




A Sound and Complete Characterization of Fair Asynchronous Session Subtyping

Mario Bravetti  

University of Bologna, Italy

Luca Padovani  

University of Bologna, Italy

Gianluigi Zavattaro  

University of Bologna, Italy

Abstract

Session types are abstractions of communication protocols enabling the static analysis of message-passing processes. Refinement notions for session types are key to support safe forms of process substitution while preserving their compatibility with the rest of the system. Recently, a fair refinement relation for asynchronous session types has been defined allowing the anticipation of message outputs with respect to an unbounded number of message inputs. This refinement is useful to capture common patterns in communication protocols that take advantage of asynchrony. However, while the semantic (*à la testing*) definition of such refinement is straightforward, its characterization has proved to be quite challenging. In fact, only a sound but not complete characterization is known so far. In this paper we close this open problem by presenting a sound and complete characterization of asynchronous fair refinement for session types. We relate this characterization to those given in the literature for *synchronous* session types by leveraging a novel labelled transition system of session types that embeds their asynchronous semantics.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Binary sessions, session types, fair asynchronous subtyping

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.11

Related Version *Full Version*: <https://arxiv.org/abs/2506.06078> [7]

Funding The authors have been partially supported by the ANR project SmartCloud ANR-23-CE25-0012 and the PNRR project CN-HPC Spoke 9 Innovation Grant RTMER.

Acknowledgements We are grateful to the anonymous CONCUR'25 reviewers for their feedback.

1 Introduction

Abstract models such as communicating finite-state machines [3] and asynchronous session types [18] are essential to reason about correctness of distributed systems whose components communicate with point-to-point FIFO channels. A fundamental issue, which makes it possible to manage system correctness in a compositional way, is the development of techniques allowing a component to be *refined* independently of the others, without compromising the correctness of the whole system. In this respect, the notion of fair asynchronous refinement for session types introduced by Bravetti, Lange and Zavattaro [6] guarantees fair termination, *i.e.* successful termination under fairness assumption: execution traces that loop forever are deemed unrealistic/unimportant and are not considered. Such a notion implies all the desirable safety and liveness properties of communicating systems, including communication safety, deadlock freedom, absence of orphan messages, and lock freedom. However, while the semantic (*à la testing*) definition of such refinement is straightforward, its (coinductive) characterization (*à la session subtyping*) has proved to be quite challenging. In fact, Bravetti, Lange and Zavattaro [6] only provide a sound but not complete characterization.



© Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro;
licensed under Creative Commons License CC-BY 4.0

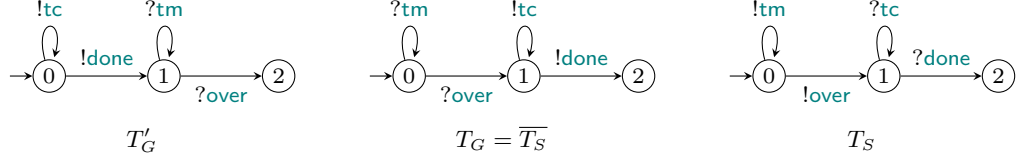
36th International Conference on Concurrency Theory (CONCUR 2025).

Editors: Patricia Bouyer and Jaco van de Pol; Article No. 11; pp. 11:1–11:17

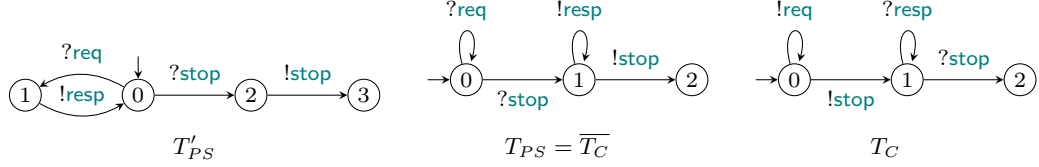
Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Satellite protocols. T'_G is the refined session type of the ground station, T_G is the session type of ground station, and T_S is the session type of the spacecraft.



■ **Figure 2** Stream processing server T'_{PS} is the refined session type of the batch processing server T_{PS} , and T_C is the session type of the client.

With respect to previous notions of asynchronous session subtyping [22, 9] (which leverage asynchrony by allowing the refined component to anticipate message emissions, but only under certain conditions) fair asynchronous subtyping [6] makes it possible to encompass subtypes that occur naturally in communication protocols, *e.g.* where two parties simultaneously send each other a finite but unspecified amount of messages before consuming them from their buffers. We illustrate this scenario in Figure 1, which depicts the interaction between a spacecraft S and a ground station G that communicate via two unbounded asynchronous channels (one in each direction). For convenience, the protocols are represented as communicating finite-state machines [3], *i.e.* with a labeled transition system-like notation where “?” represents inputs, “!” represents outputs, and the initial state is indicated by an incoming arrow. Consider T_S and T_G first. Session type T_S is the abstraction of the spacecraft, which may send a finite but unspecified number of telemetries **tm** (abstractly modeling a **for** loop), followed by a message **over**. In the second phase, the spacecraft receives a number of telecommands (**tc**), followed by a message **done**. In principle, the ground station should behave as the type T_G which is the dual of T_S (denoted by $\overline{T_S}$) where outputs become inputs and vice-versa. However, since the flyby window may be short, it makes sense to implement the ground station so that it communicates with the satellite by anticipating the output of the commands. That is, the ground station follows the type T'_G , which sends telecommands before receiving telemetries. In this way T'_G and T_S interact in a symmetric manner: they first send all of their messages and then consume the messages sent from the other partner. No communication error can occur, and the communication protocol can always terminate successfully, with empty queues. In fact T'_G and T_S also form a correct composition in the sense that every message that is sent by one participant is eventually received by the other. The session subtyping presented by Bravetti, Lange and Zavattaro [6], which is proved to be a sound characterization of fair asynchronous session refinement, makes it possible to actually *prove* that T'_G refines T_G .

Let us now consider, in Figure 2, a more complex scenario concerning a processing server PS and its client C . Consider T_{PS} and T_C first. Session type T_{PS} is the abstraction of a batch processing server and T_C that of a client. Note that these types are respectively isomorphic to T_G and T_S of Figure 1: now the client T_C sends generic requests **req** (instead of specific telemetry data **tm**) and, when it decides to stop sending via **stop**, it keeps waiting

for responses `resp` (instead of telecommand data `tc`) until it receives `stop`. As in the previous scenario, the client T_C and the batch processing server T_{PS} (which is its dual) obviously form a correct composition. Also here it is possible to consider a more efficient version of the processing server, *i.e.* a stream processing server T'_{PS} which immediately (and asynchronously) sends the response to each request it receives. In this scenario, it may be natural to send responses one at a time after each request. In fact also T'_{PS} and T_C form a correct composition. Actually, we have that T'_{PS} is a fair asynchronous session refinement of T_{PS} , meaning that T'_{PS} is a good replacement of T_{PS} in any possible context. However, the characterization provided by Bravetti, Lange and Zavattaro [6] is unable to capture this specific refinement.

Technically speaking, the coinductive characterization of subtyping given by Bravetti, Lange and Zavattaro [6] is not complete because it does not support *output covariance*, that is the possibility for a subtype to remove branches in output choices, corresponding to a reduced internal nondeterminism. This feature is needed to relate T'_{PS} and T_{PS} of Figure 2 because the state 1 (resp. 2) of T'_{PS} , in which the forced internal choice of emitting `!resp` (resp. `!stop`) is taken, has a unique corresponding state in T_{PS} , namely state 1, where two possible choices are present (either `!resp` or `!stop` can be sent).

In this paper we close the open problem of finding a sound and complete characterization of asynchronous fair refinement for session types. The characterization we present is crucially based on a novel asynchronous semantics for session types that: (i) expresses their asynchronous behavior without explicit modeling of FIFO buffers (as in the original paper of Bravetti, Lange and Zavattaro [6]), and (ii) allows us to adapt, to the asynchronous case, the approach introduced by Ciccone and Padovani [23, 12] for providing complete characterizations of *fair session subtyping* in the *synchronous* case. Being complete, our new subtyping allows us to prove that the stream processing server T'_{PS} is a refinement of the batch processing server T_{PS} .

Structure of the paper. In Section 2 we recall the notion of fair asynchronous refinement and the buffer-based asynchronous semantics of session types given by Bravetti, Lange and Zavattaro [6]. In Section 3 we present our novel alternative asynchronous semantics and prove that it is equivalent to the previous one, in the sense that it characterizes the same notion of session composition correctness. In Section 4 we present a sound and complete characterization of asynchronous fair refinement based on the semantics of Section 3. We conclude in Section 5 with a more detailed description of related work and hints at further developments. Proofs and additional technical material can be found in the long version of the paper [7].

2 Preliminaries

We start by recalling the syntax and asynchronous semantics of session types as well as the notion of fair refinement given by Bravetti, Lange and Zavattaro [6]. In the definition of the syntax, instead of using the \oplus , $\&$, and the *rec* operators, we consider an equivalent process algebraic notation with choice and prefix [20] and a coinductive syntax to deal with possibly infinite session types. Following Bravetti, Lange and Zavattaro [6] we restrict to first-order session types, *i.e.* session types in which send/receive operations are used to exchange only messages of elementary *singleton type*. Considering higher-order types, where send/receive operations can be used to exchange also sessions, would not modify significantly the proofs of our results.

We assume the existence of a set of *singleton types* ranged over by $\mathbf{a}, \mathbf{b}, \dots$ that we use to label branches in session types. The only value of type \mathbf{a} is called *tag* and is denoted by the same symbol \mathbf{a} . *Pre-session types* are the possibly infinite trees coinductively generated by the productions below:

$$\begin{array}{ll} \text{Polarity} & p ::= ? \mid ! \\ \text{Pre-session type} & S, T ::= \text{end} \mid \sum_{i \in I} p\mathbf{a}_i.S_i \end{array}$$

The *polarities* $?$ and $!$ are used to distinguish *input actions* from *output actions*. A *session type* is a pre-session type that satisfies the following well-formedness conditions:

1. in every subtree of the form $\sum_{i \in I} p\mathbf{a}_i.S_i$, I is a non-empty finite set and if I contains more than one element, then the message types \mathbf{a}_i are pairwise distinct tags;
2. the tree is *regular*, namely it is made of finitely many distinct subtrees $\sum_{i \in I} p\mathbf{a}_i.S_i$. Courcelle [13] established that every such tree can be expressed as the unique solution of a finite system of equations $\{S_i = T_i\}_{i \in I}$ where each metavariable S_i may occur (guarded) in the T_j 's;
3. every subtree $\sum_{i \in I} p\mathbf{a}_i.S_i$ contains a leaf of the form end .

A session type end describes a terminated session (*i.e.* the corresponding channel is no longer usable). A branching session type $\sum_{i \in I} p\mathbf{a}_i.S_i$ describes a channel used for sending or receiving a message of type \mathbf{a}_i and then according to S_i . The well-formedness condition (1) above requires that message types must be pairwise distinct tags if there is more than one branch. Sometimes we write $p\mathbf{a}_1.S_1 + \dots + p\mathbf{a}_n.S_n$ for $\sum_{i=1}^n p\mathbf{a}_i.S_i$. Note that the polarity p is the same in every branch, *i.e.* mixed choices are disallowed as usual.

The well-formedness condition (2) guarantees that we deal only with session types representing trees consisting of finitely many distinct subtrees as those that can be represented with a finite syntax and the *rec* operator [13].

The well-formedness condition (3) ensures that session types describe protocols that can always terminate. This assumption is not usually present in other session type syntax definitions where it is possible to also express non-terminating protocols. As we will see in the following (Definition 3), non-terminating protocols are not inhabited in our theory. In particular, we consider a notion of correct type composition such that, whatever sequence of interactions is performed, such sequence can always be extended to reach successful session termination where both types become end . In conclusion, ruling out non-terminating protocols in the syntax of session types does not affect the family of protocols we are interested in modeling and at the same time simplifies the technical development.

It is worth to mention that Bravetti, Lange and Zavattaro [6] do not consider the well formedness condition (3), hence allowing for the specification of non-terminating types. In the comparison with the related literature in Section 5 we discuss the impact of the absence of this condition on the definition of the fair asynchronous subtyping reported in that paper.

We write \bar{S} for the *dual* of S , which is the session type corecursively defined by the equations

$$\overline{\text{end}} = \text{end} \quad \overline{\sum_{i \in I} p\mathbf{a}_i.S_i} = \sum_{i \in I} \bar{p}\mathbf{a}_i.\bar{S}_i$$

where \bar{p} is the dual of the polarity p so that $\bar{?} = !$ and $\bar{!} = ?$. As usual, duality affects the direction of messages but not message types.

We now define the transition system that we use to formalize the notion of correct session type composition and the induced notion of refinement [6].

The transition system makes use of explicit FIFO queues of messages that have been sent asynchronously. A *configuration* is a term $[S, \omega_S][T, \omega_T]$ where S and T are session types equipped with two queues ω_S and ω_T of incoming messages. We write ϵ for the empty queue and we use s, s' , etc. to range over configurations.

► **Definition 1** (Transition Relation [6]). *The transition relation \rightarrow over configurations is the minimal relation satisfying the rules below (plus symmetric ones, omitted):*

1. *if $j \in I$ then $[\sum_{i \in I} !a_i.S_i, \omega_S][T, \omega_T] \rightarrow [S_j, \omega_S][T, \omega_T a_j]$;*
2. *if $j \in I$ then $[\sum_{i \in I} ?a_i.S_i, a_j \omega_S][T, \omega_T] \rightarrow [S_j, \omega_S][T, \omega_T]$.*

We write \rightarrow^ for the reflexive and transitive closure of the \rightarrow relation.*

► **Example 2.** Consider types $S = !a.?a.\text{end} + !b.?b.\text{end}$ and $T = !a.?a.\text{end}$. The configuration $[S, \epsilon][T, \epsilon]$ has the sequence of transitions $[S, \epsilon][T, \epsilon] \rightarrow [?a.\text{end}, \epsilon][T, a] \rightarrow [?a.\text{end}, a][?a.\text{end}, a] \rightarrow [?a.\text{end}, a][\text{end}, \epsilon] \rightarrow [\text{end}, \epsilon][\text{end}, \epsilon]$, which terminates in a configuration with both types equal to end and with both message queues empty. It also has, e.g., another computation $[S, \epsilon][T, \epsilon] \rightarrow [S, a][?a.\text{end}, \epsilon] \rightarrow [?b.\text{end}, a][?a.\text{end}, b]$ which terminates in a configuration with both types different from end and with both message queues not empty. \dashv

Intuitively, the composition of the two types S and T in Example 2 is *incorrect* because it can lead to a deadlocked configuration $[?b.\text{end}, a][?a.\text{end}, b]$ discussed therein. Following Bravetti, Lange and Zavattaro [6] we formalize the correct composition of two types as a *compliance* relation.

► **Definition 3** (Compliance [6]). *Given a configuration s we say that it is a correct composition if, whenever $s \rightarrow^* s'$, then also $s' \rightarrow^* [\text{end}, \epsilon][\text{end}, \epsilon]$.*

Two session types S and T are compliant if $[S, \epsilon][T, \epsilon]$ is a correct composition.

We can now formally state that the types S and T of Example 2 are not *compliant* because of the sequence of transitions discussed at the end of Example 2: $[S, \epsilon][T, \epsilon] \rightarrow [S, a][?a.\text{end}, \epsilon] \rightarrow [?b.\text{end}, a][?a.\text{end}, b]$ leading to a configuration different from the successfully terminated computation $[\text{end}, \epsilon][\text{end}, \epsilon]$, and without outgoing transitions.

Compliance induces a semantic notion of type refinement, as follows.

► **Definition 4** (Refinement [6]). *A session type S refines T , written $S \preceq T$, if, for every R s.t. T and R are compliant, then S and R are also compliant.*

In words, this definition says that a process behaving as T can be “safely” replaced by a process behaving as S when S is a refinement of T . Indeed, the peer process, which is assumed to behave according to some session type R such that R and T are compliant, will still interact correctly after the substitution has taken place. We quote “safely” above since we want to stress that the notion of session correctness being preserved here is not merely a *safety property*, but rather a combination of a safety property (if the interaction gets stuck, then it has successfully terminated) and a liveness property (the interaction can always successfully terminate).

Because of the universal quantification on the type R , Definition 4 provides few insights about what it means for S to be a refinement of T . For this reason, it is important to provide alternative (but equivalent) characterizations of type refinement. Bravetti, Lange and Zavattaro [6] present a characterization of refinement defined coinductively *à la session subtyping* [16], hence called fair asynchronous subtyping, which can be used to prove some interesting and non trivial relations as those discussed in Section 1.

► **Example 5** (Ground station and satellite communication [6]). We now present the session types describing the possible behaviours of the ground station in the satellite communication example presented in Section 1. We use type T to represent the communication behaviour of the automaton T_G in Figure 1: $T = ?\text{tm}.T + ?\text{over}.T'$ where $T' = !\text{tc}.T' + !\text{done}.\text{end}$. We use S for the automaton T'_G : $S = !\text{tc}.S + !\text{done}.S'$ where $S' = ?\text{tm}.S' + ?\text{over}.\text{end}$. Type T specifies the initially expected behaviour of the ground station as a complementary protocol

w.r.t. to the one, T_S , followed by the spacecraft. Type S specifies an implementation where the emission of the telecommands is anticipated w.r.t. the consumption of the received telemetries. The subtyping of [6] allows to prove that $S \preceq T$, hence we can conclude that the protocol with the anticipation of the telemetries correctly implements (refines) the expected communication behaviour of the ground station. \lrcorner

The coinductive subtyping in [6] is a sound but not complete characterization of fair asynchronous refinement. In particular, it is incapable of establishing relations $S \preceq T$ where S describes a more deterministic protocol than T (output covariance). We have seen an instance of this case in Section 1 when discussing the stream and batch processing servers. Liveness-preserving refinements are difficult to characterize because they are intrinsically non-local: the removal of an output action in a region of a session type may compromise the successful termination of the protocol along a branch that is “far away” from the location where the output was removed. In our case this phenomenon is further aggravated by the presence of asynchrony, which allows output actions to be anticipated and therefore to be moved around.

In order to provide a complete characterization of refinement, we use an alternative, queue-less semantics of asynchronous session types that we introduce in the next section.

3 An Alternative Asynchronous Semantics for Session Types

The first step towards our characterization of fair refinement is the definition of a novel asynchronous semantics for session types that does not make use of explicit queues. This alternative semantics will allow us to adapt the complete characterization of fair synchronous subtyping for session types [23, 12] to the asynchronous case.

We now introduce a *labelled transition system* to describe the sequences of input/output actions that are allowed by a session type. A *label* is a pair made of a *polarity* and a *tag* a :

Label $\alpha, \beta ::= p a$

We will refer to the dual of a label α with the notation $\bar{\alpha}$, where $\bar{p a} \stackrel{\text{def}}{=} \bar{p} \bar{a}$.

The labelled transition system (LTS) of asynchronous session types is *coinductively* defined thus:

$$\begin{array}{c} \text{[L-SYNC]} \\ \hline \frac{}{\sum_{i \in I} p a_i . S_i \xrightarrow{p a_k} S_k} k \in I \end{array} \qquad \begin{array}{c} \text{[L-ASYNC]} \\ \hline \frac{\forall i \in I : S_i \xrightarrow{? a_i} T_i}{\sum_{i \in I} ! a_i . S_i \xrightarrow{? a} \sum_{i \in I} ! a_i . T_i} \end{array}$$

The rule [L-SYNC] simply expresses the fact that a session type allows the input/output action described by its topmost prefix, as expected. The rule [L-ASYNC] is concerned with asynchrony and lifts input actions that are enabled deep in a session type, provided that such actions are only prefixed by output actions. The rationale for such “asynchronous transitions” is that the topmost outputs allowed by a session type may have been performed asynchronously, so that the underlying input is enabled even if the output messages have not been consumed yet. For example, the session type $!a. ?b.S$ may evolve either as

$$!a. ?b.S \xrightarrow{!a} ?b.S \xrightarrow{?b} S \quad \text{or as} \quad !a. ?b.S \xrightarrow{?b} !a.S \xrightarrow{!a} S \quad (1)$$

The first transition sequence is the expected one and describes an evolution in which the order of actions is consistent with the syntactic structure of the session type. The second

transition sequence is peculiar to the asynchronous setting and describes an evolution in which the input $?b$ is performed before the output $!a$. Asynchronous transitions seem to clash with the very nature of session types, whose purpose is to impose an order on the actions performed by a process on a channel. To better understand them, we find it useful to appeal to the usual interpretation of labelled transition systems as descriptions of those actions that trigger interactions with the environment: a process that behaves according to the session type $!a.?b.S$ is still required to send a before it starts waiting for b . However, it will be capable of receiving b before a has been consumed by the party with which it is interacting, because a may have been queued. In this sense, we see the queue as an entity associated with the process and not really part of the environment in which the process executes.

It is useful to introduce some additional notation related to actions and transitions of session types. We let φ and ψ range over strings of labels and use ε to denote the empty string. We write $\frac{\alpha_1 \dots \alpha_n}{\alpha_1 \dots \alpha_n}$ for the composition $\frac{\alpha_1}{\alpha_1} \dots \frac{\alpha_n}{\alpha_n}$, we write $S \xrightarrow{\varphi}$ if $S \xrightarrow{\varphi} T$ for some T and we write $S \not\xrightarrow{\varphi}$ if not $S \xrightarrow{\varphi}$.

Two subtle aspects of the LTS deserve further comments. First of all, asynchronous transitions are possible only provided that they are enabled *along every branch* of a session type. For example, if $S = !a.?c.S_1 + !b.(?c.S_2 + ?d.S_3)$, then we have

$$S \xrightarrow{?c} !a.S_1 + !b.S_2 \quad \text{and} \quad S \not\xrightarrow{?d} \quad \text{and} \quad S \xrightarrow{!b?d} S_3$$

The $?c$ -labelled transition is enabled because that input action is available regardless of the message (either a or b) that has been sent. On the contrary, the $?d$ -labelled transition is initially disabled because the input of d is possible only if b has been sent. So, the peer interacting with a process that follows the session type S *must* necessarily consume either a or b in order to understand whether or not it can send d .

Finally, we have to motivate the *coinductive* definition of the LTS. This aspect is related to both asynchrony and fairness. Consider a session type S such that $S = !a.S + !b.?c.T$. The question is whether or not the $?c$ -labelled transition should be enabled from S . In principle, S allows the output of an infinite sequence of a messages and so the $?c$ -labelled transition should be disabled. In this work, however, we make the assumption that this behavior is unrealistic or *unfair*, therefore we want the $?c$ -labelled transition to be enabled. From a technical standpoint, the only derivation that allows us to express an $?c$ -labelled transition for S is the following one

$$\frac{\begin{array}{c} \vdots \\ \frac{}{S \xrightarrow{?c} S'} \text{ [L-ASYNC]} \end{array} \quad \frac{}{?c.T \xrightarrow{?c} T} \text{ [L-SYNC]}}{\frac{}{S \xrightarrow{?c} S'} \text{ [L-ASYNC]}}$$

where $S' = !a.S' + !b.T$. This infinite derivation is legal only if the LTS is coinductively defined.

Technically, concerning our fairness assumption, remember that in this paper we take the viewpoint that sessions are meant to eventually terminate so that neverending communication protocols are excluded. This is reflected both in the notion of session correctness (definition 3) from which refinement is derived (definition 4), as well as in the definition of the asynchronous LTS (rule [L-ASYNC]). Concerning the nature of our fairness assumption, it turns out to be an instance of what van Glabbeek and Höfner [25] call *full fairness* where the property being preserved is the reachability of the $[\text{end}, \epsilon][[\text{end}, \epsilon]$ configuration.

The coinductive definition of the LTS may cast doubts on the significance of the labels of transitions, in the sense that some (input) transitions could be coinductively derivable for a given session type S even if they do not correspond to actions that are actually described

within S . For example, for the pre-session type $S = !a.S$ it would be possible to derive *every* transition of the form $S \xrightarrow{?b} T$ by using infinitely many applications of [L-ASYNC]. The following result rules out this possibility. The key element of the proof is the fact that, because of the well-formedness condition (3), every subtree of a session type contains a leaf of the form **end**.

► **Proposition 6.** *If $S \xrightarrow{\alpha}$, then there exists T and φ made of output actions only such that $S \xrightarrow{\varphi} T \xrightarrow{\alpha}$ and the last transition is derived by [L-SYNC].*

After looking at the transition sequences in (1), the reader may also wonder whether the LTS always satisfies the diamond property when a session type simultaneously enables both input and output actions. Crucially, this is the case.

► **Proposition 7.** *If $S \xrightarrow{?a} S'$ and $S \xrightarrow{!b} S''$, then there exists T such that $S' \xrightarrow{!b} T$ and $S'' \xrightarrow{?a} T$.*

Having established these basic facts about the LTS of session types, we introduce some more notation. If $\varphi = \alpha_1 \cdots \alpha_n$, we write $\varphi.S$ for $\alpha_1 \dots \alpha_n.S$. Note that $\varepsilon.S = S$. Then, we define a notion of *partial derivative* for session types that resembles Brzozowski's derivative for regular expressions [8]. In particular, if $S \xrightarrow{\varphi} T$ we say that T is the derivative of S after φ and we write $S(\varphi)$ for T . Note that $S(\varphi)$ is uniquely defined because of the well-formedness condition (1). We define the *inputs* and *outputs* of a session type as $\text{inp}(S) \stackrel{\text{def}}{=} \{a \mid S \xrightarrow{?a}\}$ and $\text{out}(S) \stackrel{\text{def}}{=} \{a \mid S \xrightarrow{!a}\}$. Note that $\text{inp}(\text{end}) = \text{out}(\text{end}) = \emptyset$. Occasionally we use $\text{out}(\cdot)$ also as a predicate so that $\text{out}(S)$ holds if and only if $\text{out}(S) \neq \emptyset$. Finally, the set of *traces* of a session type is defined as $\text{tr}(S) \stackrel{\text{def}}{=} \{\varphi \mid S \xrightarrow{\varphi} \text{end}\}$.

► **Example 8.** Let us formalize the protocols of the stream and batch processing servers we have sketched in Section 1. Consider the session types

$$S = ?\text{req}.\text{!resp}.S + ?\text{stop}.\text{!stop}.\text{end} \quad \text{and} \quad \begin{aligned} T &= ?\text{req}.T + ?\text{stop}.T' \\ T' &= \text{!resp}.T' + \text{!stop}.\text{end} \end{aligned}$$

and observe that $\text{tr}(T) = \{(? \text{req})^m ? \text{stop} (\text{!resp})^n \text{!stop} \mid m, n \in \mathbb{N}\}$. Note also that $S \xrightarrow{(? \text{req})^n ? \text{stop}} \text{!resp}^n \text{!stop}.\text{end}$ for every $n \in \mathbb{N}$. Therefore, S allows all the sequences of input actions allowed by T , after which it performs a subset of the sequences of output actions allowed by T . At the same time, S allows the anticipation of output actions allowed by T . \dashv

We now define a notion of correctness that adapts, to the new semantics, the notion of correct composition used in Definition 3. A *session composition* is a pair $S \parallel T$ of session types. Session compositions reduce according to the following rule:

$$\frac{S \xrightarrow{\bar{\alpha}} S' \quad T \xrightarrow{\alpha} T' \quad \text{out}(S) \subseteq \text{inp}(T)}{S \parallel T \rightarrow S' \parallel T'} \quad \text{out}(T) \subseteq \text{inp}(S) \quad (2)$$

Notice that we overload \rightarrow and \rightarrow^* , which were previously used to denote the transitions over session configurations in Definition 1. Their actual meaning is made clear by the context. Moreover, note that a session composition is *stuck* (i.e. it does not reduce) if one of the two session types in the composition enables an output for which the corresponding input is disabled in the other session type.

► **Example 9.** Consider the types $S = !a.?a.\text{end} + !b.?b.\text{end}$ and $T = !a.?a.\text{end}$ already discussed in Example 2. We have observed that their configuration $[S, \epsilon] \parallel [T, \epsilon]$ can perform transitions according to the relation in Definition 1. On the contrary, the session composition $S \parallel T$ is stuck because $b \in \text{out}(S)$ while $b \notin \text{inp}(T)$, hence $\text{out}(S) \not\subseteq \text{inp}(T)$. \dashv

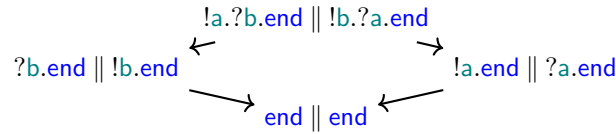
The definition of correct session composition is the same as Definition 3 adapted to the new asynchronous semantics of types. Intuitively, in a correct session composition $S \parallel T$, the only state that is allowed to be stuck is the final one, where both session types have reduced to **end** meaning that the session has successfully terminated. Moreover, every intermediate state $S' \parallel T'$ that is reachable from $S \parallel T$ must itself be on a path that leads to successful termination of the session.

► **Definition 10.** We say that $S \parallel T$ is correct, notation $S \bowtie T$, if $S \parallel T \rightarrow^* S' \parallel T'$ implies that $S' \parallel T' \rightarrow^* \text{end} \parallel \text{end}$.

A simple example of correct session composition is $!a.?b.\text{end} \parallel ?a.!b.\text{end}$, which admits only one reduction sequence

$$!a.?b.\text{end} \parallel ?a.!b.\text{end} \rightarrow ?b.\text{end} \parallel !b.\text{end} \rightarrow \text{end} \parallel \text{end}$$

where every intermediate state can reduce further towards successful termination. Note that a $?b$ -labelled transition is immediately enabled by the session type on the left hand side of the initial composition, but the matching $!b$ -labelled transition becomes enabled only after the first reduction. More interestingly, also the composition $!a.?b.\text{end} \parallel !b.?a.\text{end}$ is correct. In this case the composition may evolve non-deterministically in two ways, depending on which output message is consumed first:



In synchronous theories of (binary) session types duality implies correctness, to the point that in most cases session correctness is expressed in terms of duality. This does not hold for the notion of correctness given by Bravetti, Lange and Zavattaro [6] because they do not consider the well-formedness condition (3). In fact, every type without **end** cannot be correctly composed with any other type, including its dual. We now show that by restricting to types satisfying also the well-formedness condition (3) we recover this property which also plays a key role in the alternative characterization of subtyping discussed in the next section.

► **Proposition 11.** $\bar{S} \bowtie S$ holds for every session type S .

As discussed in Example 9, the presence of FIFO message queues in Definition 1 allows types starting with outputs to emit messages and store them in the queues without performing any check about the possibility for the receiver to consume these messages. On the contrary, the new reduction semantics of session composition (2) checks that all outputs can be consumed by the partner. Despite this difference, the two semantics can be proved “equivalent”. More specifically, we will prove the following correspondence result: two session types S and T are *compliant* (according to Definition 3) if and only if $S \bowtie T$ (according to Definition 10).

In order to state the correspondence result, it is convenient to introduce some additional notation allowing us to consider the configuration queues as sequences of input actions to be executed in order to consume the buffered messages. More precisely, given the queue $\omega = a_1 a_2 \dots a_n$, we write $? \omega$ for the corresponding sequence $?a_1 ?a_2 \dots ?a_n$ of input actions. We also use the notation $S \xrightarrow{? \omega} S'$ as a shortcut for $S \xrightarrow{?a_1 ?a_2 \dots ?a_n} S'$. Notice that the queue ω could be empty, i.e. $\omega = \epsilon$, in which case $S \xrightarrow{? \omega} S$.

In Example 2 we have seen a computation defined according to Definition 1 that cannot be mimicked by our reductions defined by rule (2). We now prove that the vice versa actually holds. More precisely we identify a way to relate session compositions to session configurations, and we prove that each reduction of a session compositions can be mimicked by a corresponding sequence of transitions of configurations. A session composition $S_1 \parallel T_1$ corresponds to several configurations $[S, \omega_S] \parallel [T, \omega_T]$ such that $S \xrightarrow{? \omega_S} S_1$ and $T \xrightarrow{? \omega_T} T_1$. That is, S_1 can be thought of as the residual of S after all the messages in the queue ω_S of incoming messages have been consumed. Similarly for T_1 and T .

► **Example 12.** Consider $S = !a. ?b.end$ and $T = ?a. !b.end$. We have $S \parallel T \rightarrow ?b.end \parallel !b.end$. Similarly, we have the following configuration transition $[S, \epsilon] \parallel [T, \epsilon] \rightarrow [?b.end, \epsilon] \parallel [T, a]$. Configuration $[?b.end, \epsilon] \parallel [T, a]$ is one of those corresponding to $?b.end \parallel !b.end$ because of the buffered message a and the type T such that $T \xrightarrow{?a} !b.end$. Now we have the reduction $?b.end \parallel !b.end \rightarrow end \parallel end$. Configuration $[?b.end, \epsilon] \parallel [T, a]$ can mimick this reduction with the two following transitions $[?b.end, \epsilon] \parallel [T, a] \rightarrow [?b.end, \epsilon] \parallel [!b.end, \epsilon] \rightarrow [?b.end, b] \parallel [end, \epsilon]$. The reached configuration corresponds to $end \parallel end$ because of the buffered message b and the type $?b.end$ such that $?b.end \xrightarrow{?b} end$. \dashv

► **Lemma 13.** Consider the configuration $[S, \omega_S] \parallel [T, \omega_T]$ with $S \xrightarrow{? \omega_S} S_1$ and $T \xrightarrow{? \omega_T} T_1$. If $S_1 \parallel T_1 \rightarrow S'_1 \parallel T'_1$ then $[S, \omega_S] \parallel [T, \omega_T] \rightarrow^* [S', \omega_{S'}] \parallel [T', \omega_{T'}]$ with $S' \xrightarrow{? \omega_{S'}} S'_1$ and $T' \xrightarrow{? \omega_{T'}} T'_1$.

We now investigate the possibility for the reduction relation defined by rule (2) to mimick sequences of transitions of corresponding configurations. This is not true in general, as shown in Example 9. However, we can prove this result under the assumption that the initial session compositions, or the initial configurations, are correct. These two cases are considered in the next two lemmas, the first of which states that a correct session composition $S \parallel T$ can mimick all computations of the configuration $[S, \epsilon] \parallel [T, \epsilon]$ and the reached compositions/configurations are related by the correspondence relation discussed above and used in Lemma 13.

► **Lemma 14.** Let $S \parallel T$ be a correct session composition. If $[S, \epsilon] \parallel [T, \epsilon] \rightarrow^* [S', \omega_{S'}] \parallel [T', \omega_{T'}]$ then $S \parallel T \rightarrow^* S_1 \parallel T_1$ with $S' \xrightarrow{? \omega_{S'}} S_1$ and $T' \xrightarrow{? \omega_{T'}} T_1$.

A similar correspondence result holds also for initial correct configurations.

► **Lemma 15.** Let $[S, \epsilon] \parallel [T, \epsilon]$ be a correct configuration. Consider the transition sequence $[S, \epsilon] \parallel [T, \epsilon] \rightarrow^* [S', \omega_{S'}] \parallel [T', \omega_{T'}]$ and two types S_1 and T_1 s.t. $S' \xrightarrow{? \omega_{S'}} S_1$ and $T' \xrightarrow{? \omega_{T'}} T_1$. If $[S', \omega_{S'}] \parallel [T', \omega_{T'}] \rightarrow [S'', \omega_{S''}] \parallel [T'', \omega_{T''}]$ then $S_1 \parallel T_1 \rightarrow^* S'_1 \parallel T'_1$ with $S'' \xrightarrow{? \omega_{S''}} S'_1$ and $T'' \xrightarrow{? \omega_{T''}} T'_1$.

We can now conclude with the main result of this section, that is the correspondence between compliance (Definition 3) and correctness (Definition 10).

► **Theorem 16.** We have that S and T are compliant (Definition 3) iff $S \bowtie T$ (Definition 10).

4 Fair Asynchronous Session Subtyping

In this section we present the main contribution of the paper, namely a sound and complete characterization of the refinement relation defined by Bravetti, Lange and Zavattaro [6] recalled in Definition 4. In light of Theorem 16, such refinement can be alternatively defined using the new labelled transition system for asynchronous session types defined in Section 3 and the notion of correctness in Definition 10:

$S \preceq T$ if and only if $R \bowtie T$ implies $R \bowtie S$ for every R

This definition corresponds to the definition of a subtyping relation for session following *Liskov's substitution principle* [19], where the property to be preserved is session correctness.

We start the characterization of \preceq by introducing the following coinductive relation which, as we will see later, turns out to be an overapproximation of \preceq .

► **Definition 17.** We say that \mathcal{S} is an asynchronous subtyping relation if $(S, T) \in \mathcal{S}$ implies:

1. if $T = \text{end}$, then $S = \text{end}$;
2. if $T \xrightarrow{?a} T'$, then $S \xrightarrow{?a} S'$ and $(S', T') \in \mathcal{S}$;
3. if $\text{out}(T)$, then $\text{out}(S)$ and $S \xrightarrow{!a} S'$ implies $T \xrightarrow{!a} T'$ and $(S', T') \in \mathcal{S}$.

The clauses of Definition 17 specify some expected requirements for a session subtyping relation: a terminated session type `end` is in relation with itself only; every input action in a supertype T should be matched by the same input action allowed in the subtype S and the corresponding continuations should still be related by subtyping; dually, every output action allowed by a subtype S should be matched by an output action allowed by the supertype T and the corresponding continuations should still be related by subtyping.

Interestingly, these clauses are essentially those found in analogous characterizations of *synchronous subtyping for session types* [16], modulo the different orientation of \preceq due to our viewpoint based on the substitution of processes rather than on the substitution of channels.¹ However, there are a couple of quirks that separate Definition 17 from sibling definitions. First of all, recall that transitions like $T \xrightarrow{?a} T'$ and $S \xrightarrow{?a} S'$ may concern “deep” input actions enabled by T and S , even when T and S begin with output actions. Hence, clause (2) may allow pairs of session types to be related even if they do not start with the same kind of actions. A simple instance of this fact is given by the session types $!a.?b.S$ and $?b.!a.S$ discussed earlier, for which we have $!a.?b.S \xrightarrow{?b} !a.S$ and $?b.!a.S \xrightarrow{?b} !a.S$. Another novelty is that the clauses (2) and (3) are no longer mutually exclusive, since it may happen that $\text{out}(T)$ holds for a T that also allows (deep) input transitions. For example, any asynchronous subtyping relation that includes the pair $(!a.?b.S, !a.?b.S)$ must also include the pair $(?b.S, ?b.S)$ because of clause (3) as well as the pair $(!a.S, !a.S)$ because of clause (2).

► **Example 18.** As a first example of a non trivial asynchronous subtyping relation we consider the session types S and T presented in Example 5 which formalizes the two possible communication protocols for the ground station discussed in Section 1. We report here the definitions of the types for reader's convenience: $S = !tc.S + !done.S'$ where $S' = ?tm.S' + ?over.end$ and $T = ?tm.T + ?over.T'$ where $T' = !tc.T' + !done.end$. Both S and T allow sending an arbitrary number of commands followed by a single `done`. Both S and T allow receiving an arbitrary number of telemetries followed by a single `over`. The difference is that in T the commands can only be sent after all the telemetries have been received, whereas in S the commands can be sent at any time, even before the first telemetry has been received. To see that S and T can be related by an asynchronous subtyping, observe that we have

$$\frac{\frac{\vdots}{S \xrightarrow{?tm} S} \quad \frac{\vdots}{S' \xrightarrow{?tm} S'} \quad [L\text{-SYNC}]}{S \xrightarrow{?tm} S} [L\text{-ASYNC}] \quad \text{and} \quad \frac{\frac{\vdots}{S \xrightarrow{?over} T'} \quad \frac{\vdots}{S' \xrightarrow{?over} end} \quad [L\text{-SYNC}]}{S \xrightarrow{?over} T'} [L\text{-ASYNC}]$$

¹ The interested reader may refer to Gay [15] for a comparison of the two viewpoints.

therefore $\mathcal{S} \stackrel{\text{def}}{=} \{(S, T), (T', T'), (\text{end}, \text{end})\}$ is an asynchronous subtyping relation. At the same time, no asynchronous subtyping relation includes the pair (T, S) since it violates the clause (3): $\text{out}(S)$ holds whereas $\text{out}(T)$ does not. Indeed, the session type \bar{S} relies on those early outputs performed by S in order to make progress and $\bar{S} \parallel T$ is stuck. \perp

► **Example 19.** Consider again the types $S = ?\text{req}.\text{!resp}.S + ?\text{stop}.\text{!stop.end}$ and $T = ?\text{req}.T + ?\text{stop}.T'$ where $T' = \text{!resp}.T' + \text{!stop.end}$ respectively modeling the protocols of the stream and batch processing servers defined in Example 8. To show that S and T are related by an asynchronous subtyping relation it is sufficient to show that the relation

$$\mathcal{S} \stackrel{\text{def}}{=} \{(\text{!resp}^n.S, T), (\text{!resp}^n.\text{!stop.end}, T') \mid n \in \mathbb{N}\} \cup \{(\text{end}, \text{end})\}$$

is an asynchronous subtyping because $(S, T) \in \mathcal{S}$. Pairs $(\text{!resp}^n.S, T)$ are necessary to deal with the transition $T \xrightarrow{?req} T$ which is matched by $\text{!resp}^n.S \xrightarrow{?req} \text{!resp}^{n+1}.S$. The pairs $(\text{!resp}^n.\text{!stop.end}, T')$ account for the transition $T \xrightarrow{?stop} T'$ which is matched by $\text{!resp}^n.S \xrightarrow{?req} \text{!resp}^n.\text{!stop.end}$. Notice that in pairs of the form $(\text{!resp}^{n+1}.\text{!stop.end}, T')$ the first type has a transition $\text{!resp}^{n+1}.\text{!stop.end} \xrightarrow{!resp} \text{!resp}^n.\text{!stop.end}$ matched by $T' \xrightarrow{!resp} T'$. Finally, $\text{!stop.end} \xrightarrow{!stop} \text{end}$ is matched by $T' \xrightarrow{!stop} \text{end}$. \perp

We now present a first result relating \preceq and asynchronous subtyping. We have that \preceq satisfies the clauses of Definition 17 hence it is an asynchronous sbtying relation.

► **Theorem 20.** \preceq is an asynchronous subtyping relation.

As we have anticipated, the largest asynchronous subtyping relation contains pairs of types that are not in subtyping relation, as illustrated by the next example.

► **Example 21.** Consider two variants of the batch processing server whose responses may be positive (**yes**) or negative (**no**). Their behavior is described by the session types $S = ?\text{req}.\text{!no}.S + ?\text{stop}.\text{!stop.end}$ and $T = ?\text{req}.(\text{!yes}.T + \text{!no}.T) + ?\text{stop}.\text{!stop.end}$. The server behaving as S always responds **no**. The server behaving as T may respond in either way. Let $T^0 \stackrel{\text{def}}{=} T$ and $T^{n+1} \stackrel{\text{def}}{=} \text{!yes}.T^n + \text{!no}.T^n$. It is relatively easy to see that $\mathcal{S} \stackrel{\text{def}}{=} \{(\text{!no}^n.S, T^n) \mid n \in \mathbb{N}\} \cup \{(\text{!stop.end}, \text{!stop.end}), (\text{end}, \text{end})\}$ is an asynchronous subtyping relation, and yet $S \not\preceq T$. Indeed, consider the session type $R \stackrel{\text{def}}{=} \text{!req}.R'$ where $R' = ?\text{yes}.\text{!stop.end} + ?\text{no}.R$. Then $R \bowtie T$ holds but $R \bowtie S$ does not. Basically, (a process behaving as) R insists on sending requests until it receives a positive response. At that point, it is satisfied and quits the interaction. However, only (a process behaving as) T is willing to send a positive response, whereas (a process behaving as) S is not. \perp

In general, a purely coinductive characterization based on the clauses of Definition 17 allows us to capture the *safety-preserving properties* of subtyping, those concerning the admissibility of interactions, because they are supported by an invariant argument. However, the very same clauses do not say anything about the *liveness-preserving properties* of subtyping, those concerning the reachability of successful termination, which must be supported by a well-foundedness argument. In order to find a sound characterization of subtyping, we have to resort to *bounded coinduction* [1, 14], a particular case of coinductive definition where we consider the largest relation that satisfies the clauses in Definition 17 and that is also *included in an inductively defined relation* that preserves the reachability of successful termination. We dub this inductive relation *convergence* and we denote it by \sqsubseteq .

Before looking at the formal definition of \sqsubseteq , let us try to acquire a rough understanding of convergence by recalling Example 21, in which we have identified two session types S and T that satisfy the clauses of Definition 17 but are not related by subtyping. In that

example, we can see that the traces that lead S to termination are a subset of the traces that lead T to termination. This is a general property that holds every time S describes a more deterministic (or “less demanding”) behavior compared to T . However, in the specific case of Example 21, the traces that lead S to termination are *too few*, to the point that there exists a potential partner (described by R in the example) that terminates solely relying on those traces of T that have disappeared in S . Contrast S and T those with $S' = !a.!a.S' + !b.end$ and $T' = !a.T' + !b.end$. Also in this case S' is more deterministic than T' and some traces that lead T' to termination have disappeared in S' . Specifically, every trace of the form $\varphi = !a^{2n+1}!b$ is allowed by T' but not by S' . However, it is also the case that for each of these traces we can find another trace $!a^{2n}!b$ that shares a common prefix with φ and that leads both S' and T' to termination *after an output action* $!b$. Since the behavior described by T' is always able to autonomously veer the interaction towards termination after *any number* of a messages, any session type R that can be correctly combined with T' must be prepared to receive this b message at any time, meaning that termination is preserved also if R is combined with S' .

Let us now define \sqsubseteq formally.

► **Definition 22.** Convergence is the relation \sqsubseteq inductively defined by the rule:

$$\frac{\forall \varphi \in \text{tr}(T) \setminus \text{tr}(S) : \exists \psi \leq \varphi, a : S(\psi!a) \sqsubseteq T(\psi!a)}{S \sqsubseteq T}$$

Intuitively, a derivation for the relation $S \sqsubseteq T$ measures the “difference” between S and T in terms of allowed traces. When $\text{tr}(T) \subseteq \text{tr}(S)$ there is essentially no difference between S and T , so the rule that defines \sqsubseteq has no premises and turns into an axiom. When $\text{tr}(T) \not\subseteq \text{tr}(S)$, then for every trace φ of actions allowed by T but not by S there is a prefix $\psi \leq \varphi$ shared by both S and T and followed by an output action $!a$ that leads to a pair of session types $S(\psi!a)$ and $T(\psi!a)$ that are “slightly less different” from each other in terms of allowed traces. Indeed, the derivation for $S(\psi!a) \sqsubseteq T(\psi!a)$ must be strictly smaller than that for $S \sqsubseteq T$, or else $S \sqsubseteq T$ would not be inductively derivable.

Note that \sqsubseteq is trivially reflexive, but apart from that it is generally difficult to understand when two session types are related by convergence because of the *non-local* flavor of the relation. However, it is easy to see that any two session types related by an asynchronous subtyping where at least one of them is finite are also related by convergence.

► **Proposition 23.** If \mathcal{S} is an asynchronous subtyping relation such that $(S, T) \in \mathcal{S}$ and at least one among S and T is finite, then $S \sqsubseteq T$.

Proof. A simple induction on either S or T , depending on which one is finite. ◀

When both S and T are infinite, understanding whether $S \sqsubseteq T$ holds or not may require some non-trivial reasoning on their traces. Let us work out a few examples.

► **Example 24.** Let $S = !a.!a.S + !b.end$ and $T = !a.T + !b.end$. To prove that $S \sqsubseteq T$, consider $\varphi \in \text{tr}(T) \setminus \text{tr}(S)$. It must be the case that $\varphi = !a^{2n+1}!b$ for some n . Take $\psi \stackrel{\text{def}}{=} !a^{2n}$. Now $S(\psi!b) = T(\psi!b) = end$, hence $S(\psi!b) \sqsubseteq T(\psi!b)$ by reflexivity of \sqsubseteq . ◻

► **Example 25.** Consider again $S = ?req.!no.S + ?stop.!stop.end$ and $T = ?req.(!yes.T + !no.T) + ?stop.!stop.end$ from Example 21. Since we conjecture $S \not\sqsubseteq T$ we can focus on a particular $\varphi \in \text{tr}(T) \setminus \text{tr}(S)$, namely $\varphi = ?req!yes\varphi' \in \text{tr}(T) \setminus \text{tr}(S)$. Note that $\text{out}(S)$ does not hold, so the only prefix of φ that we may reasonably consider is $\psi \stackrel{\text{def}}{=} ?req$ and the only output action that S may perform after such prefix is $!no$. But then $S(\psi!no) = S$ and $T(\psi!no) = T$, hence it is not possible to build a well-founded derivation for $S \sqsubseteq T$. We conclude $S \not\sqsubseteq T$. ◻

► **Example 26.** Let us prove that the session types S and T in Example 8 satisfy the relation $S \sqsubseteq T$. Consider $\varphi \in \text{tr}(T) \setminus \text{tr}(S)$. It must be the case that $\varphi = ?\text{req}^m?\text{stop!resp}^n!\text{stop}$ for some $m, n \in \mathbb{N}$ with $m \neq n$. If $m < n$, then we can take $\psi \stackrel{\text{def}}{=} ?\text{req}^m?\text{stop!resp}^m \leq \varphi$ and now $S(\psi!\text{stop}) = T(\psi!\text{stop}) = \text{end}$ and we conclude by reflexivity of \sqsubseteq . If $m > n$, then we can take $\psi \stackrel{\text{def}}{=} ?\text{req}^m?\text{stop!resp}^n \leq \varphi$ and now $S(\psi!\text{resp}) = !\text{resp}^{m-n-1}!\text{stop.end} \sqsubseteq T' = T(\psi!\text{resp})$ using Proposition 23 (because the asynchronous subtyping relation \mathcal{S} of Example 19 contains all pairs $(!\text{resp}^n!\text{stop.end}, T')$, for every n , and the type and the types $!\text{resp}^{m-n-1}!\text{stop.end}$ is finite). \sqcup

We are finally ready to state our main result which confirms that \sqsubseteq indeed provides the inductively defined space within which we can characterize \preceq as the largest asynchronous subtyping relation. Note that the characterization is both sound and complete.

► **Theorem 27.** \preceq is the largest asynchronous subtyping relation included in \sqsubseteq .

► **Example 28.** We are finally able to show that S and T in Example 8 are such that $S \preceq T$. In Example 19 we have considered the following asynchronous subtyping relation:

$$\mathcal{S} \stackrel{\text{def}}{=} \{(!\text{resp}^n.S, T), (!\text{resp}^n!\text{stop.end}, T') \mid n \in \mathbb{N}\} \cup \{(\text{end}, \text{end})\}$$

We now prove that all the pairs in \mathcal{S} are also included in \sqsubseteq , confirming that $S \preceq T$ by Theorem 27. In Example 26 we have already shown that $S \sqsubseteq T$. We can use similar arguments to show that also the pairs $(!\text{resp}^n.S, T)$ are such that $!\text{resp}^n.S \sqsubseteq T$. Consider $\varphi \in \text{tr}(T) \setminus \text{tr}(!\text{resp}^n.S)$. It must be the case that $\varphi = ?\text{req}^m?\text{stop!resp}^l!\text{stop}$ for some $m, l \in \mathbb{N}$ with $m + n \neq l$. If $m + n < l$, then we can take $\psi \stackrel{\text{def}}{=} ?\text{req}^m?\text{stop!resp}^{m+n} \leq \varphi$ and now $S(\psi!\text{stop}) = T(\psi!\text{stop}) = \text{end}$ and we conclude by reflexivity of \sqsubseteq . If $m + n > l$, then we can take $\psi \stackrel{\text{def}}{=} ?\text{req}^m?\text{stop!resp}^l \leq \varphi$ and now $S(\psi!\text{resp}) = !\text{resp}^{m+n-l-1}!\text{stop.end} \sqsubseteq T' = T(\psi!\text{resp})$ using Proposition 23. The pairs $(!\text{resp}^n!\text{stop.end}, T')$, $(!\text{stop.end}, T')$, and (end, end) all contain at least one finite type, hence they are included in \sqsubseteq by Proposition 23. \sqcup

5 Related Work and Concluding Remarks

Gay and Hole [16] introduced the first notion of subtyping for *synchronous* session types. This subtyping supports variance on both inputs and outputs so that a subtype can have more external nondeterminism (by adding branches in input choices) and less internal nondeterminism (by removing branches in output choices). Padovani [23] studied a notion of fair subtyping for *synchronous* multi-party session types that preserves a notion of correctness similar to our Definition 3. One key difference between Padovani's fair subtyping and Gay-Hole subtyping is that the variance of outputs must be limited in such a way that an output branch can be pruned only if it is not necessary to reach successful termination. Ciccone and Padovani [23, 11, 12] have presented sound and complete characterizations of fair subtyping in the synchronous case. These characterizations all combine a coinductively defined relation and an inductively defined relation (called “convergence”), which respectively capture the safety-preserving and the liveness-preserving aspects of subtyping.

Subtyping relations for asynchronous session types have been defined by Mostrous and Yoshida [21], Chen *et al.* [9] and Bravetti, Lange and Zavattaro [6]. In the asynchronous case a subtype can anticipate output actions w.r.t. its supertype because anticipated outputs can be stored in the communication queues without altering the overall interaction behaviour. The first proposal for asynchronous subtyping [21] allows a subtype to execute loops of anticipated outputs, thus delaying input actions indefinitely. This could leave orphan messages in the

buffers. The subsequent work of Chen *et al.* [9] imposes restrictions on output anticipation so as to avoid orphan messages. Bravetti, Lange and Zavattaro [6] have defined the first fair (liveness-preserving) asynchronous session subtyping along the lines of Definition 4. However, they only provided a sound (but not complete) coinductive characterization.

In this paper we present the first complete characterization of the fair asynchronous subtyping relation defined by Bravetti, Lange and Zavattaro [6] using the techniques introduced by Ciccone and Padovani in the synchronous case [23, 11, 12]. In particular, we complement a coinductively defined subtyping relation with a notion of convergence. To do so, we leverage a novel “asynchronous” operational semantics of session types in which a type can perform an input transition even if such input is prefixed by outputs, under the assumption that such input is present along *every* initial sequence of outputs.

Another fair asynchronous subtyping relation that makes use of a similar operational semantics has been recently defined by Padovani and Zavattaro [24]. Unlike the relation defined by Bravetti, Lange and Zavattaro [6] and characterized in this paper, the subtyping relation of Padovani and Zavattaro [24] preserves a liveness property that is strictly weaker than successful session termination. For this reason, their operational semantics is completely symmetric with respect to the treatment of input and output actions and no inductively-defined convergence relation is necessary in order to obtain a sound and complete characterization of subtyping.

A notable difference between the paper of Bravetti, Lange and Zavattaro [6] and our own is the fact that we focus on those session types describing protocols that can always terminate, whereas Bravetti, Lange and Zavattaro make no such assumption. This choice affects the properties of the subtyping relation. In particular, Bravetti, Lange and Zavattaro [6] show that a supertype can have additional input branches provided that such branches are *uncontrollable*. A session type is uncontrollable if there are no session types that can be correctly composed with it. Our well-formedness condition (3) in the definition of types, guarantees that all session types are controllable. We argue that the characterization presented in this paper can be easily extended to the more general case where uncontrollable session types are allowed, following the approach considered by Padovani [23]. Basically, types that do not satisfy the well-formedness condition (3) can be *normalized* by pruning away the uncontrollable subtrees. The normalization yields session types that are semantically equivalent to the original ones and that satisfy the condition (3). At that point, our characterization can be used to establish whether or not they are related by subtyping.

We envision three lines of development of this work. One line is concerned with the study of algorithmic versions of our notions of correct composition, subtyping, and convergence. As for all the known subtypings for asynchronous sessions, these notions turn out to be undecidable [7]. In the literature, sound (but not complete) algorithmic characterizations have been investigated for different variants of asynchronous session subtyping [4, 5, 2, 6]. We plan to investigate the possibility to adapt these approaches to our new formalization of fair asynchronous subtyping, possibly taking advantage of the novel operational semantics for asynchronous session types. Another line of future work concerns the application of fair asynchronous subtyping to the definition of type systems ensuring successful session termination of asynchronously communicating processes. Such type systems have been studied for *synchronous* communication by Ciccone, Dagnino and Padovani [11, 10]. A first proposal of such type system for *asynchronous* communication has been recently given by Padovani and Zavattaro [24]. However, as we have pointed out above, the subtyping relation used in their type system does not preserve successful session termination. As a consequence, their type system requires a substantial amount of additional annotations in types and in

typing judgements in order to enforce successful termination. It may be the case that relying on a subtyping relation that provides stronger guarantees, like the one characterized in the present paper, could simplify the type system and possibly enlarge the family of typeable processes.

Finally, it may be worth investigating whether the characterization of fair asynchronous subtyping studied in this paper for *binary* session types extends smoothly to *multiparty* session types as well. In the synchronous setting, it is known that there are no substantial differences between the binary and the multiparty case, except for the presence of *role names* in session types [23, 11]. We conjecture that an asynchronous LTS for multiparty session types can be defined in a very natural way along the lines presented in this paper. However, the multiparty case grants additional out-of-order transitions compared to the binary case (see the *unfair* multiparty asynchronous subtyping defined by Ghilezan *et al.* [17]) and working out all of the resulting technicalities could be challenging.

References

- 1 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
- 2 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.
- 3 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
- 4 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping. In *CONCUR*, volume 140 of *LIPICs*, pages 38:1–38:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CONCUR.2019.38.
- 5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: <https://lmcs.episciences.org/7238>.
- 6 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair asynchronous session subtyping. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:5)2024.
- 7 Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro. A Sound and Complete Characterization of Fair Asynchronous Session Subtyping. Technical report, University of Bologna, 2025. doi:10.48550/arXiv.2506.06078.
- 8 Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.
- 9 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.
- 10 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024. doi:10.1016/J.JLAMP.2024.100964.
- 11 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.

- 12 Luca Ciccone and Luca Padovani. Inference systems with corules for combined safety and liveness properties of binary session types. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/LMCS-18(3:27)2022.
- 13 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.
- 14 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.
- 15 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.
- 16 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.
- 17 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.
- 18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. doi:10.1145/2827695.
- 19 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.
- 20 Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
- 21 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π -calculus. *Inf. Comput.*, 241:227–263, 2015. doi:10.1016/J.IC.2015.02.002.
- 22 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.
- 23 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.
- 24 Luca Padovani and Gianluigi Zavattaro. Fair termination of asynchronous binary sessions, 2025. doi:10.48550/arXiv.2503.07273.
- 25 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.