\hline
\mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-IN}] \\
\hline\hline
\mathsf{src}!b_1().\pi.\mathsf{src}!b_2().\mathsf{T}_R \lesssim \pi.\mathsf{src}!b_1().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-}\mathcal{B}] \\
\hline\hline
\mathsf{sk}!b_1().\mathsf{src}!b_1().\pi.\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{sk}!b_1().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-OUT}] \\
\hline\hline
\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_1().\pi.\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{sk}?b_1().\mathsf{sk}!b_1().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-IN}] \\
\hline\hline
\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_1().\pi.\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-IN}] \\
\hline\hline
\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} & [\text{REF-}\mathcal{B}] \\
\hline\hline
\mathsf{T}^*_{\mathsf{ctl}} = \mathsf{src}!b_1().\mathsf{src}!b_2().\mathsf{T}_R \lesssim \mathsf{src}!b_1().\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1().\mathsf{src}!b_2().\pi.\mathsf{T}_{\mathsf{ctl}} = \mathsf{T}_{\mathsf{ctl}} & [\text{REF-OUT}]
\end{array}$$

Thus, we obtain the following result.

Proposition 6.1. $\mathbb{T}^*_{\mathsf{ctl}} \leqslant \mathbb{T}_{\mathsf{ctl}}$ *holds.*

The types $\mathbb{T}_{\mathsf{bp1}}, \mathbb{T}_{\mathsf{src}}, \mathbb{T}^*_{\mathsf{ctl}}$ and $\mathbb{T}_{\mathsf{sk}}$, can be implemented as processes for Processor, Source, Control and Sink, such as:

$$\begin{array}{rcl} P_{\mathsf{bp1}} & = & P_{\mathsf{bp2}} = \mu X.\mathsf{src}?data(x).\mathsf{sk}!data\langle f(x)\rangle.X \\ P_{\mathsf{src}} & = & \mu X.\sum\left\{\mathsf{ctl}?b_1().\mathsf{ctl}!b_1\langle\rangle.\mathsf{bp}_1!data(datum).\mathsf{X}\ ,\ \mathsf{ctl}?b_2().\mathsf{ctl}!b_2\langle\rangle.\mathsf{bp}_2!data\langle datum\rangle.\mathsf{X}\right\} \\ P_{\mathsf{ctl}} & = & \mathsf{src}!b_1\langle\rangle.\mathsf{src}!b_2\langle\rangle.\mu\mathsf{X}.\mathsf{src}?b_1().\mathsf{sk}?b_1().\mathsf{sk}!b_1\langle\rangle.\mathsf{src}!b_1\langle\rangle.\mathsf{src}?b_2().\mathsf{sk}?b_2().\mathsf{sk}!b_2\langle\rangle.\mathsf{src}!b_2\langle\rangle.\mathsf{X} \\ P_{\mathsf{sk}} & = & \mu X.\mathsf{ctl}!b_1\langle\rangle.\mathsf{ctl}?b_1().\mathsf{bp}_1?data(y_1).\mathsf{ctl}!b_2\langle\rangle.\mathsf{ctl}?b_2().\mathsf{bp}_2?data(y_2).X \end{array}$$

where f (used by P_{bp1} and P_{bp2}) represents processing of the data received by src, and where the variables x, y_1 and y_2, and objects $f(x)$ and *datum* are all of sort S.

Proposition 6.2 (Correctness of Optimisation). *Let:*

- $\Gamma = \mathsf{bp}_1 : (\epsilon, \mathbb{T}_{\mathsf{bp1}}), \mathsf{bp}_2 : (\epsilon, \mathbb{T}_{\mathsf{bp2}}), \mathsf{src} : (\epsilon, \mathbb{T}_{\mathsf{src}}), \mathsf{ctl} : (\epsilon, \mathbb{T}_{\mathsf{ctl}}), \mathsf{sk} : (\epsilon, \mathbb{T}_{\mathsf{sk}})$ *and*
- $\mathcal{M} = \mathsf{bp}_1 \triangleleft \mathsf{P}_{\mathsf{bp1}} \mid \mathsf{bp}_1 \triangleleft \varnothing \mid \mathsf{bp}_2 \triangleleft \mathsf{P}_{\mathsf{bp2}} \mid \mathsf{bp}_2 \triangleleft \varnothing \mid \mathsf{src} \triangleleft \mathsf{P}_{\mathsf{src}} \mid \mathsf{src} \triangleleft \varnothing \mid \mathsf{ctl} \triangleleft \mathsf{P}_{\mathsf{ctl}} \mid \mathsf{ctl} \triangleleft \varnothing \mid \mathsf{sk} \triangleleft \mathsf{P}_{\mathsf{sk}} \mid \mathsf{sk} \triangleleft \varnothing$

Then, we have that Γ *is live and* $\Gamma \vdash \mathcal{M}$*. Also,* $\mathcal{M}$ *will never reduce to error.*

Observe that Γ uses the basic unoptimised protocol $\mathbb{T}_{\mathsf{ctl}}$ for the Control, while $\mathcal{M}$ implements the optimised protocol $\mathbb{T}^*_{\mathsf{ctl}}$: the typing holds by Proposition 6.1, while the absence of errors is consequence of Corollary 4.9.

7 ADDITIONAL PRECISENESS RESULTS

We now use our main result (Theorem 5.14) to prove two additional results: our subtyping $\leqslant$ is *denotationally precise* (Section 7.1) and also *precise wrt. liveness* (Section 7.2).

7.1 Denotational Preciseness of Subtyping

The approach to preciseness of subtyping described in the previous sections is nowadays dubbed *operational preciseness.* In prior work, the canonical approach to preciseness of subtyping for a given calculus has been *denotational*: a type $\mathbb{T}$ is interpreted (with notation $[\![\mathbb{T}]\!]$) as the set that describes the meaning of $\mathbb{T}$, according to denotations of the expressions (terms, processes, *etc.*) of the calculus. E.g., λ-calculus types are usually interpreted as subsets of the domains of λ-models [Barendregt et al. 1983; Hindley 1983]. With this approach, subtyping is interpreted as set-theoretic inclusion.

Under this denotational viewpoint, a subtyping $\trianglelefteq$ is sound when $\mathbb{T} \trianglelefteq \mathbb{T}'$ implies $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$, and complete when $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$ implies $\mathbb{T} \trianglelefteq \mathbb{T}'$. Hence, a subtyping $\trianglelefteq$ is precise when:

$$\mathbb{T} \trianglelefteq \mathbb{T}' \quad \text{if and only if} \quad [\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$$

This notion is nowadays dubbed *denotational preciseness.*

We now adapt the denotational approach to our setting. Let us interpret a session type $\mathbb{T}$ as the set of closed processes typed by $\mathbb{T}$, i.e. $[\![\mathbb{T}]\!] = \{P \mid\ \vdash P : \mathbb{T}\}$.

We now show that our subtyping is denotationally precise. The denotational soundness follows from the subsumption rule [T-SUB] (Figure 5) and the interpretation of subtyping as set-theoretic inclusion. In order to prove that denotational completeness holds, we show a more general result (along the lines of Ghilezan et al. [2019]): by leveraging characteristic processes and sessions, we show that operational completeness implies denotational completeness.

Theorem 7.1. *The multiparty asynchronous subtyping* $\leqslant$ *is denotationally complete.*

Proof. We proceed by contradiction. Suppose that denotational completeness does *not* hold, i.e., there are $\mathbb{T}$ and $\mathbb{T}'$ such that $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$ but $\mathbb{T} \nleqslant \mathbb{T}'$.

From $\mathbb{T} \not\leqslant \mathbb{T}'$, by Step 1 (Section 5.1, eq. (5)), there are $\mathbb{U}$ and $\mathbb{V}'$ satisfying $\mathbb{U} \not\leqslant \mathbb{V}'$ and $\mathcal{T}(\mathbb{U}) \in [\![\mathcal{T}(\mathbb{T})]\!]_{\mathrm{SO}}$ and $\mathcal{T}(\mathbb{V}') \in [\![\mathcal{T}(\mathbb{T}')]\!]_{\mathrm{SI}}$. Then by Step 4 (Proposition 5.12(3)) we have:

$$\mathsf{r} \triangleleft \mathcal{P}(\mathbb{U}) \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'} \longrightarrow^* \mathsf{error} \tag{7}$$

where $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$, (with $\mathsf{r} \notin \mathsf{pt}(\mathbb{V}')$) is the characteristic session of Step 3 (Definition 5.9) and $\mathcal{P}(\mathbb{U})$ is the characteristic process of $\mathbb{U}$ of Step 2 (Definition 5.5).

By Step 2 (Proposition 5.6), the characteristic process of $\mathbb{U}$ is typable by $\mathbb{T}$, i.e., $\vdash \mathcal{P}(\mathbb{U}) : \mathbb{T}$; therefore, from $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$ we also have $\vdash \mathcal{P}(\mathbb{U}) : \mathbb{T}'$. Hence, by Step 3 (Definition 5.9, Proposition 5.11) for $\mathbb{V}'$ the characteristic session $\mathcal{M}_{\mathsf{r},\mathbb{V}'}$ is typable if composed with $\mathcal{P}(\mathbb{U}) : \mathbb{T}'$, i.e.:

$$\Gamma \vdash \mathsf{r} \triangleleft \mathcal{P}(\mathbb{U}) \mid \mathsf{r} \triangleleft \varnothing \mid \mathcal{M}_{\mathsf{r},\mathbb{V}'} \quad \text{for some live } \Gamma \tag{8}$$

But then, (8) and (7) contradict the soundness of the type system (Theorem 4.8). Therefore, we conclude that for all $\mathbb{T}$ and $\mathbb{T}'$, $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{T}']\!]$ implies $\mathbb{T} \leqslant \mathbb{T}'$, which is the thesis. □

Theorem 7.2 (Denotational preciseness). *The subtyping relation is denotationally precise.*

7.2 Preciseness of Subtyping wrt. Liveness

Our results also provide a stepping stone to show that that *our multiparty asynchronous subtyping is precise wrt. liveness* (Definition 4.4), as formalised in Theorem 7.3 below.

Notably, this result can be lifted to the realm of Communicating Finite-State Machines (CFSMs) [Brand and Zafiropulo 1983], by the following interpretation. *Communicating session automata (CSA)* (introduced by Deniélou and Yoshida [2012]) are a class of CFSMs with a finite control based on internal/external choices, corresponding to a session type; a typing environment Γ corresponds to a system of CSA, and reduces with the same semantics (Definition 4.3). Therefore, our notion of typing environment liveness (Definition 4.4) guarantees deadlock-freedom, orphan message-freedom, and starvation-freedom of the corresponding system of CSA — similarly to the notion of ∞-*multiparty compatibility* between CSA, by Lange and Yoshida [2019]. Correspondingly, by Theorem 7.3, our subtyping $\leqslant$ is a sound and complete preorder allowing to refine any live system of CSA (by replacing one or more automata), while preserving the overall liveness of the system.

To state the result, we adopt the notation $\Gamma\{\mathsf{p} \mapsto (\sigma, \mathbb{T})\}$ to represent the typing environment obtained from Γ by replacing the entry for p with $(\sigma, \mathbb{T})$.

Theorem 7.3. *For all session types $\mathbb{T}$ and $\mathbb{T}'$, multiparty asynchronous subtyping $\leqslant$ is:*

sound wrt. liveness: *if* $\mathbb{T} \leqslant \mathbb{T}'$, *then* $\forall\Gamma$ *with* $\Gamma(\mathsf{r}) = (\epsilon, \mathbb{T}')$, *when* Γ *is live*, $\Gamma\{\mathsf{r} \mapsto (\epsilon, \mathbb{T})\}$ *is live;*
complete wrt. liveness: *if* $\mathbb{T} \not\leqslant \mathbb{T}'$, *then* $\exists\Gamma$ *live with* $\Gamma(\mathsf{r}) = (\epsilon, \mathbb{T}')$, *but* $\Gamma\{\mathsf{r} \mapsto (\epsilon, \mathbb{T})\}$ *is not live.*

Proof. Soundness of $\leqslant$ follows by Lemma 4.7. Completeness, instead, descends from Proposition 5.12: for any $\mathbb{T}, \mathbb{T}'$ such that $\mathbb{T} \not\leqslant \mathbb{T}'$, we can build a live typing context Γ (see (6)) with $\Gamma(\mathsf{r}) = (\epsilon, \mathbb{T}')$ and cyclic communications (item 1). Observe that the environment $\Gamma\{\mathsf{r} \mapsto (\epsilon, \mathbb{T})\}$ types the session of Proposition 5.12(3), that reduces to error: hence, by the contrapositive of Corollary 4.9, we conclude that $\Gamma\{\mathsf{r} \mapsto (\epsilon, \mathbb{T})\}$ is not live. □

8 RELATED WORK, FUTURE WORK, AND CONCLUSION

Precise Subtyping for λ-Calculus and Semantic Subtyping. The notion of preciseness has been adopted as a criterion to justify the canonicity of subtyping relations, in the context of both functional and concurrent calculi. Operational preciseness of subtyping was first introduced by Blackburn et al. [2012] (and later published in [Ligatti et al. 2017]), and applied to λ-calculus with iso-recursive types. Later, Dezani-Ciancaglini and Ghilezan [2014] adapted the idea of Blackburn et al. [2012] to the setting of the concurrent λ-calculus with intersection and union types by Dezani-Ciancaglini et al. [1998], and proved both operational and denotational preciseness. In the context of the λ-calculus, a similar framework, *semantic subtyping*, was proposed by Castagna and Frisch [2005]: each type T is interpreted as the set of values having type T, and subtyping is defined as subset inclusion between type interpretations. This gives a precise subtyping as long as the calculus allows to operationally distinguish values of different types. Semantic subtyping was also studied by Castagna et al. [2008] (for a π-calculus with a patterned input and IO-types), and by Castagna et al. [2009a] (for a π-calculus with binary session types); in both works, types include union, intersection and negation. Semantic subtyping is precise for the calculi of [Castagna et al. 2008, 2009a; Frisch et al. 2008]: this is due to the type case constructor in [Frisch et al. 2008], and to the blocking of inputs for values of "wrong" types in [Castagna et al. 2008, 2009a].

Precise Subtyping for Session Types. In the context of *binary* session types, the first general formulation of precise subtyping (synchronous and asynchronous) is given by Chen et al. [2017], for a π-calculus typed by assigning session types to channels (as in the work by Honda et al. [1998]). The first result by Chen et al. [2017] is that the well-known branching-selection subtyping [Demangeon and Honda 2011; Gay and Hole 2005] is sound and complete for the *synchronous* binary session π- calculus. Chen et al. [2017] also examine an *asynchronous* binary session π-calculus, and introduce a subtyping relation (restricting the subtyping for the higher-order π-calculus by Mostrous and Yoshida [2015]) that is also proved precise. Our results imply that the binary session subtyping of Chen et al. [2017] (without delegation) is a subset of our multiparty subtyping, hence the subtyping of Chen et al. [2017] is expressible via tree decomposition. We compare our subtyping and theirs (specifically, the use n-hole contexts) in Remark 3.6; further, we prove Example 3.10 via tree decomposition — and such a proof would be more difficult with the rules of Chen et al. [2017].

In the context of *multiparty* session types, Ghilezan et al. [2019] adopt an approach similar to Chen et al. [2017] to prove that the *synchronous* multiparty extension of binary session subtyping [Gay and Hole 2005] is sound and complete, hence precise. Our subtyping becomes equivalent to Ghilezan et al. [2019] by removing rules [REF-$\mathcal{A}$] and [REF-$\mathcal{B}$] from Definition 3.2; however, such a formulation would not be much simpler than the one by Ghilezan et al. [2019], nor would simplify its algorithm: lacking asynchrony, their subtyping is already quite simple, and decidable.

An *asynchronous* subtyping relation for multiparty session types was proposed by Mostrous et al. [2009]; notably, the paper claims that the relation is decidable, but this was later disproved by Bravetti et al. [2018] (see next paragraph). The main crucial difference between our subtyping and the one by Mostrous et al. [2009] is that the latter is *unsound* in our setting: it does not guarantee orphan message freedom, hence it accepts processes that reduce to error by rule [ERR-OPHN] in Figure 3 (see the counterexamples in [Chen et al. 2017, page 46], also applicable in the multiparty setting). A recent work by Horne [2020] introduces a novel formalisation of subtyping for *synchronous* multiparty session types equipped with parallel composition. Horne's work does not address the problem of precise subtyping, and its extension to asynchronous communication seems non-trivial: in fact, the subtyping appears to be unsound in our setting, because (as in Mostrous et al. [2009]) it focuses on deadlock-freedom, and does not address, e.g., orphan message freedom. Studying such extensions and issues in the setting of Horne [2020] can lead to intriguing future work.

Session Types and Communicating Automata. Asynchronous session subtyping was shown to be undecidable, even for binary sessions, by Lange and Yoshida [2017] and Bravetti et al. [2018], by leveraging a correspondence between session types and communicating automata theories — a correspondence first established by Deniélou and Yoshida [2012] with the notion of *session automata* based on Communicating Finite-State Machines [Brand and Zafiropulo 1983]. This undecidability result prompted the research on various limited classes of (binary) session types for which asynchronous subtyping is decidable [Bravetti et al. 2019a, 2018; Lange and Yoshida 2017]: for example, if we have binary session types without choices (which, in our setting, corresponds to single-input, single-output types interacting with a single other participant), then asynchronous subtyping becomes decidable. The aim of our paper is *not* finding a decidable approximation of asynchronous *multiparty* session subtyping, but finding a canonical, *precise* subtyping. Interestingly, our SISO decomposition technique leads to: (1) intuitive but general refinement rules (see Example 3.10, where $\leqslant$ proves an example not supported by the subtyping algorithm in [Bravetti et al. 2019a]); and (2) preciseness of $\leqslant$ wrt. liveness (Theorem 7.3) which is directly usable to define the precise multiparty asynchronous refinement relation wrt. liveness in communicating session automata [Deniélou and Yoshida 2012; Deniélou and Yoshida 2013].

Future Work. We plan to investigate precise subtyping for richer multiparty session π-calculi, e.g. with multiple session initiations and delegation, with a formalisation similar to Scalas and Yoshida [2019]. We expect that, even in this richer setting, our subtyping relation will remain the same, and its preciseness will be proved in the same way — except that session types will support session types as payloads, and Figure 6 and Def. 5.5 will have more cases. This can be observed in the work by Chen et al. [2017]: they support multiple sessions, process spawning, delegation — and yet, such features don't impact the subtyping relation (except for having session types as payloads).

We also plan to study the precise subtyping for typing environment properties other than our liveness (Def. 4.4). For example, we may consider a weaker deadlock freedom property, which can be sufficient for correct session typing (as shown by Scalas and Yoshida [2019]) as long as the session calculus does *not* include error reductions in case of starvation or orphan messages (rules [ERR-STRV] and [ERR-OPHN] in Figure 3). We expect that, if we lift the liveness requirement, the resulting precise subtyping relation will be somewhat surprising and counter-intuitive, and harder to formulate. For example: due to the lack of orphan message errors, subtyping should allow processes forget their inputs in some situations, hence we should have, e.g., $\mathtt{end} \leqslant \mathtt{p}?\ell(\mathtt{nat}).\mathtt{end}$.

Another compelling topic for future work is finding non-trivial decidable approximations of our multiparty asynchronous subtyping relation. As remarked above, the relation is inherently undecidable — hence type-checking is also undecidable, because rule [T-SUB] (Figure 5) is undecidable. Our relation is trivially decidable if restricted to non-recursive types. The known decidable fragments of binary asynchronous subtyping [Bravetti et al. 2019a, 2018; Lange and Yoshida 2017] are sound wrt. our relation — but they are also limited to two-party sessions, and they may become undecidable if naively generalised to multiparty sessions. If a decidable approximation of asynchronous subtyping is adopted for rule [T-SUB] (Figure 5), then type-chcecking becomes decidable.

Conclusion. Unlike this paper, no other published work addresses precise *asynchronous multiparty* session subtyping. A main challenge was the exact formalisation of the subtyping itself, which must satisfy many *desiderata*: it must capture a wide variety of input/output reorderings performed by different participants, without being too strict (otherwise, completeness is lost) nor too lax (otherwise, soundness is lost); moreover, its definition must not be overly complex to understand, and tractable in proofs. We achieved these *desiderata* with our novel approach, based on SISO tree decomposition and refinement, which yields a simpler subtyping definition than [Chen et al. 2017] (see Remark 3.6). Moreover, our results are much more general than [Ghilezan et al. 2019]: by using live typing environments (Def. 4.4), we are not limited to sessions that match some global type; our results are also stronger, as we prove soundness wrt. a wider range of errors (see Figure 3).

ACKNOWLEDGMENTS

We thank the POPL reviewers for their insightful comments and suggestions, Mariangiola Dezani-Ciancaglini for her initial collaboration, and Simon Castellan for the fruitful discussion. This work was supported by: EU Horizon 2020 project 830929 ("CyberSec4Europe"); EU COST Actions CA15123 ("EUTypes") and IC1201 ("BETTY"); EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS; MPNTR and SFRS #6526707 ("AI4TrustBC").

REFERENCES

Davide Ancona, Viviana Bono, Mario Bravetti, Giuseppe Castagna, Joana Campos, Simon J. Gay, Elena Giachino, Einar Broch Johnsen, Viviana Mascardi, Nicholas Ng, Luca Padovani, Pierre-Malo Deniélou, Nils Gesbert, Raymond Hu, Francisco Martins, Fabrizio Montesi, Rumyana Neykova, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3, 95–230. https://doi.org/10.1561/2500000031

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic* 48, 4 (1983), 931–940. https://doi.org/10.2307/2273659

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008 (LNCS, Vol. 5201)*, Franck van Breugel and Marsha Chechik (Eds.). Springer, 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

Jeremy Blackburn, Ivory Hernandez, Jay Ligatti, and Michael Nachtigal. 2012. *Completely Subtyping Iso-recursive Types.* Technical Report CSE-071012. University of South Florida.

Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *JACM* 30, 2 (1983). https://doi.org/10.1145/322374.322380

Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2019a. A Sound Algorithm for Asynchronous Session Subtyping. 140 (2019), 38:1–38:16. https://doi.org/10.4230/LIPIcs.CONCUR.2019.38

Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2019b. A Sound Algorithm for Asynchronous Session Subtyping (extended version). Extended version of [Bravetti et al. 2019a], available at: http://arxiv.org/abs/1907.00421.

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

Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. 2008. Semantic subtyping for the pi-calculus. *Theoretical Computer Science* 398, 1-3 (2008), 217–242. https://doi.org/10.1016/j.tcs.2008.01.049

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009a. Foundations of session types. In *PPDP 2009*. ACM, 219–230. https://doi.org/10.1145/1599410.1599437

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *ICALP 2005 (LNCS, Vol. 3580)*. Springer, 30–34. https://doi.org/10.1007/11523468_3

Giuseppe Castagna, Nils Gesbert, and Luca Padovani. 2009b. A Theory of Contracts for Web Services. *ACM Trans. Program. Lang. Syst.* 31, 5, Article 19 (July 2009). https://doi.org/10.1145/1538917.1538920

David Castro-Perez and Nobuko Yoshida. 2020a. CAMP: Cost-Aware Multiparty Session Protocols. *PACMPL* SPLASH/OOPSLA (2020), 30 pages. To appear.

David Castro-Perez and Nobuko Yoshida. 2020b. Compiling First-Order Functions to Session-Typed Parallel Code. In *29th International Conference on Compiler Construction*. ACM, 143–154. https://doi.org/10.1145/3377555.3377889

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* 13, 2 (2017). https://doi.org/10.23638/LMCS-13(2:12)2017

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. On the Preciseness of Subtyping in Session Types. In *PPDP'14*. ACM, 135–146. https://doi.org/10.1145/2643135.2643138

Romain Demangeon and Kohei Honda. 2011. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *CONCUR 2011 (LNCS, Vol. 6901)*. Springer, 280–296. https://doi.org/10.1007/978-3-642-23217-6_19

Romain Demangeon and Nobuko Yoshida. 2015. On the Expressiveness of Multiparty Sessions. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India (LIPIcs, Vol. 45)*, Prahladh Harsha and G. Ramalingam (Eds.). https://doi.org/10.4230/LIPIcs.FSTTCS.2015.560

Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *ESOP 2012*. 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP 2013 (LNCS, Vol. 7966)*. Springer, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Adolfo Piperno. 1998. A Filter Model for Concurrent lambda-Calculus. *SIAM J. Comput.* 27, 5 (1998), 1376–1419. https://doi.org/10.1137/S0097539794275860

Mariangiola Dezani-Ciancaglini and Silvia Ghilezan. 2014. Preciseness of Subtyping on Intersection and Union Types. In *RTA-TLCA 2-14 (LNCS, Vol. 8560)*. Springer, 194–207. https://doi.org/10.1007/978-3-319-08918-8_14

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of ACM* 55, 4 (2008), 19:1–19:64. https://doi.org/10.1145/1391289.1391293

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

Simon Gay and Antonio Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools.* River Publishers. https://doi.org/10.13052/rp-9788793519817

Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. 2019. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.* 104 (2019), 127–173. https://doi.org/10.1016/j.jlamp.2018.12.002

Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. 2020. Precise Subtyping for Asynchronous Multiparty Sessions (Extended Version). Available at: https://arxiv.org/abs/2010.13925.

J. Roger Hindley. 1983. The Completeness Theorem for Typing Lambda-Terms. *Theoretical Computer Science* 22 (1983), 1–17. https://doi.org/10.1016/0304-3975(83)90136-6

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. In *ESOP'98 (LNCS, Vol. 1381)*. Springer, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL 2008*. ACM, 273–284. https://doi.org/10.1145/1328438.1328472

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

Ross Horne. 2020. Session Subtyping and Multiparty Compatibility Using Circular Sequents. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:22. https://doi.org/10.4230/LIPIcs.CONCUR.2020.12

Raymond Hu. 2017. Distributed Programming Using Java APIs Generated from Session Types. (2017), 287–308. https://doi.org/10.13052/rp-9788793519817

Hai Huang, Padmanabhan Pillai, and Kang G. Shin. 2002. Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems. In *USENIX Annual Technical Conference, General Track*. https://www.usenix.org/legacy/event/usenix02/full_papers/huang/huang.pdf

Julien Lange and Nobuko Yoshida. 2017. On the Undecidability of Asynchronous Session Subtyping. In *FoSSaCS 2017 (LNCS, Vol. 10203)*. 441–457. https://doi.org/10.1007/978-3-662-54458-7_26

Julien Lange and Nobuko Yoshida. 2019. Verifying Asynchronous Interactions via Communicating Session Automata. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. 97–117. https://doi.org/10.1007/978-3-030-25540-4_6

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 4 (March 2017), 36 pages. https://doi.org/10.1145/2994596

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

Dimitris Mostrous and Nobuko Yoshida. 2015. Session Typing and Asynchronous Subtying for Higher-Order π-Calculus. *Information and Computation* 241 (2015), 227–263. https://doi.org/10.1016/j.ic.2015.02.002

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (LNCS, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 316–332. https://doi.org/10.1007/978-3-642-00590-9_23

Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. 2015. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC 2015 (LNCS, Vol. 9031)*. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS 2012 (LNCS, Vol. 7304)*. Springer, 202–218. https://doi.org/10.1007/978-3-642-30561-0_15

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Benjamin C. Pierce and Davide Sangiorgi. 1996. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science* 6, 5 (1996), 409–453.

Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction.* Cambridge University Press. https://doi.org/10.1017/CBO9780511777110

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *PACMPL* 3, POPL (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE '94 (LNCS, Vol. 817)*. Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. 2008. Session-Based Compilation Framework for Multicore Programming. In *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures (LNCS, Vol. 5751)*, Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine (Eds.). Springer, 226–246. https://doi.org/10.1007/978-3-642-04167-9_12