On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping

Mario Bravetti, Marco Carbone, Gianluigi Zavattaro

On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping

Mario Bravetti

University of Bologna, Department of Computer Science and Engineering / INRIA FOCUS

Marco Carbone

Department of Computer Science, IT University of Copenhagen

Gianluigi Zavattaro

University of Bologna, Department of Computer Science and Engineering / INRIA FOCUS

Abstract

Session types are behavioural types for guaranteeing that concurrent programs are free from basic communication errors. Recent work has shown that asynchronous session subtyping is undecidable. However, since session types have become popular in mainstream programming languages in which asynchronous communication is the norm rather than the exception, it is crucial to detect significant decidable subtyping relations. Previous work considered extremely restrictive fragments in which limitations were imposed to the size of communication buffer (at most 1) or to the possibility to express multiple choices (disallowing them completely in one of the compared types). In this work, for the first time, we show decidability of a fragment that does not impose any limitation on communication buffers and allows both the compared types to include multiple choices for either input or output, thus yielding a fragment which is more significant from an applicability viewpoint. In general, we study the boundary between decidability and undecidability by considering several fragments of subtyping. Notably, we show that subtyping remains undecidable even if restricted to not using output covariance and input contravariance.

Keywords: Session Types, Asynchronous Subtyping, Undecidability

---

1. Introduction

Session types [1, 2] are types for controlling the communication behaviour of processes over channels. In a very simple but effective way, they express the pattern of sends and receives that a process must perform. Since they can guarantee freedom from some basic programming errors, session types are becoming popular with many main stream language implementations, e.g., Haskell [3], Go [4] or Rust [5].

As an example, consider a client that invokes service operations by following the protocol expressed by the session type

$$\oplus\{op1\colon \&\{resp1\colon \mathbf{end}\},\ op2\colon \&\{resp2\colon \mathbf{end}\}\}$$

indicating that the client decides whether to call operation $op1$ or $op2$ and then waits for receiving the corresponding response ($resp1$ or $resp2$, respectively). For the sake of simplicity we consider session types where (the type of) communicated data is abstracted away. The symmetric behaviour of the service is represented by the complementary (so-called *dual*) session type

$$\&\{op1\colon \oplus\{resp1\colon \mathbf{end}\},\ op2\colon \oplus\{resp2\colon \mathbf{end}\}\}$$

indicating that the server receives the call to operation $op1$ or $op2$ and then sends the corresponding response ($resp1$ or $resp2$, respectively).

We call *output selection* the construct $\oplus\{l_1 : T_1, \ldots, l_n : T_n\}$. It is used to denote a point of choice in the communication protocol: each choice has a label l_i and a continuation T_i. In communication protocols, when there is a point of choice, there is usually a peer that internally takes the decision and the other involved peers receive communication of the selected branch. Output selection is used to describe the behaviour of the peer that takes the decision: indeed, in our example it is the client that decides which operation to call. Symmetrically, we call *input branching* the construct $\&\{l_1 : T_1, \ldots, l_n : T_n\}$. It is used to describe the behaviour of a peer that receives communication of the selection done by some other peers. In the example, indeed, the service receives from the client the decision about the selected operation.[1]

When composing systems whose interaction protocols have been specified with session types, it is significant to consider variants of their specifications that still preserve safety properties. In the above example, the client can be safely replaced by another one with session type

$$\oplus\{op1\colon \&\{resp1\colon \mathbf{end}\}\}$$

indicating that it can call only one specific service operation. But also the service can be safely replaced by another one accepting an additional operation:

$$\&\{op1\colon \oplus\{resp1\colon \mathbf{end}\},\ op2\colon \oplus\{resp2\colon \mathbf{end}\},\ op3\colon \oplus\{resp3\colon \mathbf{end}\}\}$$

Subtyping relations have been formally defined for session types, e.g., by Gay and Hole [7] and Chen et al. [8], in order to precisely capture this safe replacement notion. For instance, a subtyping relation like that of Gay and Hole [7] (denoted

[1] In session type terminology [1, 6], the output selection/input branching constructs are usually simply called *selection/branching*; we call them output selection/input branching because we consider a simplified syntax for session types in which there is no specific separate construct for sending one output/receiving one input. Anyway, such output/input types can be seen as an output selection/input branching with only one choice.

by $\leq_{\mathsf{s}}$), where processes are assumed to simply communicate via synchronous channels[2], would imply, for the client, that:

$$\oplus\{op1\colon \&\{resp1\colon \mathbf{end}\}\} \leq_{\mathsf{s}} \oplus\{op1\colon \&\{resp1\colon \mathbf{end}\},\ op2\colon \&\{resp2\colon \mathbf{end}\}\}$$

according to the so-called *output covariant* property, while, for the server

$$\begin{array}{l}\&\{op1\colon \oplus\{resp1\colon \mathbf{end}\},\ op2\colon \oplus\{resp2\colon \mathbf{end}\},\ op3\colon \oplus\{resp3\colon \mathbf{end}\}\} \\ \qquad\qquad \leq_{\mathsf{s}} \&\{op1\colon \oplus\{resp1\colon \mathbf{end}\},\ op2\colon \oplus\{resp2\colon \mathbf{end}\}\}\end{array}$$

according to the so-called *input contravariant* property.

When processes communicate via asynchronous channels, a more generous notion of subtyping $\leq$ like that of Chen et al. [8] can be considered. E.g., a process using an asynchronous channel to call a service operation, that receives the corresponding response and then sends a huge amount of data (requiring heavy computation), could be safely replaced by a more efficient one that computes and sends all the data immediately without waiting for the response:

$$\oplus\{op\colon \oplus\{huge\_data\colon \&\{resp\colon \mathbf{end}\}\}\} \leq \oplus\{op\colon \&\{resp\colon \oplus\{huge\_data\colon \mathbf{end}\}\}\}$$

Intuitively, this form of asynchronous subtyping reflects the possibility to *anticipate the output* of the huge data w.r.t. to the input of the response because such data are stored in a buffer waiting for their reader to consume them.

1.1. Previous Results

Recently, Bravetti et al. [10] and Lange and Yoshida [11] have independently shown that asynchronous subtyping (the subtyping relation with output anticipation) is undecidable. In particular, in Bravetti et al., this is done by showing undecidability of the much simpler *single-choice relation* $\ll$ that is defined as a restriction of asynchronous subtyping $\leq$ where related $T \ll S$ types are such that: all output selections in T have a single choice (output selections are covariant, thus S is allowed have output selections with multiple choices) and all input branchings in S have a single choice (input branchings are contravariant, thus T is allowed to have input branchings with multiple choices). Moreover, those papers prove decidability for very small fragments of the asynchronous subtyping relation: the most significant one basically requires one of the two compared types to be such that *all its input branchings and output selections have a single choice*. In particular, in Bravetti et al. this is done by showing decidability of the two relations $\ll_{\mathsf{sin}}$ (*single-choice input* $\ll$) and $\ll_{\mathsf{sout}}$ (*single-choice output* $\ll$), both defined as *further* restrictions of $\ll$ where for related $T \ll_{\mathsf{sin}} S$ ($T \ll_{\mathsf{sout}} S$, resp.) types we *additionally* require that: all input branchings (output selections, resp.) in T and S have a single choice. Other decidable fragments, considered by Lange and Yoshida, pose limitations on the communication behaviour that

[2]Here, we focus on the so-called process-oriented subtyping, as opposed to channel-based subtyping [9].

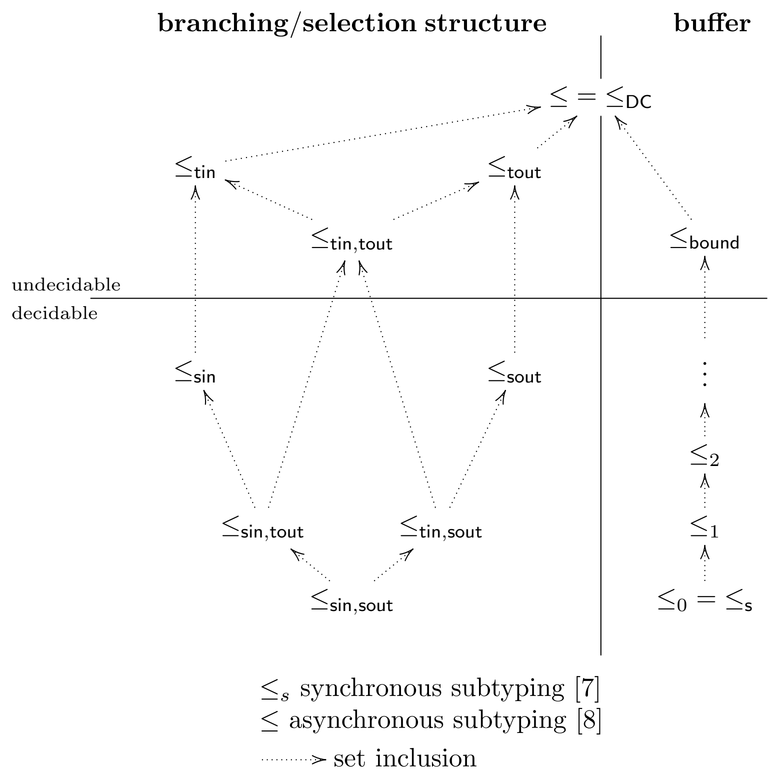

Figure 1: Lattice of the asynchronous subtyping relations considered in this paper.

causes communication buffers to store at most one message, or they are used in half duplex modality (messages can be sent in one direction, only if the buffer for the opposite direction has been emptied). Although asynchronous subtyping is undecidable, it is important to reason about more significant cases for which such a relation is decidable. This because session types have become popular in mainstream programming languages, and, in such cases, asynchronous communications are the norm rather than the exception.

1.2. Contributed Results

The aim of this paper is to detect significant decidable fragments of asynchronous session subtyping and to establish a more precise boundary between decidability and undecidability. In particular, concerning decidability, as discussed above, the few decidable fragments of asynchronous subtyping known so far are extremely restrictive: our relations $\ll_{\mathsf{sin}}$, $\ll_{\mathsf{sout}}$ [10] and the decidable relations considered by Lange and Yoshida [11]. Here, for the first time, we show decidability of a fragment that does not impose any limitation on communication buffers and allows both the subtype and the supertype to include multiple choices (either for input branchings or for output selections), thus opening the possibility for some practical applicability in restricted specific scenarios (e.g. session types for clients/services in web-service systems, see below). More precisely, while $\ll_{\mathsf{sin}}$ ($\ll_{\mathsf{sout}}$, resp.), being it defined as a restriction of $\ll$, admits multiple choices only for output selections in the supertype (input branchings in the subtype, resp.), here we consider and show decidability for a much larger relation, we denote by $\leq_{\mathsf{sin}}$ ($\leq_{\mathsf{sout}}$, resp.). Such a relation is defined as the *restriction of the whole* $\leq$ *relation* (instead of the $\ll$ relation), where, in related types, all input branchings (output selections, resp.) must have a single choice. Therefore, differently from $\ll_{\mathsf{sin}}$ ($\ll_{\mathsf{sout}}$, resp.), in $\leq_{\mathsf{sin}}$ ($\leq_{\mathsf{sout}}$, resp.) *both the subtype and the supertype can include multiple choices* for output selections (for input branchings, resp.). The combination of non restricted buffers and presence of multiple choices on both related types requires a *totally new approach* for guaranteeing the termination of the subtyping algorithm (both for the termination condition itself and for the related decidability proof). For instance, if multiple choices are admitted for input branchings, the termination condition has to deal with complex recurrent patterns to be checked on the leafs of trees representing input branchings (with multiple choices), instead of detecting simple repetitions on strings representing sequences of single-choice inputs (as in previous work [10, 11]).

Concerning undecidability, all previous results [10, 11] exploit the capability of asynchronous subtyping of matching input branchings/output selections by means of covariance/contravariance. We here show that asynchronous subtyping remains undecidable even if we restrict it by disallowing this feature. As asynchronous subtyping is based on the combination of output covariance/input contravariance and output anticipation deriving from asynchronous communication, our result means that (provided that the syntax of types is not constrained) the source of undecidability is to be precisely localized into the output anticipation capability. The undecidability proof has a structure similar to that of Bravetti et al. [10], where the termination problem for queue machines, a well-known Turing-equivalent formalism, is encoded into asynchronous session subtyping. However, differently from Bravetti et al. [10], not having covariance/contravariance makes it impossible to encode queue machines deterministically. As we will see, the need to cope with nondeterminism makes it necessary to restrict the class of encodable queue machines to a new ad-hoc fragment that we introduce in this paper and we prove being Turing-equivalent: single consuming queue machines. Moreover a much more complex encoding, that uses nondeterminism, must be adopted.

In general, the contribution of this work is to analyze restrictions of asynchronous subtyping and classifying them into decidable and undecidable fragments. More precisely, as detailed in the following, we focus on two kinds of restrictions of asynchronous subtyping: limitations to the branching/selection structure and to the communication buffer, giving rise to the numerous relations shown in the lefthand part and righthand part of Figure 1, respectively (see Section 2 for formal definitions of all the relations). The relations are depicted as a lattice according to their inclusion as sets of pairs. Notice that decidability/undecidability is not logically related to set inclusion (e.g. the emptyset and the set of all pairs are both decidable and are the bottom and the top of the lattice).

Concerning asynchronous subtyping $\leq$ itself, we consider the *orphan message free* notion of subtyping introduced in Chen et al. [8]: it is commonly recognized that a convenient notion of asynchronous subtyping should prevent existence of messages that remain "orphan", i.e. that are never consumed from the communication buffer. Operationally, this implies that inputs in the supertype cannot be indefinitely delayed by output anticipation: eventually such inputs must be performed by the subtype so to correspondingly consume messages from the buffer. As a side result, in this paper we introduce a new, elegant, way of defining orphan message free asynchronous subtyping. The new definition is based on just adding a constraint about closure under duality to the standard (non orphan message free) coinductive definition of asynchronous subtyping [12]. We thus use such a novel approach based on dual closeness to give a concise definition of asynchronous subtyping: we call $\leq_{\mathsf{DC}}$ the obtained relation. We then show $\leq_{\mathsf{DC}}$ (also included in Figure 1) to be equal to orphan message free subtyping $\leq$ of Chen et al. [8]. We also show that the most significant decidable and undecidable fragments of $\leq$, i.e. the $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ relation and subtyping without covariance/contravariance, are dual closed subtyping relations according to our new definition.

Limitations to the branching/selection structure. We limit the asynchronous subtyping $\leq$ capability of managing input branchings/output selections, giving rise to the subtyping relations shown in the lefthand part of Figure 1, as follows:

- requiring that in both the subtype and the supertype output selections (input branchings, resp.) have a *single choice*: in this case the sout (sin, resp.) subscript is added to $\leq$;
- requiring that each output selection (input branching, resp.) performed by the subtype is matched by an output selection (input branching, resp.) performed by the supertype with the exactly the same *total* set of labels, i.e. output covariance (input contravariance, resp.) is not admitted: in this case the tout (tin, resp.) subscript is added to $\leq$.

We now summarize our decidability/undecidability results for these relations.

- *Decidability of asynchronous subtyping for single-in types ($\leq_{\mathsf{sin}}$) or single-out types ($\leq_{\mathsf{sout}}$).* We consider the class of single-in (single-out, resp.) session types, i.e. types where all output selections (input branchings, resp.) have a single choice. We present and prove correct an algorithm for deciding whether two single-out (resp. single-in, by exploiting the closure under duality property) types are in the subtyping relation. From a modeling viewpoint, assuming binary sessions to happen between a single-in and a single-out party, this entails that internal decisions are taken by the single-in party, while the single-out one passively accepts them. This kind of behaviour can occur in, e.g., web-service systems where a client internally chooses a request-response operation [13] and then waits for a corresponding (non branching) input, while the server accepts invocations on several operations but then it reacts by answering on a related response channel (independently of the actual returned data/error); see the examples at the beginning of this Introduction section. Our algorithms for subtyping of single-out/single-in types could thus be used in typing systems for server/client code. With minor variants to the machinery introduced to show decidability of $\leq_{\mathsf{sin}}$ and $\leq_{\mathsf{sout}}$, we also show that $\leq_{\mathsf{sin,tout}}$, $\leq_{\mathsf{tin,sout}}$ and $\leq_{\mathsf{sin,sout}}$ are decidable.

- *Undecidability of Asynchronous Subtyping without Output Covariance and Input Contravariance ($\leq_{\mathsf{tin,tout}}$).* As discussed above, subtyping for session types makes use of output covariance and input contravariance: an output $\oplus\{l_i : T_i\}_{i\in I}$ is a subtype of an output with more labels $\oplus\{l_j : T_j\}_{j\in J}$, for $I \subset J$; and an input $\&\{l_i : T_i\}_{i\in I}$ is a subtype of an input with less labels $\&\{l_j : T_j\}_{j\in J}$, for $J \subset I$. Existing results on the undecidability of asynchronous subtyping exploit its capability of relating types with a different number of branches. We consider a restricted form of subtyping, the $\leq_{\mathsf{tin,tout}}$ relation, which disallows this feature, i.e. which does not use output covariance and input contravariance. We show that, also with such a restriction, subtyping remains undecidable by encoding the termination problem for single consuming queue machines, a Turing-equivalent formalism that (as already explained) we introduce on purpose, into $\leq_{\mathsf{tin,tout}}$. The same encoding we use for $\leq_{\mathsf{tin,tout}}$ shows also undecidability of $\leq_{\mathsf{tin}}$, $\leq_{\mathsf{tout}}$ and $\leq$ (thus also providing an alternative proof for the undecidability of $\leq$ with respect to those by Bravetti et al. [10] and Lange and Yoshida [11]).

Limitations to the communication buffer. We limit the communication buffer capability, giving rise to the subtyping relations shown in the righthand part of Figure 1, by restricting the capability of $\leq$ to anticipate outputs: this is equivalent to putting an upper limit to communication buffers between two parties, a common fact in practice. In this context our decidability/undecidability results are the following ones.

- *Decidability of* k*-bounded Subtyping ($\leq_{\mathsf{k}}$), with* $\mathsf{k} \geq 0$*.* In k-bounded asynchronous subtyping we restrict the capability of $\leq$ to anticipate outputs: they can only be anticipated w.r.t. a number of inputs that is less or equal to k. We give and prove correct an algorithm for deciding whether any two session types are in a k-bounded subtyping relation. Notice that, in the case $\mathsf{k} = 0$ we obtain synchronous subtyping $\leq_{\mathsf{s}}$ [7]. Moreover, if we consider $\mathsf{k} = 1$ we have a notion of subtyping along the lines of that, we already mentioned, obtained by Lange and Yoshida [11] imposing restrictions on the communication behaviour.

- *Undecidability of Bounded Asynchronous Subtyping ($\leq_{\mathsf{bound}}$).* We say that a pair of session types is in bounded asynchronous subtyping relation if there exists a k such that such pair is in k-bounded subtyping relation. Bounded asynchronous subtyping relates types that do not unboundedly put messages in a buffer. For instance, the types

$$\mu\mathbf{t}. \oplus \{huge\_data : \oplus\{huge\_data : \&\{ack : \mathbf{t}\}\}\}$$

$$\mu\mathbf{t}.\&\{ack : \oplus\{huge\_data : \mathbf{t}\}\}$$

are related by asynchronous subtyping but not by *bounded* asynchronous subtyping: the augmented data production frequency of the subtype requires to store an unbounded amount of huge data. Since in practice buffers are bounded, this could have been an acceptable candidate notion for replacing standard asynchronous subtyping, however we prove that it is undecidable as well. We do this by showing undecidability of a property for queue machines: bounded non termination.

Outline. In Section 2 we present session types and definition of asynchronous subtyping $\leq$, the novel dual closed reformulation $\leq_{\mathsf{DC}}$, the fragments of $\leq$ shown in Figure 1 and a discussion about their properties. Section 3 presents all decidability results, notably for k-bounded subtyping ($\leq_{\mathsf{k}}$) and for subtyping over single-in types ($\leq_{\mathsf{sin}}$) and over single-out types ($\leq_{\mathsf{sout}}$). Section 4 presents all undecidability results, notably for bounded subtyping ($\leq_{\mathsf{bound}}$) and for subtyping without output covariance and input contravariance ($\leq_{\mathsf{tin,tout}}$). Section 5 discusses related work and Section 6 presents concluding remarks. Detailed proofs of theorems, lemmas and propositions can be found in the Appendix. We chose to put proof technicalities, that often include additional definitions and intermediate results, in the Appendix so not to disrupt the paper prose.

2. Session Types and Asynchronous Subtyping

We begin by formally introducing the various ingredients needed for our technical development.

We start with the formal syntax of binary session types. Similarly to Chen et al. [8] we do not use a dedicated construct for sending an output/receiving an input, we instead represent outputs and inputs directly inside choices. More precisely, we consider output selection $\oplus\{l_i : T_i\}_{i\in I}$, expressing an internal choice among outputs, and input branching $\&\{l_i : T_i\}_{i\in I}$, expressing an external choice among inputs. Each possible choice is labeled by a label l_i, taken from a global set of labels L, followed by a session continuation T_i. Labels in a branching/selection are assumed to be pairwise distinct.

Definition 2.1 (Session Types). *Given a set of labels L, ranged over by l, the syntax of binary session types is given by the following grammar:*

$$T \quad ::= \quad \oplus\{l_i : T_i\}_{i\in I} \quad | \quad \&\{l_i : T_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.T \quad | \quad \mathbf{t} \quad | \quad \mathbf{end}$$

A session type is single-out *if, for all of its subterms $\oplus\{l_i : T_i\}_{i\in I}$, $|I| = 1$. Similarly, a session type is* single-in *if, for all of its subterms $\&\{l_i : T_i\}_{i\in I}$, $|I| = 1$.*

In the sequel, we leave implicit the index set $i \in I$ in input branchings and output selections when it is already clear from the denotation of the types. Note also that we abstract from the type of the message that could be sent over the channel, since this is orthogonal to our theory. Types $\mu\mathbf{t}.T$ and $\mathbf{t}$ denote standard tail recursion for recursive types. We assume recursion to be guarded: in $\mu\mathbf{t}.T$, the recursion variable $\mathbf{t}$ occurs within the scope of an output or an input type. In the following, we will consider closed terms only, i.e., types with all recursion variables $\mathbf{t}$ occurring under the scope of a corresponding definition $\mu\mathbf{t}.T$. Type **end** denotes the type of a channel that can no longer be used.

In our development, it is crucial to count the number of times we need to unfold a recursion $\mu\mathbf{t}.T$. This is formalised by the following function:

Definition 2.2 (*n*-**unfolding).**

$$
\begin{array}{ll}
\mathsf{unfold}^0(T) = T & \mathsf{unfold}^1(\oplus\{l_i : T_i\}_{i\in I}) = \oplus\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mu\mathbf{t}.T) = T\{\mu\mathbf{t}.T/\mathbf{t}\} & \mathsf{unfold}^1(\&\{l_i : T_i\}_{i\in I}) = \&\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mathbf{end}) = \mathbf{end} & \mathsf{unfold}^n(T) = \mathsf{unfold}^1(\mathsf{unfold}^{n-1}(T))
\end{array}
$$

The definition of asynchronous subtyping uses the notion of input context, a type context consisting of a sequence of inputs preceding holes where types can be placed:

Definition 2.3 (Input Context). *An input context $\mathcal{A}$ is a session type with multiple holes defined by the syntax:* $\mathcal{A} \ ::= \quad [\,]^n \quad | \quad \&\{l_i : \mathcal{A}_i\}_{i\in I}$. *An input context $\mathcal{A}$ is well-formed whenever all its holes $[\,]^n$, with $n \in \mathbb{N}^+$, are consistently enumerated, i.e. there exists $m \geq 1$ such that $\mathcal{A}$ includes one and only one $[\,]^n$ for each $n \leq m$. Given a well-formed input context $\mathcal{A}$ with holes indexed over $\{1, \dots, m\}$ and types $T_1, \dots, T_m$, we use $\mathcal{A}[T_k]^{k\in\{1,\dots,m\}}$ to denote the type obtained by filling each hole k in $\mathcal{A}$ with the corresponding term T_k.*

From now on, whenever using input contexts we will assume them to be well-formed, unless otherwise specified.

For example, consider the input context

$$\mathcal{A} = \&\{l_1 : [\,]^1,\ l_2 : [\,]^2\}$$

we have:

$$\mathcal{A}[\oplus\{l : T_i\}]^{i\in\{1,2\}} = \&\big\{l_1 : \oplus\{l : T_1\}, l_2 : \oplus\{l : T_2\}\big\}$$

We start by considering the standard notion of asynchronous subtyping $\leq$ given by Chen et al. [8]. We choose it because of its orphan message free property that is commonly recognized to be convenient: only subtypes are allowed that do not cause incoming messages to remain "orphan" (because they are never consumed from the communication buffer). In the definition of asynchronous subtyping given by Chen et al., orphan message freedom causes a specific dedicated constraint to be included (which is, e.g., instead not present in the asynchronous subtyping definition by Mostrous and Yoshida [12]). We now formally present the asynchronous subtyping relation $\leq$, rephrased w.r.t. that of Chen et al. [8] in a technical format that is convenient for showing our results, which follows a coinductive simulation-like definition.

Definition 2.4 (Asynchronous Subtyping, $\leq$). *$\mathcal{R}$ is an asynchronous subtyping relation if $(T, S) \in \mathcal{R}$ implies that:*

1. *if $T = \mathbf{end}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \mathbf{end}$;*
2. *if $T = \oplus\{l_i : T_i\}_{i \in I}$ then $\exists n \geq 0, \mathcal{A}$ such that*
   - $\mathsf{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in \{1,\ldots,m\}}$,
   - $\forall k \in \{1, \ldots, m\}.\, I \subseteq J_k$,
   - *$\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k \in \{1,\ldots,m\}}) \in \mathcal{R}$ and*
   - *if $\mathcal{A} \neq [\,]^1$ then $\forall i \in I.\ \& \in T_i$ (no orphan message constraint);*
3. *if $T = \&\{l_i : T_i\}_{i \in I}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \&\{l_j : S_j\}_{j \in J}$, $J \subseteq I$ and $\forall j \in J.(T_j, S_j) \in \mathcal{R}$;*
4. *if $T = \mu\mathbf{t}.T'$ then $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$.*

where with "$\& \in T_i$" we mean that T_i contains at least an input branching. T is an asynchronous subtype of S, written $T \leq S$, if there is an asynchronous subtyping relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$.

Intuitively, two types T and S are related by $\leq$, whenever S is able to simulate T, but with a few twists: type S is allowed to anticipate outputs nested in its syntax tree (asynchrony); and, output and input types enjoy covariance and contravariance, respectively. Moreover, the above definition includes the no orphan message constraint [8], namely: we allow the supertype inputs to be delayed only if also the subtype contains some input.

A *synchronous* subtyping relation $\leq_{\mathsf{s}}$ like that of Gay and Hole [7] is obtained by requiring that, in item 2. of the above Definition 2.4, it always holds $\mathcal{A} = [\,]^1$.

Example 2.5. *Consider $T = \mu\mathbf{t}. \oplus \{l : \&\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}\}$ and $S = \mu\mathbf{t}.\&\{l_1 : \&\{l_2 : \oplus\{l : \mathbf{t}\}\}\}$. We have $T \leq S$ because the following is an asynchronous subtyping relation:*

$$
\begin{array}{ll}
\{ & (T, S)\ ,\ (\oplus\{l : \&\{l_1 : T, l_2 : T\}\}, S)\ ,\ (\&\{l_1 : T, l_2 : T\}, \&\{l_1 : \&\{l_2 : S\}\})\ , \\
& (T, \&\{l_2 : S\})\ ,\ (\oplus\{l : \&\{l_1 : T, l_2 : T\}\}, \&\{l_2 : S\})\ , \\
& \quad (\&\{l_1 : T, l_2 : T\}, \&\{l_2 : \&\{l_1 : \&\{l_2 : S\}\}\}), \\
& (T, \&\{l_1 : \&\{l_2 : S\}\})\ ,\ (\oplus\{l : \&\{l_1 : T, l_2 : T\}\}, \&\{l_1 : \&\{l_2 : S\}\})\ , \\
& \quad (\&\{l_1 : T, l_2 : T\}, \&\{l_1 : \&\{l_2 : \&\{l_1 : \&\{l_2 : S\}\}\}\})\ , \\
& (T, \&\{l_2 : \&\{l_1 : \&\{l_2 : S\}\}\})\ ,\ \ldots \qquad \}
\end{array}
$$

Note that the relation contains infinitely many pairs that differ in the sequence of inputs, alternatively labeled with l_1 and l_2, that are accumulated at the beginning of the r.h.s. type.

We now introduce an alternative way of defining orphan message free asynchronous subtyping, which is more elegant/concise: it obtains the orphan message freedom property by requiring *closure under duality* of the type relation being defined instead of making use of an explicit orphan message free constraint as in Definition 2.4.

For session types, we define the usual notion of duality: given a session type T, its dual $\overline{T}$ is defined as: $\overline{\oplus\{l_i : T_i\}_{i\in I}} = \&\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\&\{l_i : T_i\}_{i\in I}} = \oplus\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{\mathbf{t}} = \mathbf{t}$, and $\overline{\mu\mathbf{t}.T} = \mu\mathbf{t}.\overline{T}$. In the sequel, we say that a relation $\mathcal{R}$ on session types is *dual closed* if $(S,T) \in \mathcal{R}$ implies $(\overline{T}, \overline{S}) \in \mathcal{R}$.

Definition 2.6 (Asynchronous Dual Closed Subtyping, $\leq_{\mathsf{DC}}$). *$\mathcal{R}$ is an asynchronous dual closed subtyping relation whenever it is* dual closed *and $(T,S) \in \mathcal{R}$ implies* 1., 3., *and* 4. *of Definition 2.4, plus a modified version of* 2. *where the last constraint (the no orphan message constraint) is removed.*

T is an asynchronous dual closed subtype of S, written $T \leq_{\mathsf{DC}} S$, if there is an asynchronous dual closed subtyping relation $\mathcal{R}$ such that $(T,S) \in \mathcal{R}$.

We observe that our definition is formally different from the ones found in literature. In particular, with respect to that by Mostrous and Yoshida [12], it additionally requires the subtyping relation to be dual closed. Below, we state that the dual closeness requirement is equivalent to imposing the orphan message free constraint, i.e. the last item of condition 2 in Definition 2.4 (both guarantee orphan-message freedom):

Theorem 2.7. *Given two session types T and S, we have $T \leq S$ if and only if $T \leq_{\mathsf{DC}} S$.*

2.1. Subtyping Relation Restrictions

As already discussed in the Introduction, we focus on two kinds of restrictions of asynchronous subtyping: limitations to the branching/selection structure and to the communication buffer, giving rise to the numerous relations shown in the lefthand part and righthand part of Figure 1, respectively.

We now define fragments of $\leq$ obtained by posing limitations to the branching/selection structure.

Definition 2.8. *Restrictions of the asynchronous subtyping relation are denoted by adding subscripts to the $\leq$ notation, with the following meaning:*

- *whenever we add subscript* sout *(sin, resp.) we additionally require in Definition 2.4 both T and S to be single-out (single-in, resp.),*
- *whenever we add subscript* tout *(tin, resp.) we additionally require in Definition 2.4 $I = J_k$ in point 2. ($I = J$ in point 3., resp.).*

The latter means that each output selection (input branching, resp.) performed by the subtype is matched by an output selection (input branching, resp.) performed by the supertype with the exactly the same *total* set of labels, i.e. output covariance (input contravariance, resp.) is not admitted.

Notice that, while it holds that $\leq = \leq_{\mathsf{DC}}$, not all fragments of $\leq$ are asynchronous dual closed subtyping relations. For instance this does not hold for $\leq_{\mathsf{sin}}$, $\leq_{\mathsf{sout}}$, $\leq_{\mathsf{tin}}$ and $\leq_{\mathsf{tout}}$, which perform a limitation, but not its "dual" one. It holds, instead, for the following two relations that we will show to be in the boundary between decidability and undecidability.

Proposition 2.9. *The $\leq_{\mathsf{tin},\mathsf{tout}}$ relation and the $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ relation are asynchronous dual closed subtyping relations.*

Dual closeness of the $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ relation is a direct consequence of the fact that $T \leq_{\mathsf{sin}} S$ if and only if $\overline{S} \leq_{\mathsf{sout}} \overline{T}$, which obviously derives from dual closeness of $\leq$ and from the dual of a single-in type being a single-out type and vice-versa. This fact, together with the following proposition, will be used to infer decidability of $\leq_{\mathsf{sin}}$ and $\leq_{\mathsf{sin},\mathsf{tout}}$ relations from that of $\leq_{\mathsf{sout}}$ and $\leq_{\mathsf{tin},\mathsf{sout}}$, respectively.

Proposition 2.10. *The $\leq_{\mathsf{sin},\mathsf{tout}}$ and $\leq_{\mathsf{tin},\mathsf{sout}}$ relations are such that: $T \leq_{\mathsf{sin},\mathsf{tout}} S$ if and only if $\overline{S} \leq_{\mathsf{tin},\mathsf{sout}} \overline{T}$.*

We now consider variants of $\leq$ obtained by posing limitations to the communication buffer.

We can define a variant decidable relation by putting an upper-bound to the messages that can be buffered. Technically speaking, when an output in the r.h.s. is anticipated during the subtyping simulation, we impose a bound to the number of inputs that are in front of such output.

We say that an input context $\mathcal{A}$ is k-*bounded* if the maximal number of nested inputs in $\mathcal{A}$ is less or equal to k.

Definition 2.11 (k-bounded Asynchronous Subtyping, with $\mathsf{k} \geq 0$). *The k-bounded asynchronous subtyping $\leq_{\mathsf{k}}$ is defined as in Definition 2.4, with the only difference that the input context $\mathcal{A}$ in item 2. is required to be k-bounded.*

Notice that the case $\mathsf{k} = 0$ yields synchronous subtyping: since $\mathcal{A} = [\,]^1$ is the only 0-bounded input context, we obviously have $\leq_0 = \leq_{\mathsf{s}}$.

Lange and Yoshida [11] show the decidability of asynchronous subtyping for a subclass of session types, called *alternating*, that in our setting corresponds to impose that every output in a subtype is immediately followed by an input, while every input in a supertype is followed by an output. For instance, this property is satisfied by the following pair of types:

$$T = \mu\mathbf{t}. \oplus \{l_2 : \&\{l_1 : \mathbf{t}\}\} \qquad S = \mu\mathbf{t}.\&\{l_1 : \oplus\{l_2 : \mathbf{t}\}\}$$

It is not difficult to see that $T \leq_1 S$. The key point of the proof of decidability of asynchronous subtyping for alternating session types by Lange and Yoshida is the observation that if T and S are alternating, then $T \leq S$ if and only if $T \leq_1 S$.

As we explained in the Introduction, we also consider the more generic notion of *bounded* asynchronous subtyping. This relation is in our opinion of interest because it reflects real cases in which it is possible to assume bounded buffers, without an a priory knowledge of the actual bound.

Definition 2.12 (Bounded Asynchronous Subtyping, $\leq_{\mathsf{bound}}$). *We say that T is a* bounded asynchronous subtype *of S, written $T \leq_{\mathsf{bound}} S$, if there exists k such that $T \leq_{\mathsf{k}} S$.*

3. Decidability Results

We now present decidability results for k-bounded asynchronous subtyping and asynchronous subtyping for single-out/single-in session types.

3.1. A Subtyping Procedure

We start by giving a procedure (an algorithm that does not necessarily terminate) for the general subtyping relation, which is known to be undecidable [10, 11]. Such a procedure is inspired by the one proposed by Mostrous et al. [14] for asynchronous subtyping in multiparty session types. In order to do so, we introduce two functions on the syntax of types. The function $\mathsf{outDepth}$ calculates how many unfolding are necessary for bringing an output outside a recursion. If that is not possible, the function is undefined (denoted by $\bot$).

Definition 3.1 (outDepth). *The partial function* $\mathsf{outDepth}(T)$ *is inductively defined as follows:*

$$\mathsf{outDepth}(\oplus\{l_i : T_i\}_{i\in I}) = 0 \quad \mathsf{outDepth}(\&\{l_i : T_i\}_{i\in I}) = \mathsf{max}\{\mathsf{outDepth}(T_i) \mid i \in I\}$$

$$\mathsf{outDepth}(\mu\mathbf{t}.T) = 1 + \mathsf{outDepth}(T\{\mathbf{end}/\mathbf{t}\}) \quad \mathsf{outDepth}(\mathbf{end}) = \bot$$

where $\mathsf{max}\{\mathsf{outDepth}(T_i) \mid i \in I\} = \bot$*, if* $\mathsf{outDepth}(T_i) = \bot$ *for some* $i \in I$*; similarly,* $1 + \bot = \bot$.

As an example of application of $\mathsf{outDepth}$ consider, for any T_1 and T_2, $\mathsf{outDepth}(\oplus\{l_1 : T_1,\ l_2 : T_2\}) = 0$. On the other hand, consider the type $T_{\mathsf{ex}} = \&\Big\{l_1 : \mu\mathbf{t}.\oplus\big\{l_2 : T_1\big\},\ l_3 : \mu\mathbf{t}.\&\big\{l_4 : \mu\mathbf{t}'.\oplus\{l_5 : T_2\}\big\}\Big\}$: clearly, $\mathsf{outDepth}\big(T_{\mathsf{ex}}\big) = 2$. We then define $\mathsf{outUnf}()$, a variant of the unfolding function given in Definition 2.2, which unfolds only where it is necessary, in order to reach an output:

Definition 3.2 (outUnf). *The output unfolding* $\mathsf{outUnf}(T)$ *is a partial function defined whenever* $\mathsf{outDepth}(T)$ *is defined. Given* $\mathsf{outDepth}(T) = n$*,* $\mathsf{outUnf}(T)$ *is computed using the same inductive rules of* $\mathsf{unfold}^n(T)$*, excluding the rule for* $\oplus\{l_i : T_i\}_{i\in I}$ *that, instead of recursively unfolding* T_i*, returns the same term* $\oplus\{l_i : T_i\}_{i\in I}$*.*

The function above differs from unfold^n: for example, $\mathsf{unfold}^2\big(T_{ex}\big)$ would unfold twice both subterms $\mu\mathbf{t}.\oplus\{l_2 : T_1\}$ and $\mu\mathbf{t}.\&\big\{l_4 : \mu\mathbf{t}'.\oplus\{l_5 : T_2\}\big\}$. On the other hand, applying outUnf to the same term would unfold once the term reached with l_1 and twice the one reached with l_3.

In the subtyping procedure defined below we make use of outUnf in order to have that recursive definitions under the scope of an output are never unfolded. This guarantees that during the execution of the procedure, even if the set of reached terms could be unbounded, all the subterms starting with an output are taken from a bounded set of terms. This is important to guarantee termination[3] of the algorithm that we will define in Section 3.3 as an extension of the procedure described below.

[3]Technically speaking, this property of the unfolding is used in the proof of Theorem 3.11.

Subtyping Procedure. An environment Σ is a set containing pairs (T, S), where T and S are types. Judgements are triples of the form $\Sigma \vdash T \leq_{\mathsf{a}} S$ which intuitively read as "in order to succeed, the procedure must check whether T is a subtype of S, provided that pairs in Σ have already been visited". Our *subtyping procedure*, applied to the types T and S, consists of deriving the state space of our judgments using the rules in Figure 2 bottom-up starting from the initial judgement $\emptyset \vdash T \leq_{\mathsf{a}} S$. More precisely, we use the transition relation $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ to indicate that if $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusions of one of the rules in Figure 2, then $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is produced by the corresponding premises. The procedure explores the reachable judgements according to this transition relation. We give highest priority to rule Asmp, thus ensuring that at most one rule is applicable.[4] The idea behind Σ is to avoid cycles when dealing with recursive types. Rules RecR_1 and RecR_2 deal with the case in which we need to unfold recursion in the type on the right-hand side. If the type on the left-hand side is not an output then the procedure simply adds the current pair to Σ and continues. On the other hand, if an output must be found, we apply RecR_2 which checks whether such output is available. Rule Out allows nested outputs to be anticipated (when not under recursion) and condition $\left(\mathcal{A} \neq [\,]^1\right) \Rightarrow \forall i \in I. \& \in T_i$ makes sure there are no orphan messages. The remaining rules are self-explanatory. $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is the reflexive and transitive closure of the transition relation among judgements. We write $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow_{\mathsf{ok}}$ if the judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusion of one of the axioms Asmp or End, and $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow_{\mathsf{err}}$ to mean that no rule can be applied to $\Sigma \vdash T \leq_{\mathsf{a}} S$. Due to input branching and output selection, the rules In and Out could generate branching also in the state space to be explored by the procedure. Namely, given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$, there are several subsequent judgements $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ sucht that $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{a}} S'$. The procedure could (i) successfully terminate because all the explored branches reach a successful judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{ok}}$, (ii) terminate with an error in case at least one judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ is reached, or (iii) diverge because no branch terminates with an error and at least one branch never reaches a succesful judgement.

Example 3.3. *Consider* $T = \mu\mathbf{t}. \oplus \big\{l_1 : \&\{l_2 : \mathbf{t}\}\big\}$ *and* $S = \mu\mathbf{t}. \oplus \Big\{l_1 : \&\big\{l_2 : \&\{l_2 : \mathbf{t}\}\big\}\Big\}$. *Clearly, the two types T and S are related by asynchronous subtyping, i.e. $T \leq S$. However, the subtyping procedure on $\emptyset \vdash T \leq_{\mathsf{a}} S$ does not terminate:*

[4] The priority of Asmp is sufficient because all the other rules are alternative, i.e., given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ there are no two rules different from Asmp that can be both applied.

$$\frac{\begin{array}{c}(\mathcal{A} \neq [\,]^1) \Rightarrow \forall i \in I.\& \in T_i \\ \forall n.I \subseteq J_n \qquad \forall i \in I\,.\,\Sigma \vdash T_i \leq_{\mathsf{a}} \mathcal{A}[S_{ni}]^n\end{array}}{\Sigma \vdash \oplus\{l_i \colon T_i\}_{i\in I} \leq_{\mathsf{a}} \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j \in J_n}]^n}\ \mathsf{Out}$$

$$\frac{J \subseteq I \quad \forall j \in J\,.\,\Sigma \vdash T_j \leq_{\mathsf{a}} S_j}{\Sigma \vdash \&\{l_i \colon T_i\}_{i\in I} \leq_{\mathsf{a}} \&\{l_j \colon S_j\}_{j\in J}}\ \mathsf{In} \qquad \frac{}{\Sigma \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}}\ \mathsf{End}$$

$$\frac{}{\Sigma, (T,S) \vdash T \leq_{\mathsf{a}} S}\ \mathsf{Asmp} \qquad \frac{\Sigma, (\mu\mathbf{t}.T, S) \vdash T\{\mu\mathbf{t}.T/\mathbf{t}\} \leq_{\mathsf{a}} S}{\Sigma \vdash \mu\mathbf{t}.T \leq_{\mathsf{a}} S}\ \mathsf{RecL}$$

$$\frac{T = \mathbf{end} \lor T = \&\{l_i \colon T_i\}_{i\in I} \quad \Sigma, (T, \mu\mathbf{t}.S) \vdash T \leq_{\mathsf{a}} S\{\mu\mathbf{t}.S/\mathbf{t}\}}{\Sigma \vdash T \leq_{\mathsf{a}} \mu\mathbf{t}.S}\ \mathsf{RecR}_1$$

$$\frac{\mathsf{outDepth}(S) \geq 1 \quad \Sigma, (\oplus\{l_i \colon T_i\}_{i\in I}, S) \vdash \oplus\{l_i \colon T_i\}_{i\in I} \leq_{\mathsf{a}} \mathsf{outUnf}(S)}{\Sigma \vdash \oplus\{l_i \colon T_i\}_{i\in I} \leq_{\mathsf{a}} S}\ \mathsf{RecR}_2$$

Figure 2: A Procedure for Checking Subtyping

$$\begin{array}{l}
\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow \\
\big\{(T,S)\big\} \vdash \oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\} \leq_{\mathsf{a}} S \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S)\big\} \\
\qquad \vdash \oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\} \leq_{\mathsf{a}} \oplus\Big\{l_1 \colon \&\big\{l_2 \colon \&\{l_2 \colon S\}\big\}\Big\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S)\big\} \vdash \&\{l_2 \colon T\} \leq_{\mathsf{a}} \&\big\{l_2 \colon \&\{l_2 \colon S\}\big\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S)\big\} \vdash T \leq_{\mathsf{a}} \&\{l_2 \colon S\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S), (T, \&\{l_2 \colon S\})\big\} \vdash \oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\} \leq_{\mathsf{a}} \&\{l_2 \colon S\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S), (T, \&\{l_2 \colon S\}), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, \&\{l_2 \colon S\})\big\} \\
\qquad \vdash \oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\} \leq_{\mathsf{a}} \&\Big\{l_2 \colon \oplus\big\{l_1 \colon \&\big\{l_2 \colon \&\{l_2 \colon S\}\big\}\big\}\Big\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, S), (T, \&\{l_2 \colon S\}), (\oplus\big\{l_1 \colon \&\{l_2 \colon T\}\big\}, \&\{l_2 \colon S\})\big\} \\
\qquad \vdash \&\{l_2 \colon T\} \leq_{\mathsf{a}} \&\Big\{l_2 \colon \&\big\{l_2 \colon \&\{l_2 \colon S\}\big\}\Big\} \rightarrow \\
\ldots
\end{array}$$

Notice that the last step above is obtained by application of the rule Out *by considering the input context* $\mathcal{A} = \&\{l_2 : [\,]\}$.

The example above shows that the procedure could diverge; the next result proves that this can happen only if the checked types are in subtyping relation. More precisely, types T and S are *not* in subtyping relation if and only if the procedure on $\emptyset \vdash T \leq_{\mathsf{a}} S$ terminates with an error; formally

Proposition 3.4. *Given the types T and S, we have that there exist Σ', T', S' such that $T \not\leq S$ if and only if $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$.*

This means that: if $T \not\leq S$ then the procedure on $\emptyset \vdash T \leq_{\mathsf{a}} S$ surely terminates with an error; if, instead, $T \leq S$ then the procedure terminates successfully or diverges.

3.2. k*-bounded Asynchronous Subtyping*

In the previous subsection we have shown that the standard subtyping procedure does not terminate in general. In order to guarantee termination, Lange and Yoshida [11] have considered limitations to the communication buffer, like half-duplex (in this case asynchronous and synchronous subtyping coincides) or alternating protocols (in this case the buffer will store at most one message). We now prove a more general decidability result. We show that, for every k, we can define an algorithm for the notion of k-bounded asynchronous subtyping introduced in Section 2.1, building on the subtyping procedure defined previously.

We consider an algorithm, that we denote with $\leq_{\mathsf{a}}^{\mathsf{k}}$, obtained from the above procedure for $\leq_{\mathsf{a}}$ simply by imposing that the input context $\mathcal{A}$, used in rule Out in Figure 2, is always k-bounded. Then, the following result holds:

Theorem 3.5. *The algorithm for $\leq_{\mathsf{a}}^{\mathsf{k}}$ always terminates and, given the types T and S, there exist Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}}^{\mathsf{k}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ if and only if $T \not\leq_{\mathsf{k}} S$.*

3.3. Asynchronous Subtyping for Single-Out or Single-In Types

In Example 3.3 we have seen that, if we consider the terms $T = \mu\mathbf{t}.\oplus\{l_1 : \&\{l_2 : \mathbf{t}\}\}$ and $S = \mu\mathbf{t}.\oplus\Big\{l_1 : \&\{l_2 : \&\{l_2 : \mathbf{t}\}\}\Big\}$, the subtyping procedure in Figure 2 applied to $\emptyset \vdash T \leq_{\mathsf{a}} S$ does not terminate. The problem is that the termination rule Asmp cannot be applied because the term on the r.h.s. (i.e. the supertype) generates always new terms in the form $\&\Big\{l_2 : \&\{l_2 : \ldots \&\{l_2 : S\} \ldots\}\Big\}$.

Notice that, in this particular example, these infinitely many distinct terms are obtained by adding single inputs (i.e. single-choice input branchings) in front of the term in the r.h.s.: we call this *linear input accumulation*.

For simple cases like this one, solutions have been proposed by Lange and Yoshida [11] and Bravetti et al. [10]. The idea is to extend the subtyping procedure in Figure 2 with additional termination rules able to detect when it is no longer necessary to continue because it entered a deterministic loop (where the only possible future behavior of the procedure is to repeat indefinitely the same linear input accumulation). This approach holds only under two assumptions, both satisfied by the subtyping relations considered in Lange and Yoshida [11] and Bravetti et al. [10]: while checking subtyping output selections in the l.h.s. (i.e. the subtype) are always single-choice and the same holds for input branchings in the r.h.s. (i.e. the supertype). This implies that there is a linear input accumulation, which is the repetition of a specific sequence of input labels. The combination of these two assumptions guarantees that the subtyping procedure proceeds deterministically: this makes it possible to detect whether it enters a loop because the unique kind of loops are the deterministic ones.

In this section we show that it is possible to relax at least one of these two assumptions: either deal with the case in which the input accumulation is not linear, or deal with the case in which output selections in the l.h.s. are not single-choice. More precisely, the two cases that we consider are the following ones: subtyping between single-out session types (where input branchings in the r.h.s. are not constrained to be single-choice as in previous approaches) and subtyping between single-in session types (where output selections in the l.h.s. are not constrained to be single-choice as in previous approaches), i.e. the two relations $\leq_{\mathsf{sout}}$ and $\leq_{\mathsf{sin}}$, respectively, that we introduced in Section 2.1. The idea is to find an algorithm for one of the two cases and apply it also to the other one by exploiting type duality.

In the single-in case we surely have linear input accumulation but the subtyping procedure is no longer deterministic due to non-single output selections in the l.h.s. that have multiple possible continuations. This causes the approach proposed in Lange and Yoshida [11] and Bravetti et al. [10] to fail because now the procedure can incur in nondeterministic loops (so it is not guaranteed to repeat indefinitely the accumulation behavior detected by the additional termination rule they consider). On the other hand, in the single-out case we loose the linear input accumulation but we do not have output selections to cause the problematic nondeterminism discussed above.

The latter advantage led us to opt for the single-out case, which we were able to manage by adopting a totally new approach where the input accumulation is represented *in the form of a tree* (thus accounting for all possible alternative accumulated input behaviors at the same time).

We start with an example of subtyping between single-out types that cannot be managed with the approach in Lange and Yoshida [11] and Bravetti et al. [10] because there is non-linear input accumulation.

Example 3.6. *Consider* $T = \mu\mathbf{t}.\oplus\big\{l_1 : \&\{l_2 : \mathbf{t}, l_3 : \mathbf{t}\}\big\}$ *and* $S = \mu\mathbf{t}.\oplus\Big\{l_1 : \&\big\{l_2 : \&\{l_2 : \mathbf{t}\}, l_3 : \mathbf{t}\big\}\Big\}$*. We now comment the application of the subtyping procedure on* $\emptyset \vdash T \leq_{\mathsf{a}} S$*.*

$$\begin{array}{l}
\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow \\
\big\{(T,S)\big\} \vdash \oplus\big\{l_1 : \&\{l_2 : T, l_3 : T\}\big\} \leq_{\mathsf{a}} S \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 : \&\{l_2 : T, l_3 : T\}\big\}, S)\big\} \vdash \\
\qquad \oplus\big\{l_1 : \&\{l_2 : T, l_3 : T\}\big\} \leq_{\mathsf{a}} \oplus\Big\{l_1 : \&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}\Big\} \rightarrow \\
\big\{(T,S), (\oplus\big\{l_1 : \&\{l_2 : T, l_3 : T\}\big\}, S)\big\} \vdash \\
\qquad \&\{l_2 : T, l_3 : T\} \leq_{\mathsf{a}} \&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}
\end{array}$$

At this point, the subtyping procedure has two continuations, one for the label l_2 *and one for the label* l_3*. In case of label* l_3 *we reach the judgement:*

$$\big\{(T,S), (\oplus\big\{l_1 : \&\{l_2 : T, l_3 : T\}\big\}, S)\big\} \vdash T \leq_{\mathsf{a}} S$$

on which the termination rule Asmp *can be applied. In case of label* l_2 *we have:*

$$
\begin{array}{l}
\big\{(T,S),(\oplus\big\{l_1\colon \&\{l_2:T,l_3:T\}\big\},S)\big\} \vdash T \leq_{\mathsf{a}} \&\{l_2\colon S\} \rightarrow\\
\big\{(T,S),(\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},S),(T,\&\{l_2\colon S\})\big\}\\
\quad\quad \vdash \oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\} \leq_{\mathsf{a}} \&\{l_2\colon S\} \rightarrow\\
\big\{(T,S),(\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},S),(T,\&\{l_2\colon S\}),\\
\quad (\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},\&\{l_2\colon S\})\big\}\\
\quad\quad \vdash \oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\} \leq_{\mathsf{a}} \&\Big\{l_2\colon \oplus\big\{l_1\colon \&\big\{l_2\colon \&\{l_2\colon S\},l_3\colon S\big\}\big\}\Big\} \rightarrow\\
\big\{(T,S),(\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},S),(T,\&\{l_2\colon S\}),\\
\quad (\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},\&\{l_2\colon S\})\big\}\\
\quad\quad \vdash \&\{l_2\colon T,l_3\colon T\} \leq_{\mathsf{a}} \&\Big\{l_2\colon \&\big\{l_2\colon \&\{l_2\colon S\},l_3\colon S\big\}\Big\} \rightarrow\\
\big\{(T,S),(\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},S),(T,\&\{l_2\colon S\}),\\
\quad (\oplus\big\{l_1\colon \&\{l_2\colon T,l_3\colon T\}\big\},\&\{l_2\colon S\})\big\}\\
\quad\quad \vdash T \leq_{\mathsf{a}} \&\big\{l_2\colon \&\{l_2\colon S\},l_3\colon S\big\} \rightarrow\\
\ldots
\end{array}
$$

Notice that in the last judgement, the r.h.s. has a non-linear input accumulation starting with an input choice on two labels l_2 *and* l_3*.*

3.3.1. Asynchronous Subtyping for Single-Out Types

We now present our novel approach to asynchronous subtyping that can be applied to single-out types, hence also to the types in the above Example 3.6, that will be used as a running example in this section. As anticipated, the main novelty is the ability to deal with non-linear input accumulation by representing it as a tree. We need to be able to extract the leafs from these trees: this is done by the *leaf set* function defined as follows.

Definition 3.7 (Leaf Set). *Given a session type* S*, we write* $\mathsf{noIn}(S)$ *if* S *is not of the form* $\&\{l_i\colon S_i\}_{i\in I}$*. Given a session type* T*, we define*

$$\mathsf{leafSet}(T)=\{T_1,\ldots,T_n \mid \mathsf{noIn}(T_i) \textit{ and } \exists \textit{ input context } \mathcal{A} \textit{ s.t. } T=\mathcal{A}[T_k]^{k\in\{1\ldots n\}}\}$$

The leaf set of a session type T is the set of subterms different from inputs that are reachable from its root through a path of inputs. For example, the leaf set of the term $\&\{l_1 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}, l_3 : \&\{l_4 : \oplus\{l_2 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}\}\}\}$ is $\{\mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}, \oplus\{l_2 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}\}$. If we consider the r.h.s. term in the last judgement in Example 3.6, we have that $\mathsf{leafSet}(\&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}) = \{S\}$.

During the check of subtyping, according to Figure 2 (rule Out), when a term in the r.h.s. having input accumulation has to mimic an output in front of the l.h.s., such output must be present in front of all the leafs of the tree. In this case, the checking continues by anticipating the output from all the leafs. The following auxiliary function *output anticipation* indicates the way a term changes after having anticipated a sequence of outputs. Notice that in the definition we make use of the assumption on single-out session types, by considering single-choice output selections.

Definition 3.8 (Output Anticipation). *Partial function* $\mathsf{antOut}(T, l_{i_1} \cdots l_{i_n})$, *with* T *single-out session type and* $l_{i_1} \cdots l_{i_n}$ *sequence of labels, is inductively defined as follows:*

$$\mathsf{antOut}(T, l_{i_1} \cdots l_{i_n}) = \begin{cases} T & \text{if } n = 0 \\ \mathcal{A}[T_k]^k & \text{if } \mathsf{outUnf}(\mathsf{antOut}(T, l_{i_1} \cdots l_{i_{n-1}})) = \mathcal{A}[\oplus\{l_{i_n} \colon T_k\}]^k \end{cases}$$

We say that T *can infinitely anticipate outputs, written* $\mathsf{antOutInf}(T)$*, if there exists an infinite sequence of labels* $l_{i_1} \cdots l_{i_j} \cdots$ *such that* $\mathsf{antOut}(T, l_{i_1} \cdots l_{i_n})$ *is defined for every* n*.*

The function $\mathsf{antOut}(T, \tilde{l})$ anticipates all outputs in the sequence $\tilde{l}$. For example, the function applied to $\&\{l : \mu\mathbf{t}. \oplus \{l_1 : \oplus\{l_2 : \mathbf{t}\}\},\ l' : \oplus\{l_1 : \mu\mathbf{t}. \oplus \{l_2 : \oplus\{l_1 : \mathbf{t}\}\}\}\}$ and the sequence (l_1, l_2) would return the same term, while it would be undefined with the sequence (l_1, l_1). If we go back to our running Example 3.6, we have that $\mathsf{antOut}(S, l_1) = \&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}$. Moreover, we have that $\mathsf{antOutInf}(S)$ holds because the label l_1 can be infinitely anticipated.

The definition of $\mathsf{antOutInf}(T)$ is not algorithmic in that it quantifies on every possible natural number n. Nevertheless, as we show below, it can be decided by checking whether for every session type obtained from T by means of output anticipations, all the terms populating its leaf set can anticipate the same output label. Although such process may generate infinitely many session types, the terms populating the leaf sets are finite and are over-approximated by the function $\mathsf{reach}(T)$, which always returns a finite set and is defined as:

Definition 3.9 (Reachable Types). *Given a single-out session type* T*,* $\mathsf{reach}(T)$ *is the minimal set of session types such that:*

1. $T \in \mathsf{reach}(T)$*;*
2. $\&\{l_i \colon T_i\}_{i \in I} \in \mathsf{reach}(T)$ *implies* $T_i \in \mathsf{reach}(T)$ *for every* $i \in I$*;*
3. $\mu\mathbf{t}.T' \in \mathsf{reach}(T)$ *implies* $T'\{\mu\mathbf{t}.T'/\mathbf{t}\} \in \mathsf{reach}(T)$*;*
4. $\oplus\{l \colon T'\} \in \mathsf{reach}(T)$ *implies* $T' \in \mathsf{reach}(T)$*.*

Notice that $\mathsf{reach}(T)$ is populated by those session types obtained by consuming in sequence the initial inputs and outputs, and by unfolding recursion only when it is at the top level. As an example, consider the session type S of the Example 3.6. We have

$$\mathsf{reach}(S) = \Big\{S, \oplus\big\{l_1 \colon \&\big\{l_2 \colon \&\{l_2 \colon S\}, l_3 \colon S\big\}\big\}, \&\big\{l_2 \colon \&\{l_2 \colon S\}, l_3 \colon S\big\}, \&\{l_2 \colon S\}\Big\}$$

For every type T, we have that the terms in $\mathsf{reach}(T)$ are finite; in fact, during the generation of such terms, eventually the term **end** or a term already considered is reached. The latter occurs after consumption of all the inputs and outputs in front of a recursion variable already unfolded.

Proposition 3.10. *Given a single-out session type* T*,* $\mathsf{reach}(T)$ *is finite and it is decidable whether* $\mathsf{antOutInf}(T)$*.*

Subtyping algorithm for single-out types. We are now ready to present the new termination condition that once added to the subtyping procedure in Figure 2 makes it a valid algorithm for checking subtyping for single-out types. The termination condition is defined as an additional rule, named $\mathsf{Asmp2}$, that complements the already defined Asmp rule by detecting those cases in which the subtyping procedure in Figure 2 does not terminate.

The new rule is defined parametrically on the session type Z, which is the type on the right-hand side of the initial pair of types to be checked (i.e. the algorithm is intended to check $V \leq Z$, for some type Z). We start from the initial judgement $\emptyset \vdash V \leq_{\mathsf{t}} Z$ and then apply from bottom to top the rules in Figure 2, where $\leq_{\mathsf{a}}$ is replaced by $\leq_{\mathsf{t}}$, plus the following additional rule:

$$\frac{\begin{array}{c} S \in \mathsf{reach}(Z) \qquad \mathsf{antOutInf}(S) \qquad |\gamma| < |\beta| \\ \mathsf{leafSet}(\mathsf{antOut}(S,\gamma)) = \mathsf{leafSet}(\mathsf{antOut}(S,\beta)) \end{array}}{\Sigma, (T, \mathsf{antOut}(S,\gamma)) \vdash T \leq_{\mathsf{t}} \mathsf{antOut}(S,\beta)} \;\mathsf{Asmp2}$$

We first observe that this termination rule can be applied to the last judgement of our running Example 3.6. We have already seen that $S \in \mathsf{reach}(S)$, $\mathsf{antOutInf}(S)$ holds, $\mathsf{antOut}(S, l_1) = \&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}$ and that $\mathsf{leafSet}(\&\big\{l_2 : \&\{l_2 : S\}, l_3 : S\big\}) = \{S\}$. We now observe that $\mathsf{antOut}(S, \varepsilon) = S$ and $\mathsf{leafSet}(S) = \{S\}$, hence we can conclude that we can apply the above termination rule $\mathsf{Asmp2}$ to the last judgement in Example 3.6 by instantiating $\gamma = \varepsilon$ and $\beta = l_1$.

The first property of the new algorithm that we prove is termination. Intuitively, we have that this new termination rule guarantees to catch all those cases where the term on the right grows indefinitely, by anticipating outputs and accumulating inputs. These infinitely many distinct types are anyway obtainable starting from the finite set $\mathsf{reach}(Z)$, by means of output anticipations. Hence there exists $S \in \mathsf{reach}(Z)$ that can generate infinitely many of these types: this guarantees $\mathsf{antOutInf}(S)$ to be true. As observed above, the leaves of such infinitely many terms are themselves taken from the finite set $\mathsf{reach}(Z)$. This guarantees that the algorithm, among the types that can be obtained from S, visits two terms having the same leaf set. These, even if syntactically different, are equivalent as far as the subtyping game is regarded.

Concerning the precise definition of the algorithm, in order to avoid the possibility of applying two distinct rules to the same judgement, we give rule $\mathsf{Asmp2}$ the same priority as rule Asmp (both rules have highest priority). Also in this case, we use $\Sigma \vdash T \leq_{\mathsf{t}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ to denote that the latter can be obtained from the former by one rule application, and $\Sigma \vdash T \leq_{\mathsf{t}} S \rightarrow_{\mathsf{err}}$, to denote that there is no rule that can be applied to the judgement $\Sigma \vdash T \leq_{\mathsf{t}} S$.

We can now state the termination and soundness of the algorithm:

Theorem 3.11. *Given two single-out session types T and S, the algorithm applied to the initial judgement $\emptyset \vdash T \leq_{\mathsf{t}} S$ terminates.*

Theorem 3.12. *Given two single-out session types T and S, we have that there exist Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ if and only if there exist Σ'', T'', S'' such that $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$.*

Finally, we can conclude the decidability of asynchronous subtyping for single-out session types.

Corollary 3.13 (Decidability for Single-out Types). *Asynchronous subtyping for single-out session types $\leq_{\mathsf{sout}}$ is decidable.*

We now show that the above decidability results hold also for the $\leq_{\mathsf{tin,sout}}$ relation (where we further restrict the asynchronous subtyping relation not to admit contravariance on input branchings). In the algorithm we just modify the rule In of Figure 2 by changing the constraint $J \subseteq I$ in the premise into $J = I$, thus obtaining modified versions of $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ (and $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow_{\mathsf{err}}$) and $\Sigma \vdash T \leq_{\mathsf{t}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ (and $\Sigma \vdash T \leq_{\mathsf{t}} S \rightarrow_{\mathsf{err}}$). We have that Proposition 3.4, where relation $\leq_{\mathsf{tin}}$ is considered instead of $\leq$, termination Theorem 3.11 and soundness Theorem 3.12, where the modified judgments $\leq_{\mathsf{a}}$ and $\leq_{\mathsf{t}}$ are considered, still hold (they are proved with exactly the same proofs as those reported in Appendix B for the original statements).

Corollary 3.14. *Asynchronous subtyping for single-out session types without input contravariance $\leq_{\mathsf{tin,sout}}$ is decidable.*

3.3.2. Asynchronous Subtyping for Single-in Types

First of all we notice that an obvious consequence of Corollary 3.13 is that also $\leq_{\mathsf{sin,sout}}$ is decidable (we just have to add a preliminary check verifying that both types are single-in). Moreover, exploiting dual closeness, i.e. the fact that $T \leq_{\mathsf{sin}} S$ if and only if $\overline{S} \leq_{\mathsf{sout}} \overline{T}$ (see Section 2.1), we can use the algorithm presented for single-out types also for the case of single-in types.

Corollary 3.15 (Decidability for Single-in Types). *Asynchronous subtyping for single-in session types $\leq_{\mathsf{sin}}$ is decidable.*

We can therefore identify an asynchronous dual closed subtyping relation that stands in the boundary of decidability.

Corollary 3.16 (Decidability for Single-in or Single-out Types). *The asynchronous dual closed subtyping relation $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ is decidable.*

Finally, similarly as we did for $T \leq_{\mathsf{sin}} S$, by exploiting Proposition 2.10 we can use the modified algorithm employed for $\leq_{\mathsf{tin,sout}}$ subtyping for deciding the remaining relation $\leq_{\mathsf{sin,tout}}$.

Corollary 3.17. *Asynchronous subtyping for single-in session types without output covariance $\leq_{\mathsf{sin,tout}}$ is decidable.*

4. Undecidability Results

We now move to undecidability results. We first consider bounded asynchronous subtyping $\leq_{\mathsf{bound}}$. The proof in this case is a variant of the proof we already presented in our previous work [10], where we encoded the problem of checking (non) termination in queue machines (a well-known Turing powerful formalism) into checking session subtyping. Technically speaking, we resort to a different property, namely *bounded non termination*, that we here show to be undecidable for queue machines.

The second, and main, undecidability result concerns subtyping without output covariance and input contravariance $\leq_{\mathsf{tin},\mathsf{tout}}$. The proof in this case requires deep modifications to our proof technique, due to the impossibility to exploit covariance/contravariance in the queue machine encoding. We deal with the absence of covariance/contravariance by saturating each point of choice on the entire considered alphabet. This has a strong impact on the encoding because it introduces additional choices, in the session types, whose continuations do not correspond to the behaviour of the considered queue machine. This problem is solved by ensuring that these additional choices and the corresponding continuations are irrelevant as far as subtyping checking is concerned. Such solution, however, works only for a fragment of queue machines (that we call single-consuming queue machines) that we prove to be Turing complete as well.

We consider this second result interesting for the following reason: the previous undecidability proofs [10, 11] made use of both output covariance/input contravariance (already present in synchronous session subtyping) and output anticipation (specific for asynchronous subtyping), hence our new proof shows that (provided that the syntax of types is not constrained, e.g, imposing single choices for output selections or input branchings) the source of undecidability is to be precisely localized into the latter, as the former is not necessary to prove undecidability.

We first report the definition of queue machines.

4.1. Queue Machines

Queue machines are a formalism similar to pushdown automata, but with a queue instead of a stack. Queue machines are Turing-equivalent [15].

Definition 4.1 (Queue Machine). *A queue machine M is defined by a six-tuple $(Q, \Sigma, \Gamma, \$, s, \delta)$ where:*

- *Q is a finite set of states;*
- *$\Sigma \subset \Gamma$ is a finite set denoting the input alphabet;*
- *Γ is a finite set denoting the queue alphabet;*
- *$\$ \in \Gamma - \Sigma$ is the initial queue symbol;*
- *$s \in Q$ is the start state;*
- *$\delta : Q \times \Gamma \to Q \times \Gamma^*$ is the transition function.*

A *configuration* of a queue machine is an ordered pair (q, γ) where $q \in Q$ is its current state and $\gamma \in \Gamma^*$ is the content of the queue (Γ^* is the Kleene closure of Γ). The starting configuration on an input string $x \in \Sigma^*$ is $(s, x\$)$.

a) Finite Control

$$[\![q]\!]^{\mathcal{S}} = \begin{cases} \mu\mathbf{q}.\&\{A{:}\oplus\{B_1^A : \cdots \oplus \{B_{n_A}^A : [\![q']\!]^{\mathcal{S}\cup q}\}\}\}_{A\in\Gamma} \\ \qquad \text{if } q \notin \mathcal{S} \text{ and } \delta(q,A) = (q', B_1^A \cdots B_{n_A}^A) \\ \\ \mathbf{q} \qquad \text{if } q \in \mathcal{S} \end{cases}$$

b) Queue

$$[\![C_1 \cdots C_m]\!] = \&\{C_1 {:} \ldots \&\{C_m : \mu\mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A\in\Gamma}\}\}$$

Figure 3: Encoding of the Finite Control and the Queue of a Queue Machine

The transition relation $\rightarrow_M$ from one configuration to the next one is defined as $(p, A\alpha) \rightarrow_M (q, \alpha\gamma)$, when $\delta(p, A) = (q, \gamma)$. A machine M accepts an input x if it blocks by emptying the queue. Formally, x is accepted by M if $(s, x\$) \rightarrow_M^* (q, \epsilon)$ where ϵ is the empty string and $\rightarrow_M^*$ is the reflexive and transitive closure of $\rightarrow_M$. Intuitively, a queue machines is a Turing machine with a special tape that works as a FIFO queue.

The Turing completeness of queue machines is discussed by Kozen [15] (page 354, solution to exercise 99). A configuration of a Turing machine (tape, current head position and internal state) can be encoded in a queue, and a queue machine can simulate each move of the Turing machine by repeatedly consuming and reproducing the queue contents, only changing the part affected by the move itself. The undecidability of termination for queue machines follows directly from such encoding.

4.2. Bounded Asynchronous Subtyping

We now consider the notion of bounded asynchronous subtyping $\leq_{\mathsf{bound}}$ we introduced in Section 2.1,

The proof of undecidability of $\leq_{\mathsf{bound}}$ follows the approach we already used to prove the undecidability of single-choice asynchronous subtyping $\ll$ [10] (that we have commented in the Introduction). The idea is to define, given a queue machine M and its input x, two session types S and T, such that S is a subtype of T if and only if M does not accept x. More precisely, the type S models the finite control of the queue machine M while the type T models the queue that initially contains the sequence $x\$$.

More precisely, the encoding of queue machines is as follows [10].

Definition 4.2 (Queue Machine Encoding). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine, and let $C_1, \cdots, C_m \in \Gamma$, with $m \geq 0$, $q \in Q$ and $\mathcal{S} \subseteq Q$. The* finite control encoding function $[\![q]\!]^{\mathcal{S}}$ *and the* queue encoding function $[\![C_1 \cdots C_m]\!]$ *are defined as in Figure 3(a) and Figure 3(b) respectively. The initial encoding of M with input x is given by the pair of types $[\![s]\!]^{\emptyset}$ and $[\![x\$]\!]$.*

The basic idea behind the encoding of the finite control is to use a recursively defined type with a recursion variable $\mathbf{q}$ for each state q of the encoded queue machine M. The type corresponding to the recursion variable $\mathbf{q}$ starts with an input with multiple choices, one for each possible symbol that can be consumed from the queue. The continuation is composed of a sequence of single-choice inputs labeled with the symbols $B^A_1 \dots B^A_{n_A}$, where $B^A_1 \dots B^A_{n_A}$ are the symbols enqueued by the queue machine when, in state q, consumes A from the queue. Assuming that q' is the new state of M after execution of this step (i.e. $\delta(q, A) = (q', B^A_1 \dots B^A_{n_A})$), the type becomes the one corresponding to the recursion variable $\mathbf{q}'$.

On the other hand, the type modeling the queue with contents $C_1 \dots C_m$ is denoted with $[\![C_1 \dots C_m]\!]$: this type starts with a sequence of single-choice inputs labeled with the symbols $C_1 \dots C_m$, followed by a recursive type. Such type starts with an output with multiple-choices, one for each symbols that can be enqueued, followed by a single-choice input having the same label. This particular type has the following property: if one label A of the multiple-choice output is selected for anticipation during the subtyping simulation game, the corresponding single-choice input labeled with A is enqueued at the end of the sequence of inputs preceding the recursive definition. This perfectly corresponds to the behaviour of the queue in the modeled queue machine.

As mentioned above, this encoding has been already used to prove the undecidability of $\ll$ [10]. More precisely, we proved that given a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an initial input x, we have that $[\![s]\!]^{\emptyset} \ll [\![x\$]\!]$ if and only if x is not accepted by M (i.e. M does not terminate on input x). The same result does not hold for the bounded asynchronous subtyping because there are cases in which M does not accept x but $[\![s]\!]^{\emptyset} \not\leq_{\mathsf{bound}} [\![x\$]\!]$, in particular, those cases in which the subtyping simulation game generates unbounded accumulation of inputs. For this reason we have to consider a more complex undecidable property for queue machines: *bounded non termination*, i.e., the ability of a queue machine to have an infinite computation while keeping the length of the queue bounded. We now define the notion of boundedness for queue machines and then prove that bounded non termination is undecidable.

Definition 4.3 (Queue Machine Boundedness). *Let M be a queue machine and x a possible input. We say that M is bound on input x if there exists k such that, for every q and γ such that $(s, x\$) \rightarrow^*_M (q, \gamma)$, we have that $|\gamma| \leq k$.*

Lemma 4.4. *Given a queue machine M and an input x, it is undecidable whether M does not terminate and is bound on x.*

Following the proof technique we already used to prove undecidability of $\ll$, i.e. by reducing the termination problem for queue machines into subtyping checking [10], we can prove also the undecidability of $\leq_{\mathsf{bound}}$ by reduction from the bounded non termination problem.

Theorem 4.5. *Given a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string x, we have that $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$ if and only if M does not terminate and is bound on x.*

Corollary 4.6. *Bounded asynchronous subtyping $\leq_{\mathsf{bound}}$ is undecidable.*

4.3. Undecidability of Asynchronous Subtyping without Output Covariance and Input Contravariance

We now move to the proof of undecidability of $\leq_{\mathsf{tin,tout}}$, the asynchronous subyping relation, we introduced in Section 2.1, that does not admit output covariance and input contravariance by imposing matching choices to have the same set of labels.

The proof technique is still based on an encoding of queue machines, but we have to significantly improve the encoding discussed in the previous subsection. In fact, the encoding of Figure 3 exploits both input contravariance (in the matching between the multiple-choice input at the beginning of the encoding of the finite control and the initial single-choice inputs of the queue encoding) and output covariance (in the matching between the multiple-choice output at the beginning of the recursive part of the queue encoding and the single-choice outputs in the encoding of the finite control).

The new encoding that we propose saturates all choices, both inputs and outputs, with labels corresponding to the entire queue alphabet. The addition of these labels and of the corresponding continuations, introduces new possible paths in the subtyping simulation game. We are able to make these additional behaviour irrelevant, but at the price of restricting the class of encoded queue machines. These queue machines are named *single consuming queue machines*; their characteristic is to guarantee that in two subsequence actions, at least one of the two will enqueue symbols.

Definition 4.7 (Single Consuming Queue machine). *We say that a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ is* single consuming *if $\delta(q, a) = (q', \epsilon)$, for some q, a and q', implies that there exist no b and q'' such that $\delta(q', b) = (q'', \epsilon)$.*

We have that single consuming queue machines are still Turing-complete (see Appendix C.2 for the detailed proof based on an encoding of queue machines into single consuming queue machines):

Theorem 4.8. *Given a single consuming queue machine M and an input x, the termination of M on x is undecidable.*

We prove the undecidability of $\leq_{\mathsf{tin,tout}}$ by encoding single consuming queue machines into the subtyping simulation game. Following the approach already discussed in the previous subsection, given a queue machine, our encoding generates a pair of types, say T and S, such that T encodes the finite control and S encodes the queue. Then, the subtyping $T \leq_{\mathsf{tin,tout}} S$ simulates the execution of the machine.

We are now ready to present the definition of the new encoding where we make use of the following new notation: $\{l_i : T_i\}_{i\in I} \uplus \{l_j : T_j\}_{j\in J} = \{l_k : T_k\}_{k\in I\cup J}$.

Definition 4.9 (Encoding Single Consuming Queue machines). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine such that $q \in Q$, $\mathcal{S} \subseteq Q$ and $C_1, \cdots, C_m \in \Gamma$, with $m \geq 0$. The* finite control encoding function $[\![q]\!]^{\mathcal{S}}$ *and the* queue encoding function $[\![C_1 \cdots C_m]\!]$ *are defined as in Figure 4(a) and Figure 4(b) respectively.*

a) Finite Control

$$[\![q]\!]^{\mathcal{S}} = \begin{cases} \mu\mathbf{q}.\&\{A : \{\!\{B_1^A \cdots B_{n_A}^A\}\!\}_{q'}^{\mathcal{S}\cup\{q\}}\}_{A\in\Gamma} & \text{if } q \notin \mathcal{S} \text{ and } \delta(q,A) = (q', B_1^A \cdots B_{n_A}^A) \\ \mathbf{q} & \text{if } q \in \mathcal{S} \end{cases}$$

b) Queue

$$[\![C_1 \dots C_m]\!] = \begin{cases} \mu\mathbf{t} \oplus \big\{A : \&\big(\{A : \mathbf{t}\} \uplus \{A' : T''\}_{A'\in\Gamma\setminus\{A\}}\big)\big\}_{A\in\Gamma} & \text{if } m = 0 \\ \&\big(\{C_1 : [\![C_2 \dots C_m]\!]\} \uplus \{A' : T''\}_{A'\in\Gamma\setminus\{C_1\}}\big) & \text{otherwise} \end{cases}$$

---

where:

$$\{\!\{B_1 \cdots B_m\}\!\}_r^{\mathcal{T}} = \begin{cases} [\![r]\!]^{\mathcal{T}} & \text{if } m = 0 \\ \oplus\big(\{B_1 : \{\!\{B_2 \dots B_m\}\!\}_r^{\mathcal{T}}\} \uplus \{A' : T'\}_{A'\in\Gamma\setminus\{B_1\}}\big) & \text{otherwise} \end{cases}$$

$$T' = \mu\mathbf{t}.\&\big\{A_1 : \oplus\{A_2 : \mathbf{t}\}_{A_2\in\Gamma}\big\}_{A_1\in\Gamma}$$

$$T'' = \mu\mathbf{t}.\&\big\{A_1 : \&\big\{A_2 : \oplus\{A_3 : \mathbf{t}\}_{A_3\in\Gamma}\big\}_{A_2\in\Gamma}\big\}_{A_1\in\Gamma}$$

Figure 4: Encoding of the Finite Control and the Queue of a Single Consuming Queue Machine

As discussed in the previous subsection, the idea is that the type encoding the finite control is able to perform an input on each of the symbols in Γ, and continue according to the definition of the transition function δ. The type representing the queue then matches such input with the correct symbol depending on the state of the queue. For instance, in the encoding described in the previous subsection, if we denote with T and S the types representing the finite control and the queue respectively, and if $\Gamma = \{A, B\}$ and symbol A is on the head of the queue, we have $T = \&\{A : \dots, B : \dots\}$ and $S = \&\{A : \dots\}$: type T is able to react to any symbol that may be present on the queue (like the transition function δ), while type S reacts with the actual value on the queue, symbol A. Unfortunately, such idea exploits contravariance for inputs. Therefore, it must be the case, in the new encoding, that the input in S is of the form $\&\{A : \dots, B : \dots\}$. We make sure that if label A is selected then the simulation of the queue machine continues. Otherwise, an infinite subtyping simulation game is started (starting from B in the example).

Also the insertion of symbols in the queue was simulated in the encoding of the previous subsection by exploiting output covariance. The type representing the finite control performs a single-choice output that is matched by a multiple-choice output having the effect of adding a corresponding symbol at the end of the input accumulated in the type modeling the queue. Also in this case, we have to add choices to the type modeling the finite control: also in this case we ensure that these extra paths start an infinite subtyping simulation game.

These additional paths make the subtyping simulation game highly nondeterministic and such that several paths that the game can take differ from what the encoded machine does. We discuss in detail the various cases which our encoding in Figure 4 can be in:

1. *The encoding of the finite control reads the correct symbol.* We represent the machine reading a symbol A from the queue while being in state q, with an input type of the form $\&\{A : \{\!\{B_1^A \cdots B_{n_A}^A\}\!\}_{q'}^{\mathcal{S}\cup\{q\}}\}_{A\in\Gamma}$, where each branch corresponds to a possible symbol that can be read. On the other hand, a queue $C_1 \cdots C_m$ is encoded as an input type of the form $\&\big(\{C_1 : [\![C_2 \dots C_m]\!]\} \uplus \{A' : T''\}_{A'\in\Gamma\setminus\{C_1\}}\big)$ where the branch with label C_1 represents the actual content of the queue. Hence, in the simulation game, if the finite control reads symbol A and this is matched by the correct symbol in the queue, then the type $\{\!\{B_1^A \cdots B_{n_A}^A\}\!\}_{q'}^{\mathcal{S}\cup\{q\}}$ deals with inserting symbols $B_1^A \cdots B_{n_A}^A$ into the queue.
2. *The encoding of the finite control reads the wrong symbol.* In this case, the encoding of the finite control picks a symbol that is not that in the queue head. In order to match it, the encoding of the queue will take $\{A' : T''\}_{A'\in\Gamma\setminus\{C_1\}}$. Type T'' is designed in a way that it can match every move of the finite control, by repeatedly alternating two inputs with a subsequent output on every queue symbol. Note that, since inputs cannot be anticipated, matching every move is feasible only if the encoded machine is single consuming.
3. *The encoding of the finite control writes the correct symbol.* Once the finite control has read a symbol, it performs $\{\!\{B_1 \cdots B_m\}\!\}_r^{\mathcal{T}}$, which simulates the writing of $B_1 \cdots B_m$ into the queue. If $m = 0$ then it moves to the encoding of the next state according to function δ. Otherwise, it translates to the type $\oplus\big(\{B_1 : \{\!\{B_2 \dots B_m\}\!\}_r^{\mathcal{T}}\} \uplus \{A' : T'\}_{A'\in\Gamma\setminus\{B_1\}}\big)$. The queue, in order to match B_1 (and $B_2, \dots, B_m$) can always anticipate outputs with the term $\mu\mathbf{t} \oplus \{A : \&\big(\{A : \mathbf{t}\} \uplus \{A' : T''\}_{A'\in\Gamma\setminus\{A\}}\big)\}_{A\in\Gamma}$ which, after consuming a label A will add an input with label A, simulating the adding of A to the queue.
4. *The encoding of the finite control writes the wrong symbol.* In this case, the finite control writes a symbol to the queue with $\oplus\big(\{B_1 : \{\!\{B_2 \dots B_m\}\!\}_r^{\mathcal{T}}\} \uplus \{A' : T'\}_{A'\in\Gamma\setminus\{B_1\}}\big)$. However, the simulation executes the wrong output (with any $A' \neq B_1$) and continues as T'. In this case, T' continues removing and adding any value from the queue, indefinitely. Note that it may remove the wrong value from the queue overlapping with case 2. In this case, the requirement that the queue machine is single consuming is not necessary.

Example 4.10. *In order to further clarify our encoding, consider a queue machine with states $\{s, q\}$ (where s is the starting state), queue alphabet $\Gamma = \{X, Y\}$ and transition relation δ such that $\delta(s, A) = (q, A)$ and $\delta(q, A) = (s, \epsilon)$, for every $A \in \Sigma$. Clearly, the machine terminates on any input. The encoding* *of the finite control is the following session type:*

$$[\![s]\!]^{\emptyset} = \mu\mathbf{s}.\&\left\{\begin{array}{ll} X : \oplus\{X : [\![q]\!]^{s}, & Y : T'\} \\ Y : \oplus\{Y : [\![q]\!]^{s}, & X : T'\} \end{array}\right\}$$

$$[\![q]\!]^{\{s\}} = \mu\mathbf{q}.\&\left\{\begin{array}{ll} X : \oplus\{X : \mathbf{s}, & Y : T'\} \\ Y : \oplus\{Y : \mathbf{s}, & X : T'\} \end{array}\right\}$$

Assume, e.g., that the queue initially contains the string XY. The machine will empty the queue by visiting state q twice and terminate in state s with the empty queue. If we now run the subtyping simulation game between the encoding of finite control above and the encoding of the queue we will end up with two types that are not in subtyping: the encoding of the state s starting with an input and the encoding of the empty queue that does not match it.

The encoding of the finite-control and of the queue are such that the following properties hold: given a queue machine M with initial state s and initial queue symbol \$, if M does not accept x then it is possible to define an asynchronous subtyping relation that includes the pair $([\![s]\!]^{\emptyset}, [\![x\$]\!])$; moreover, if $[\![s]\!]^{\emptyset} \leq_{\mathsf{tin},\mathsf{tout}} [\![x\$]\!]$ then it is possible to conclude that M does not terminate (i.e. does not accept) on input x. We thus have the following:

Theorem 4.11. *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, we have $[\![s]\!]^{\emptyset} \leq_{\mathsf{tin},\mathsf{tout}} [\![x\$]\!]$ if and only if M does not terminate on x.*

We can therefore conclude that subtyping without output covariance and input contravariance is undecidable.

Corollary 4.12 (Undecidability of Subtying without Co/contravariance). *The asynchronous dual closed subtyping relation $\leq_{\mathsf{tin},\mathsf{tout}}$ is undecidable.*

In the same way we can also show that $\leq_{\mathsf{tin}}$ and $\leq_{\mathsf{tout}}$ are undecidable and provide an alternative proof of undecidability of $\leq$. This because, since for the types obtained with the encoding (for which the ability to match via covariance/contravariance is irrelevant) obviously such relations coincide, i.e. $[\![s]\!]^{\emptyset} \leq_{\mathsf{tin},\mathsf{tout}} [\![x\$]\!]$ if and only if $[\![s]\!]^{\emptyset} \leq_{\mathsf{tin}} [\![x\$]\!]$ if and only if $[\![s]\!]^{\emptyset} \leq_{\mathsf{tout}} [\![x\$]\!]$ if and only if $[\![s]\!]^{\emptyset} \leq [\![x\$]\!]$, Theorem 4.11 holds also if we replace the $\leq_{\mathsf{tin},\mathsf{tout}}$ relation with one of such relations.

Corollary 4.13. *Asynchronous subtyping relations $\leq_{\mathsf{tin}}$, $\leq_{\mathsf{tout}}$ and $\leq$ are undecidable.*

5. Related Work

Subtyping for Session Types. Subtyping for session types was first introduced by Gay and Hole [7][5] for a session-based π-calculus where communication is synchronous, i.e., an output directly synchronises with an input. In such case, the relation allows no output anticipation. However, as in our case, outputs are covariant and inputs are contravariant.

To the best of our knowledge, Mostrous et al. [14] were the first to adapt the notion of session subtyping to an asynchronous setting. Their computation model is a session π-calculus with asynchronous communication that makes use of session queues for maintaining the order in which messages are sent. They introduce the idea of output anticipation, which is also a main feature of our theory. Mostrous and Yoshida [12] extended the notion of asynchronous subtyping to session types for the higher-order π-calculus. In the same article, Mostrous and Yoshida observe that their definition of asynchronous subtyping allows orphan messages. Orphan message are prohibited with the definition of subtyping given by Chen et al. [8]. In their article, they show that such a definition is both sound and complete w.r.t. type safety and orphan message freedom.

Undecidability Results. Mostrous et al. [14] proposed a procedure to check asynchronous subtyping for multiparty session types. Differently from what stated therein, the procedure does not terminate due to unbounded message accumulation in the queues, e.g. for the terms in Example 3.3. Such a procedure inspired the one we presented in Section 3.1. The problem of unbounded accumulation was observed by Mostrous and Yoshida [12]. The impossibility to define a correct algorithm has been independently proved by Lange and Yoshida [11] and Bravetti et. al [10]. Lange and Yoshida [11] reduce Turing machine termination into a notion of compatibility for communicating automata and, then, transfer such a result to session types. This proof technique applies only to dual closed subtyping relations, like the one by Chen et al. [8]. The proof by Bravetti et. al [10], on the other hand, exploits a direct encoding of queue machines into session subtyping. This made it possible to prove undecidability of all the other notions of asynchronous subtyping in the literature. Unlike the encoding in this paper (Figure 4), both encodings take advantage of the use of output covariance and input contravariance. For example, by exploiting this feature, the queue machine encoding by Bravetti et al. [10] (Figure 3) is much simpler than the encoding we need to use here. We notice that our results on undecidability focus on binary session types. However, it is immediate to generalise this kind of undecidability results from binary to multiparty sessions (binary session types are just multiparty session types with only two roles [10]).

Decidability Results. Synchronous subtyping for binary session types is decidable [7]. Both Bravetti et al. [10] and Lange and Yoshida [11] investigate

[5] The Gay and Hole subtyping is contravariant on outputs and covariant on inputs. This is because a channel-based subtyping [9] is considered instead of our process-oriented subtyping.

fragments of session types for which asynchronous subtyping becomes decidable. However, such fragments are much more limited, and far from having practical applications, with respect to those considered here. Both address cases where one of the compared types is a *single-choice session type*, i.e. all its branchings and selections are single-choice. Thus they are both, basically, special cases of our subtyping for single-in or single-out types ($\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$). In particular, Lange and Yoshida give an algorithm for deciding subtyping between a general session type and a single-choice session type. Although it may seem that such case is not properly included in our decidable subtyping relation for single-out/single-in types, covariance and contravariance ensure that all types containing at least one multiple input branch and one multiple output selection (both reachable in the subtyping simulation game) cannot be related with a single-choice type. Bravetti et al. [10] prove decidability for relations $\ll_{\mathsf{sin}}$ and $\ll_{\mathsf{sout}}$ that pose an analogous restriction to the branching/selection structure, but that allow for orphan messages. $\ll_{\mathsf{sin}}$ and $\ll_{\mathsf{sout}}$ are fragments where related types (T, S) are such that, either T is single-choice and S is single-in ($\ll_{\mathsf{sin}}$), or T is single-out and S is single-choice ($\ll_{\mathsf{sout}}$). For types that do not produce orphan messages, the sutyping of Bravetti et al. [10] is just a special case of our single-in ($\leq_{\mathsf{sin}}$) and single-out ($\leq_{\mathsf{sout}}$) session subtyping.

Additionally, Lange and Yoshida state the decidability of subtyping for half-duplex communication [16] and alternating machines: the former coincides with synchronous subtyping while the latter can be reduced to 1-bounded asynchronous subtyping as discussed in Section 3.2.

Comparison with our Previous Work [10]. Concerning decidability results, the approaches taken in this and our previous paper are both based on initially considering a (non always terminating) subtyping procedure $\leq_{\mathsf{a}}$, inspired from Mostrous et al. [14], and then presenting a subtyping algorithm $\leq_{\mathsf{t}}$ obtained by adding a new termination $\mathsf{Asmp2}$ rule to the definition of the $\leq_{\mathsf{a}}$ procedure (the further additional $\mathsf{Asmp3}$ rule in our previous work is not needed when dealing with orphan message-free subtyping, as we do here). Such an $\mathsf{Asmp2}$ rule is crucial to guarantee the termination of the $\leq_{\mathsf{t}}$ algorithm: the structure of the $\mathsf{Asmp2}$ rule and the related proof of algorithm correctness and termination constitute the main contribution of this paper (as far as decidability results are concerned). In particular, since here multiple choices are admitted for input branchings, the termination condition $\mathsf{Asmp2}$ needs to deal with input accumulation in the form of a tree, instead of simple linear accumulation as in our previous work (see the discussion at the beginning Section 3.3 for details about this comparison). As a consequence $\mathsf{Asmp2}$ is completely modified: it has to deal with complex recurrent patterns to be checked on the leaves of trees representing input branchings (with multiple choices), instead of detecting simple repetitions on strings representing sequences of single-choice inputs. A detailed comparison follows. Concerning the subtyping procedure $\leq_{\mathsf{a}}$, the one considered in this paper (Figure 2) is novel in just two details: an additional orphan message-free condition is considered in the Out rule (because here, we consider orphan message-free subtyping) and the usage of the outUnf unfolding instead of unfold^n (so to perform the minimal needed per-branch unfolding). The latter is needed in this paper because, when the procedure is turned into a subtyping algorithm $\leq_{\mathsf{t}}$, we have to deal, in the new $\mathsf{Asmp2}$ termination condition, with input trees instead of strings. Correctness of $\leq_{\mathsf{a}}$ w.r.t. asynchronous subtyping is stated by Proposition 3.4 in this paper, which corresponds to Lemma 5.1 of our previous work [10]. Differently from the proof of that Lemma, here we need to cope with the different way of performing unfolding of right-hand terms in the asynchronous subtyping definition (via unfold^n) and in $\leq_{\mathsf{a}}$ (via outUnf). Termination of the $\leq_{\mathsf{t}}$ algorithm is stated by Theorem 3.11 in this paper, which corresponds to Lemma 5.2 of our previous work [10]. Here due to the new structure of the $\mathsf{Asmp2}$ rule, the proof is based on a much more complex characterization of right-hand types produced, starting from the initial Z one, by the subtyping algorithm. Such types are characterized as $\mathsf{antOut}(S, \gamma)$ with γ being a sequence of output labels and S being a type such that $S \in \mathsf{reach}(Z)$ and $\mathsf{antOutInf}(S)$. For the proof in this paper it is thus crucial to show, that $\mathsf{reach}(Z)$ is finite and $\mathsf{antOutInf}(S)$ is decidable, as stated by Proposition 3.10, which has no counterpart in our previous work. Correctness of the $\leq_{\mathsf{t}}$ algorithm w.r.t. $\leq_{\mathsf{a}}$ procedure is stated by Theorem 3.12 in this paper, which corresponds to Lemma 5.3 of our previous work. In both papers correctness follows from the following property: whenever the termination rule $\mathsf{Asmp2}$ is applied by the $\leq_{\mathsf{t}}$ algorithm, the subtyping procedure $\leq_{\mathsf{a}}$ is guaranteed to proceed forever. However, being $\mathsf{Asmp2}$ completely different, here we need to perform a totally new proof. In particular, in our previous work we simply had to prove the existence of a cycle causing the initial input string in the right-hand type to get longer in a repetitive way, here, instead, we have to manage the more complex case of input trees: this entails considering all possible branchings in order to prove that the subtyping procedure proceeds forever.

Concerning undecidability results, both the approaches taken in the two papers are based on resorting to undecidability of queue machines via a suitable encoding of them into subtyping. In particular, for showing undecidability of bounded asynchronous subtyping ($\leq_{\mathsf{bound}}$) we resort to the same encoding as in our previous work, but we consider a more complex property of queue machines that we show to be undecidable: bounded non termination. On the other hand, for showing undecidability of asynchronous subtyping without output covariance and input contravariance ($\leq_{\mathsf{tin,tout}}$), we need to introduce the novel class of single consuming queue machines and to perform a much more complex encoding which, differently from the previous one, uses nondeterminism (in order to avoid usage of covariance and contravariance, spurious nondeterministic paths that proceed forever have to be added by the encoding). The undecidability proof thus becomes significantly more complex in that the above encoding makes it much more intricated to relate the queue machine behaviour with the subtyping game and the novel class of queue machines. Moreover, introducing a new restricted class of queue machines requires showing them to be Turing-equivalent (by providing an encoding of standard queue machines into them).

6. Conclusion

In this article, we have shed light on the boundaries between decidability and undecidability of asynchronous session subtyping by analyzing two kinds of restrictions: to the branching/selection structure of inputs/outputs and to the capabilities of the communication buffer. In particular, considering all the relations in Figure 1, we have shown: decidability for those in the lower part, notably of k-bounded subtyping and of subtyping over single-out or single-in session types; and the undecidability for those in the upper part, notably of bounded subtyping and of subtyping without output covariance and input contravariance.

As future work, we plan to develop typing systems for server/client code in the context of web services, exploiting our subtyping algorithms for single-out/single-in session types. Note that, in practice, server code typically connects, as a client, to other services (e.g. a database server) using another binary session, according to the commonly used multitier architecture. Thus, in general, when typing code, we would use for a specific session one of the two algorithms above depending if the code is playing the role of the client or of the server in that session.

Moreover, we plan to investigate whether other kinds of restriction w.r.t. the two above allow us to obtain a decidable relation (thus retaining general branching/selection structure for both inputs and outputs and not limiting communication buffers).

References

[1] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: 7th European Symposium on Programming (ESOP'98), Vol. 1381 of LNCS, Springer, 1998, pp. 122–138. doi:10.1007/BFb0053567.

[2] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), 2008, pp. 273–284. doi:10.1145/1328438.1328472.

[3] S. Lindley, J. G. Morris, Embedding session types in Haskell, in: Proceedings of the 9th International Symposium on Haskell (Haskell 2016), 2016, pp. 133–145. doi:10.1145/2976002.2976018.

[4] N. Ng, N. Yoshida, Static deadlock detection for concurrent Go by global session graph synthesis, in: Proceedings of the 25th International Conference on Compiler Construction (CC 2016), 2016, pp. 174–184. doi:10.1145/2892208.2892232.

[5] T. B. L. Jespersen, P. Munksgaard, K. F. Larsen, Session types for Rust, in: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, 2015, pp. 13–22. doi:10.1145/2808098.2808100.

[6] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (2016) 9. doi:10.1145/2827695.

[7] S. J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Inf. 42 (2-3) (2005) 191–225. doi:10.1007/s00236-005-0177-z.

[8] T. Chen, M. Dezani-Ciancaglini, N. Yoshida, On the preciseness of subtyping in session types, in: 16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14), ACM, 2014, pp. 135–146. doi:10.1145/2643135.2643138.

[9] S. J. Gay, Subtyping supports safe session substitution, in: A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, 2016, pp. 95–108. doi:10.1007/978-3-319-30936-1_5.

[10] M. Bravetti, M. Carbone, G. Zavattaro, Undecidability of asynchronous session subtyping, Inf. Comput. To appear.
URL http://arxiv.org/abs/1611.05026

[11] J. Lange, N. Yoshida, On the undecidability of asynchronous session subtyping, in: Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, 2017, pp. 441–457. doi:10.1007/978-3-662-54458-7_26.

[12] D. Mostrous, N. Yoshida, Session typing and asynchronous subtyping for the higher-order π-calculus, Inf. Comput. 241 (2015) 227–263. doi:10.1016/j.ic.2015.02.002.

[13] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL), Tech. rep., W3C (2001).
URL https://www.w3.org/TR/wsdl

[14] D. Mostrous, N. Yoshida, K. Honda, Global principal typing in partially commutative asynchronous sessions, in: 18th European Symposium on Programming (ESOP'09), Vol. 5502 of LNCS, Springer, 2009, pp. 316–332. doi:10.1007/978-3-642-00590-9_23.

[15] D. Kozen, Automata and computability, Springer, New York, 1997.

[16] G. Cécé, A. Finkel, Verification of programs with half-duplex communication, Inf. Comput. 202 (2) (2005) 166–190. doi:10.1016/j.ic.2005.05.006.

Appendix A. Proofs of Section 2

Appendix A.1. Proof of Theorem 2.7 and Propositions 2.9 and 2.10

We start by proving Theorem 2.7. This is done by separately showing, as preliminary lemmas, both implications (one in each direction) to hold. The proof of such lemmas will be then also exploited to prove Propositions 2.9 and 2.10.

Lemma Appendix A.1. *Given two session types T and S, we have that $T \leq_{\mathsf{DC}} S$ implies $T \leq S$.*

Proof. Given an asynchronous dual closed subtyping relation $\mathcal{R}$ we show that $\mathcal{R}$ is also a (orphan-message-free) subtyping relation. To this aim we need to prove that if $(T, S) \in \mathcal{R}$ and $T = \oplus\{l_i : T_i\}_{i \in I}$ then the additional item in 2. of Definition 2.4 holds, i.e.

- if $\mathcal{A} \neq [\,]^1$ then $\forall i \in I. \& \in T_i$

From $\mathcal{A} \neq [\,]^1$ it follows that S, after some possible unfoldings, starts with an input (it must be in the form $\mathcal{A}[S_k]^{k \in \{1,\ldots,m\}}$). As $\mathcal{R}$ is an asynchronous dual closed subtyping relation we have $(\overline{S}, \overline{T}) \in \mathcal{R}$. We observe that $\overline{S}$, after some possible unfoldings, starts with an output and $\overline{T} = \&\{l_i : \overline{T}_i\}_{i \in I}$. For item 2. of Definition 2.6, we have that $\overline{T} = \mathcal{A}'[\oplus\{l_j : V_{kj}\}_{j \in J_k}]^{k \in \{1,\ldots,m\}}$, for some input context $\mathcal{A}'$. This means that all $\overline{T}_i$ contain at least an output selection, which implies that all T_i contain at least one input branching. $\square$

The following lemma states that $T \leq S$ implies $T \leq_{\mathsf{DC}} S$. This is proved by showing that given $T \leq S$ we have also $T \leq_{\mathsf{DC}} S$ because there exists an asynchronous dual closed subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$. Such relation is defined as follows: $\mathcal{R} = \{(T, S), (\overline{S}, \overline{T}) \mid T \leq S\}$. The proof that each pair $(T, S) \in \mathcal{R}$ satisfies the items in Definition 2.6 has only one complex case, namely, the one in which we assume $\overline{S} \leq \overline{T}$, $T = \oplus\{l_i : T_i\}_{i \in I}$ and $S = \mu\mathbf{t}_1. \ldots \mu\mathbf{t}_n.\&\{l_j : S_j\}_{j \in J}$ (case numbered 2b in the proof). In this case we have to reason on all initial output paths of $\overline{S}$ and use the *no orphan message constraint* of Definition 2.4 to be sure that such paths cannot be infinite, i.e. they eventually end in a state performing an input choice, and moreover each of these reachable input choices must match with the initial input choice of $\overline{T}$.

Lemma Appendix A.2. *Given two session types T and S, we have that $T \leq S$ implies $T \leq_{\mathsf{DC}} S$.*

Proof. We show that, given $T \leq S$, it is possible to define an asynchronous dual closed subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$. Consider

$$\mathcal{R} = \{(T, S), (\overline{S}, \overline{T}) \mid T \leq S\}$$

The relation $\mathcal{R}$ is dual closed by definition. It remains to show that it satisfies the four items in Definition 2.6. Let $(T, S) \in \mathcal{R}$. There are two cases: $T \leq S$ or $\overline{S} \leq \overline{T}$. In the first case all the item holds by definition of orphan-message-free subtyping relation. We consider now the second case, i.e. $\overline{S} \leq \overline{T}$, and proceeds with a case analysis.

1. $T = \mathbf{end}$.
   We have $\overline{T} = \mathbf{end}$. Having $\overline{S} \leq \mathbf{end}$, by definition of $\leq$, in particular by n applications of item 4. (with $n \geq 0$) and one application of item 1., it follows that $\overline{S} = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n.\mathbf{end}$. Hence $S = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n.\mathbf{end}$, then we can conclude what requested, i.e., $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \mathbf{end}$.
2. $T = \oplus\{l_i : T_i\}_{i \in I}$.
   We have $\overline{T} = \&\{l_i : \overline{T}_i\}_{i \in I}$. Having $\overline{S} \leq \&\{l_i : \overline{T}_i\}_{i \in I}$, by definition of $\leq$, we have two possible cases.
   (a) By n applications of item 4. (with $n \geq 0$) and one application of item 3., it follows that $\overline{S} = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n.\&\{l_j : \overline{S}_j\}_{j \in J}$, with $I \subseteq J$ and $\mathsf{unfold}^n(\overline{S}) = \&\{l_j : \overline{S'}_j\}_{j \in J}$ with $\overline{S'}_i \leq \overline{T}_i$ for every $i \in I$. Hence $S = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n. \oplus \{l_j : S_j\}_{j \in J}$, then we can conclude what requested, i.e., $\mathsf{unfold}^n(S) = [\oplus\{l_j : S'_j\}_{j \in J}]^1$, $I \subseteq J$ and $\forall i \in I.(T_i, S'_i) \in \mathcal{R}$. Notice that we have used the fact that $\mathsf{unfold}^n(\overline{S}) = \overline{\mathsf{unfold}^n(S)}$ and we have considered an input context $\mathcal{A} = []^1$.
   (b) By n applications of item 4. (with $n \geq 0$) and one application of item 2., it follows that $\overline{T} = \&\{l_i : \overline{T}_i\}_{i \in I} = \mathcal{A}[\oplus\{l_p : \overline{T_{kp}}\}_{p \in J_k}]^{k \in \{1,\ldots,m\}}$ (hence with $\mathcal{A} \neq []^1$), and $\overline{S} = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n. \oplus \{l_j : \overline{S}_j\}_{j \in J}$, with $\forall k \in \{1, \ldots, m\}.J \subseteq J_k$ and $\mathsf{unfold}^n(\overline{S}) = \oplus\{l_j : \overline{S'}_j\}_{j \in J}$ with $\forall j \in J.\overline{S'}_j \leq \mathcal{A}[\overline{T_{kj}}]^{k \in \{1,\ldots,m\}}$. Hence $S = \mu\mathbf{t}_1 \ldots . \mu\mathbf{t}_n.\&\{l_j : S_j\}_{j \in J}$. We now observe that there exists an input context $\mathcal{A}'$ and n', m' such that $\mathsf{unfold}^{n'}(S) = \mathcal{A}'[\oplus\{l_h : S_{kh}\}_{h \in L_k}]^{k \in \{1,\ldots,m'\}}$ with $\forall k \in \{1, \ldots, m'\}.I \subseteq L_k$. This follows from the fact that $\overline{S} \leq \&\{l_i : \overline{T}_i\}_{i \in I}$: by repeated application of the rule 2. of Definition 2.4 (that includes the no orphan message constraint), we have the guarantee that along all branches of $\overline{S}$ (and its unfoldings) it is guaranteed to reach an input branching, and by application of rule 3. (in particular the contra-variance on input branchings), the labels of such choices include the set of labels of the initial input branching of $\&\{l_i : \overline{T}_i\}_{i \in I}$. We conclude by showing that what is requested, i.e., $\forall i \in I.(T_i, \mathcal{A}'[S_{ki}]^{k \in \{1,\ldots,m'\}}) \in \mathcal{R}$, actually holds. This follows from the fact that $\overline{\mathcal{A}'[S_{ki}]^{k \in \{1,\ldots,m'\}}} \leq \overline{T}_i$, which is a consequence of $\overline{S} \leq \overline{T}$. In fact, this implies that also $\mathsf{unfold}^{n'}(\overline{S}) \leq \overline{T}$ because an orphan-message-free subtyping relation is still such even if we add pairs $(\mathsf{unfold}^r(V), Z)$ assuming (V, Z) already in the relation. Having $\mathsf{unfold}^{n'}(\overline{S}) = \overline{\mathsf{unfold}^{n'}(S)} = \overline{\mathcal{A}'}[\&\{l_h : \overline{S_{kh}}\}_{h \in L_k}]^{k \in \{1,\ldots,m'\}}$ and $\overline{T} = \&\{l_i : \overline{T}_i\}_{i \in I}$, it is easy to see that, given an orphan-message-free subtyping relation $\mathcal{R}'$ such that $(\overline{\mathcal{A}'}[\&\{l_h : \overline{S_{kh}}\}_{h \in L_k}]^{k \in \{1,\ldots,m'\}}, \&\{l_i : \overline{T}_i\}_{i \in I}) \in \mathcal{R}'$, the relation obtained by enriching $\mathcal{R}'$ with the pairs $(\overline{\mathcal{A}''}[\overline{S_{ki}}]^{k \in K \subseteq \{1,\ldots,m'\}}, \overline{T'}_i)$, where $\mathcal{A}''$ and types T'_i, with $i \in I$, are such that $(\overline{\mathcal{A}''}[\&\{l_h : \overline{S_{kh}}\}_{h \in L_k}]^{k \in K \subseteq \{1,\ldots,m'\}}, \&\{l_i : \overline{T'}_i\}_{i \in I}) \in \mathcal{R}'$, is still an orphan-message-free subtyping relation. Above we adopt an abuse of notation for input contexts: $\overline{\mathcal{B}}[W_k]^{k \in K \subseteq \{1,\ldots,t\}}$ does not have holes numbered consistently from 1 to t, but some numbers in $\{1, \ldots, t\}$ could be missing.
3. $T = \&\{l_i : T_i\}_{i \in I}$.

We have $\overline{T} = \oplus\{l_i : \overline{T}_i\}_{i\in I}$. Having $\overline{S} \leq \oplus\{l_i : \overline{T}_i\}_{i\in I}$, by definition of $\leq$, in particular by n applications of item 4. (with $n \geq 0$) and one application of item 2., it follows that $\overline{S} = \mu\mathbf{t}_1 \ldots \mu\mathbf{t}_n.\oplus\{l_j : \overline{S}_j\}_{j\in J}$, with $J \subseteq I$, and $\mathsf{unfold}^n(\overline{S}) = \oplus\{l_j : \overline{S'}_j\}_{j\in J}$ with $\overline{S'}_j \leq \overline{T}_j$ for every $j \in J$. Hence $S = \mu\mathbf{t}_1 \ldots \mu\mathbf{t}_n.\&\{l_j : S_j\}_{j\in J}$, then we can conclude what requested, i.e., $\mathsf{unfold}^n(S) = \&\{l_j : S'_j\}_{j\in J}$, $J \subseteq I$ and $\forall j \in J.(T_j, S'_j) \in \mathcal{R}$. Notice that we have used the fact that $\mathsf{unfold}^n(\overline{S}) = \overline{\mathsf{unfold}^n(S)}$.

4. $T = \mu\mathbf{t}.T'$.
   We first observe that $V \leq \mu\mathbf{t}.Z$ implies $V \leq Z\{\mu\mathbf{t}.Z/\mathbf{t}\}$. This directly follows from the fact that if $(V, \mu\mathbf{t}.Z)$ belongs to an orphan-message-free subtyping relation, then the same relation enriched with the pair $(V, Z\{\mu\mathbf{t}.Z/\mathbf{t}\})$ is still an orphan-message-free subtyping relation. We now proceed by considering $\overline{T} = \mu\mathbf{t}.\overline{T'}$. As $\overline{S} \leq \overline{T}$, we have $\overline{S} \leq \mu\mathbf{t}.\overline{T'}$. By the above observation we have $\overline{S} \leq \overline{T'}\{\mu\mathbf{t}.\overline{T'}/\mathbf{t}\}$ that implies what requested, i.e., $(T'\{\mu\mathbf{t}.T'/\mathbf{t}\}, S) \in \mathcal{R}$. $\square$

Theorem 2.7. *Given two session types T and S, we have $T \leq S$ if and only if $T \leq_{\mathsf{DC}} S$.*

Proof. Direct consequence of Lemmas Appendix A.1 and Appendix A.2. $\square$

Proposition 2.9. *The $\leq_{\mathsf{tin,tout}}$ relation and the $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ relation are asynchronous dual closed subtyping relations.*

Proof. We first show that $\leq_{\mathsf{tin,tout}}$ is an asynchronous dual closed subtyping relation. We consider $\mathcal{R} = \{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{tin,tout}} S\}$ and show that it is an asynchronous subtyping relation when in Definition 2.4 we require $I = J_k$ in item 2. and $I = J$ in item 3. to hold. This implies $\{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{tin,tout}} S\} \subseteq \leq_{\mathsf{tin,tout}}$, thus showing that $\leq_{\mathsf{tin,tout}}$ is dual closed. Given $(T, S) \in \mathcal{R}$, we show that $\overline{S} \leq_{\mathsf{tin,tout}} \overline{T}$ implies items 1.-4. of Definition 2.4 (where we require $I = J_k$ in item 2. and $I = J$ in item 3.), apart from the no orphan message constraint of item 2., by case analysis on the structure of type T exactly as in the proof of Lemma Appendix A.2 (where $\leq_{\mathsf{tin,tout}}$ is considered instead of $\leq$ and all subset inclusions related to covariance/contravariance are replaced by subset equalities). Concerning the no orphan message constraint of item 2., in the case 2.a of the proof of Lemma Appendix A.2 just an $[\,]^1$ input context arises (so it obviously holds); in the case 2.b, instead, a generic input context $\mathcal{A}'$ arises: if $\mathcal{A}' \neq [\,]^1$ then this means that $\overline{S}$, after some possible unfoldings, starts with an output and the constraint is an immediate consequence of the fact that $\overline{S} \leq_{\mathsf{tin,tout}} \overline{T}$ (as in the proof of Lemma Appendix A.1).

We now show that $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$ is an asynchronous dual closed subtyping relation. We use T^{in} and T^{out} to denote the set of single-in and single-out session types, respectively. We have $\leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}} = (\leq \cap T^{in} \times T^{in}) \cup (\leq \cap T^{out} \times T^{out}) = (\leq_{\mathsf{DC}} \cap T^{in} \times T^{in}) \cup (\leq_{\mathsf{DC}} \cap T^{out} \times T^{out})$, due to Theorem 2.7. We now show that, for any $(T, S) \in \leq_{\mathsf{sin}} \cup \leq_{\mathsf{sout}}$, all constraints considered by Definition 2.6 hold. We take $(T, S) \in \leq_{\mathsf{DC}} \cap T^{in} \times T^{in}$, the other case $(T, S) \in \leq_{\mathsf{DC}} \cap T^{out} \times T^{out}$ is dealt with symmetrically. Since $(T, S) \in \leq_{\mathsf{DC}}$ we have that (T, S) satisfies all constraints in items 1.-4. of Definition 2.6: we just have to additionally observe that, since all reached pairs belong to $\leq_{\mathsf{DC}}$, they also obviously belong to $\leq_{\mathsf{DC}} \cap T^{in} \times T^{in}$. Concerning the duality constraint, from $(T, S) \in \leq_{\mathsf{DC}}$, we have $(\overline{S}, \overline{T}) \in \leq_{\mathsf{DC}}$, hence $(\overline{S}, \overline{T}) \in (\leq_{\mathsf{DC}} \cap T^{out} \times T^{out})$. $\square$

Proposition 2.10. *The $\leq_{\mathsf{sin,tout}}$ and $\leq_{\mathsf{tin,sout}}$ relations are such that: $T \leq_{\mathsf{sin,tout}} S$ if and only if $\overline{S} \leq_{\mathsf{tin,sout}} \overline{T}$.*

Proof. Concerning the only if part, we show $\{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{sin,tout}} S\} \subseteq \leq_{\mathsf{tin,sout}}$ as follows. We consider $\mathcal{R} = \{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{sin,tout}} S\}$ and show that it is an asynchronous subtyping relation when in Definition 2.4 we require $I = J$ in item 3. to hold and related types to be both single-out. Given $(\overline{S}, \overline{T}) \in \mathcal{R}$, we obviously have that $\overline{S}$ and $\overline{T}$ are both single-out and we show that $T \leq_{\mathsf{sin,tout}} S$ implies items 1.-4. of Definition 2.4 (where in item 3. we require $I = J$) as in the proof of Proposition 2.9. The only difference is that, when resorting to the case analysis in the proof of Lemma Appendix A.2 we consider $\leq_{\mathsf{sin,tout}}$ instead of $\leq$ and we replace all subset inclusions related to covariance/contravariance in item 3. and the subset inclusion $J \subseteq J_k$ in item 2. by equalities.

Concerning the if part, we show $\{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{tin,sout}} S\} \subseteq \leq_{\mathsf{sin,tout}}$ in a completely symmetric way by observing that $\mathcal{R} = \{(\overline{S}, \overline{T}) \,|\, T \leq_{\mathsf{tin,sout}} S\}$ is an asynchronous subtyping relation when in Definition 2.4 we require $I = J_k$ in item 2. to hold and related types to be both single-in. In this case, when resorting to the case analysis in the proof of Lemma Appendix A.2 we consider $\leq_{\mathsf{tin,sout}}$ instead of $\leq$ and we replace all subset inclusions related to covariance/contravariance in item 2., apart from $J \subseteq J_k$, by equalities. $\square$

Appendix B. Proofs of Section 3

Appendix B.1. Proof of Proposition 3.4

Proposition 3.4 states that the procedure defined in Figure 2 is a semi-algorithm for checking whether $T \not\leq S$. The proof of the proposition is divided in two parts. The first one shows that if it is not possible to reach a judgement in which no rule can be applied, i.e. there exist no Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$, then it is possible to define an asynchronous subtyping relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$. The second part shows that if $T \leq S$ then the procedure either continues indefinitely or terminates successfully by application of the Asmp or End rules.

Proposition 3.4. *Given the types T and S, we have that there exist Σ', T', S' such that $T \not\leq S$ if and only if $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$.*

Proof. We prove the two implications separately.

We start with the *only if* part and proceed by contraposition, i.e, we assume that it is not true that $\exists \Sigma', T', S'.\ \emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ and show that $T \leq S$.

In order to do this we need to perform a preliminary observation: under the assumption that it is not true that $\exists \Sigma', T', S'.\ \emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$, even if we remove rule Asmp from the procedure it is still impossible to reach a judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$, i.e. a judgement on which no rule can be applied. This can be showed as follows. Let $\rightarrow_{noAsmp}$ be our decision procedure under the assumption that Asmp is not used. Observe that the set of pairs Σ in the judgements is irrelevant for the decision procedure $\rightarrow_{noAsmp}$ because Asmp is the unique rule influenced by it. Now, by contraposition, assume $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$. Since the standard procedure $\rightarrow^*$ cannot reach $\rightarrow_{\mathsf{err}}$, we must have that there exists an intermediary judgement $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ where Asmp is applied. That is, there exists $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ such that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ (notice the use of the standard procedure), $(T'', S'') \in \Sigma''$ and $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S'' \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S'$. Within the sequence of rule applications $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S'' \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ we consider the last judgement $\Sigma''' \vdash T''' \leq_{\mathsf{a}} S'''$ such that $(T''', S''') \in \Sigma''$ (such judgement exists as the first one $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ already has this property). It is not restrictive to assume that in the sequence $\Sigma''' \vdash T''' \leq_{\mathsf{a}} S''' \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ there are no two judgements $\Sigma_1 \vdash T_1 \leq_{\mathsf{a}} S_1$ and $\Sigma_2 \vdash T_2 \leq_{\mathsf{a}} S_2$ with $T_1 = T_2$ and $S_1 = S_2$ (otherwise we can shorten the sequence $\Sigma''' \vdash T''' \leq_{\mathsf{a}} S''' \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ by removing the steps between the judgements $\Sigma_1 \vdash T_1 \leq_{\mathsf{a}} S_1$ and $\Sigma_2 \vdash T_1 \leq_{\mathsf{a}} S_1$,[6] obtaining a new one having the same properties but without the second judgement $\Sigma_2 \vdash T_1 \leq_{\mathsf{a}} S_1$). Consider now, in the standard application of the procedure $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$, the intermediary judgement $\Sigma_i \vdash T''' \leq_{\mathsf{a}} S'''$ that added (T''', S''') to the environment. We have that $(T''', S''') \notin \Sigma_i$ (otherwise rule Asmp was applied that does not introduce any new pair in Σ_i) and moreover $\Sigma_i \subset \Sigma''$ as the sets of type pairs grow monotonically during the execution of the decision procedure $\rightarrow^*$. These last observations guarantee that there exists a standard application of the procedure $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma_i \vdash T''' \leq_{\mathsf{a}} S''' \rightarrow^* \Sigma'_i \vdash T'''' \leq_{\mathsf{a}} S'''' \rightarrow_{\mathsf{err}}$ simply by considering from $\Sigma_i \vdash T''' \leq_{\mathsf{a}} S'''$ the same rules used in the sequence $\Sigma''' \vdash T''' \leq_{\mathsf{a}} S''' \rightarrow^*_{noAsmp} \Sigma' \vdash T' \leq_{\mathsf{a}} S'$. This holds because the pairs of types in the judgements traversed by this sequence are not in Σ'' (due to the assumption on the judgement $\Sigma''' \vdash T''' \leq_{\mathsf{a}} S'''$) hence also not in Σ_i, and moreover such pairs are all distinct (guaranteed by the assumption on the absence of two judgements $\Sigma_1 \vdash T_1 \leq_{\mathsf{a}} S_1$ and $\Sigma_2 \vdash T_2 \leq_{\mathsf{a}} S_2$ with $T_1 = T_2$ and $S_1 = S_2$).

Consider now the relation $\mathcal{R} = \{(T', S') \mid \exists \Sigma'. \Sigma' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}\}$ where $\mathcal{S}$ is the minimal set of judgements satisfying the following:

- $\emptyset \vdash T \leq_{\mathsf{a}} S \in \mathcal{S}$;
- if $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$ and $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow \Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$, without applying rule Asmp or RecR_2, then $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S'' \in \mathcal{S}$;
- if $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$ and $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow \Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ by applying RecR_2, then $\Sigma'' \vdash T'' \leq_{\mathsf{a}} \mathsf{unfold}^{\mathsf{outDepth}(S')}(S') \in \mathcal{S}$.

We observe that to each judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$ it is always possible to apply at least one rule. In fact, if this is not possible, we would have also $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^*_{noAsmp} \Sigma'' \vdash T'' \leq_{\mathsf{a}} S'' \rightarrow_{\mathsf{err}}$ for a judgement $\Sigma'' \vdash T'' \leq_{\mathsf{a}} S''$ with $T'' = T'$ and S'' less unfolded than S'. In fact, the unique difference between

[6]The sequence of rules applied to $\Sigma_2 \vdash T_1 \leq_{\mathsf{a}} S_1$ can be applied also to $\Sigma_1 \vdash T_1 \leq_{\mathsf{a}} S_1$ because, as already observed, $\rightarrow_{noAsmp}$ is not sensitive to the differences between Σ_1 and Σ_2.

the judgements in $\mathcal{S}$ and those reachable without adopting Asmp is that those in $\mathcal{S}$ are more unfolded (see the difference between $\mathsf{outUnf}(S)$ used in rule RecR_2 and $\mathsf{unfold}^{\mathsf{outDepth}(S')}(S')$ used in the definition of $\mathcal{S}$).

We finally show that $\mathcal{R}$ is an (orphan-message-free) subtyping relation according to Definition 2.4. Let $(T', S') \in \mathcal{R}$. Then $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$ and it is possible to apply at least one rule to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$. We proceed by cases on T'.

- If $T' = \mathbf{end}$ then item 1. of Definition 2.4 for pair (T', S') is shown by induction on $k = \mathsf{nrec}(S')$, i.e. the number of unguarded (not prefixed by some input or output) occurrences of recursions $\mu\mathbf{t}.S''$ in S' for any $S'', \mathbf{t}$.

  - Base case $k = 0$. The only rule applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is End, that immediately yields the desired pair of $\mathcal{R}$.

  - Induction case $k > 0$. The only rules applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ are Asmp and RecR_1. In the case of Asmp we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$. RecR_1 can be applied to $\Sigma'' \vdash T' \leq_{\mathsf{a}} S'$. So for some Σ''' ($= \Sigma'$ or $= \Sigma''$) we have that the procedure applies rule RecR_1 to $\Sigma''' \vdash T' \leq_{\mathsf{a}} S'$. Hence $\Sigma''' \vdash T' \leq_{\mathsf{a}} S' \rightarrow \Sigma'''' \vdash T' \leq_{\mathsf{a}} \mathsf{unfold}^1(S')$. Since $\mathsf{nrec}(\mathsf{unfold}^1(S')) = k - 1$, by induction hypothesis item 1. of Definition 2.4 holds for pair $(T', \mathsf{unfold}^1(S'))$, hence it holds for pair (T', S').

- If $T' = \oplus\{l_i : T_i\}_{i \in I}$ then item 2. of Definition 2.4 for pair (T', S') is shown as follows.

  - If $\mathsf{outDepth}(S') = 0$ then the only rule applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is Out, that immediately yields the desired pairs of $\mathcal{R}$.

  - If $\mathsf{outDepth}(S') \geq 1$ then the only rules applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ are Asmp and RecR_2. In the case of Asmp we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$. RecR_2 can be applied to $\Sigma'' \vdash T' \leq_{\mathsf{a}} S'$. So for some Σ''' ($= \Sigma'$ or $= \Sigma''$) we have that the procedure applies rule RecR_2 to $\Sigma''' \vdash T' \leq_{\mathsf{a}} S'$. Hence $(T', \mathsf{unfold}^{\mathsf{outDepth}(S')}(S')) \in \mathcal{R}$. Since $\mathsf{outDepth}(\mathsf{unfold}^{\mathsf{outDepth}(S')}(S')) = 0$, we end up in the previous case. Therefore item 2. of Definition 2.4 holds for pair $(T', \mathsf{unfold}^{\mathsf{outDepth}(S')}(S'))$, hence it holds for pair (T', S').

- If $T' = \&\{l_i : T_i\}_{i \in I}$ then item 3. of Definition 2.4 for pair (T', S') is shown by induction on $k = \mathsf{nrec}(S')$.

  - Base case $k = 0$. The only rule applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is In, that immediately yields the desired pairs of $\mathcal{R}$.

  - Induction case $k > 0$. The only rules applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ are Asmp and RecR_1. In the case of Asmp we have that $(T', S') \in \Sigma'$, hence there exists Σ'' with $(T', S') \notin \Sigma''$ such that $\Sigma'' \vdash T' \leq_{\mathsf{a}} S' \in \mathcal{S}$.

RecR_1 can be applied to $\Sigma'' \vdash T' \leq_{\mathsf{a}} S'$. So for some Σ''' ($= \Sigma'$ or $= \Sigma''$) we have that the procedure applies rule RecR_1 to $\Sigma''' \vdash T' \leq_{\mathsf{a}} S'$. Hence $\Sigma''' \vdash T' \leq_{\mathsf{a}} S' \rightarrow \Sigma'''' \vdash T' \leq_{\mathsf{a}} \mathsf{unfold}^1(S')$. Since $\mathsf{nrec}(\mathsf{unfold}^1(S')) = k-1$, by induction hypothesis item 3. of Definition 2.4 holds for pair $(T', \mathsf{unfold}^1(S'))$, hence it holds for pair (T', S').

- If $T' = \mu\mathbf{t}.T'$ then item 4. of Definition 2.4 for pair (T', S') holds because the only rule applicable to $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is RecL that immediately yields the desired pair of $\mathcal{R}$.

We now prove the *if* part and proceed by contraposition. We assume that $T{\leq}S$ and show that there exist no Σ', T', S', such that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$. So we can assume the existence of a relation $\mathcal{R}$ that is an (orphan-message-free) subtyping relation, according to Definition 2.4, such that $(T, S) \in \mathcal{R}$.

We say that $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow_w \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ if $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ and: the last rule applied is one of Out, In or RecL rules; while all previous ones are RecR_1 or RecR_2 rules. As another notation we use input-output-end contexts $\mathcal{B}$ defined as the input contexts in Definition 2.3 with the difference that also the output construct and **end** are part of the grammar in the definition.

We start by showing that $\exists \Sigma.\, \emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow_w^* \Sigma \vdash T' \leq_{\mathsf{a}} S'$ implies $S' = \mathcal{B}[S_k]^{k \in \{1\ldots m\}}$, $S_k = \mu\mathbf{t_k}.S'_k$, for some $\mathbf{t_k}$ and S'_k, and $\exists n_1, \ldots, n_m.\, (T', \mathcal{B}[\mathsf{unfold}^{n_k}(S_k)]^{k \in \{1\ldots m\}}) \in \mathcal{R}$. The proof is by induction on the length of such computation $\rightarrow_w^*$. The base case is for a 0 length computation: it yields $(T, S) \in \mathcal{R}$ which holds. For the inductive case we assume it to hold for all computations of a length k and we show it to holds for all computations of length $k+1$, by considering all judgements $\Sigma' \vdash T'' \leq_{\mathsf{a}} S''$ such that $\Sigma \vdash T' \leq_{\mathsf{a}} S' \rightarrow_w \Sigma' \vdash T'' \leq_{\mathsf{a}} S''$. This is shown by first considering the case in which rule Asmp applies to $\Sigma \vdash T' \leq_{\mathsf{a}} S'$: in this case there is no such a judgement and there is nothing to prove. Then we consider the case in which $T' = \mathbf{end}$ and $\Sigma \vdash \mathbf{end} \leq_{\mathsf{a}} S' \rightarrow^* \Sigma''' \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}$ (by applying RecR_1 rules) and rule End applies to $\Sigma''' \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}$. Also in this case there is no such a judgement $\Sigma' \vdash T'' \leq_{\mathsf{a}} S''$ and there is nothing to prove. Finally, we proceed by an immediate verification that judgements $\Sigma' \vdash T'' \leq_{\mathsf{a}} S''$ produced in remaining cases are required to be in $\mathcal{R}$ by items 2., 3. and 4. of Definition 2.4: $T' = \oplus\{l_i : T_i\}_{i \in I}$ ($\rightarrow_w$ is a possibly empty sequence of RecR_2 applications followed by Out application), $T' = \&\{l_i : T_i\}_{i \in I}$ ($\rightarrow_w$ is a possibly empty sequence of RecR_1 applications followed by In application) or $T' = \mu\mathbf{t}.T'$ ($\rightarrow_w$ is simply RecL application).

We finally observe that, given a judgement $\Sigma \vdash T' \leq_{\mathsf{a}} S'$ such that $S' = \mathcal{B}[S_k]^{k \in \{1\ldots m\}}$, $S_k = \mu\mathbf{t_k}.S'_k$, for some $\mathbf{t_k}$ and S'_k, and $\exists\, n_1, \ldots, n_m\,.\, (T', \mathcal{B}[\mathsf{unfold}^{n_k}(S_k)]^{k \in \{1\ldots m\}}) \in \mathcal{R}$ we have:

- either rule Asmp applies to $\Sigma \vdash T' \leq_{\mathsf{a}} S'$, or
- $T' = \mathbf{end}$ and, by item 1. of Definition 2.4, there exists Σ' such that $\Sigma \vdash \mathbf{end} \leq_{\mathsf{a}} S' \rightarrow^* \Sigma' \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}$ (by applying RecR_1 rules) and rule

End is the unique rule applicable to $\Sigma' \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}$, with RecR_1 being the unique rule applicable to intermediate judgements, or

- by items 2., 3. and 4. of Definition 2.4, there exist Σ', T'', S'' such that $\Sigma \vdash T' \leq_{\mathsf{a}} S' \to_w^* \Sigma' \vdash T'' \leq_{\mathsf{a}} S''$, with each intermediate judgement having a unique applicable rule. In particular this holds for $T' = \oplus\{l_i : T_i\}_{i\in I}$ ($\to_w$ is a possibly empty sequence of RecR_2 applications followed by Out application), $T' = \&\{l_i : T_i\}_{i\in I}$ ($\to_w$ is a possibly empty sequence of RecR_1 applications followed by In application) or $T' = \mu\mathbf{t}.T'$ ($\to_w$ is simply RecL application). □

Appendix B.2. Proof of Theorem 3.5

Theorem 3.5 states that $\leq_{\mathsf{a}}^{\mathsf{k}}$ indeed provides an algorithm for checking whether $T \not\leq_{\mathsf{k}} S$, this is proved in the following by also resorting to the proof of Proposition 3.4.

Theorem 3.5. *The algorithm for $\leq_{\mathsf{a}}^{\mathsf{k}}$ always terminates and, given the types T and S, there exist Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}}^{\mathsf{k}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \to_{\mathsf{err}}$ if and only if $T \not\leq_{\mathsf{k}} S$.*

Proof. We first observe that the decision algorithm for k-bounded asynchronous subtyping terminates. By contraposition, if the algorithm does not terminate, there exists an infinite sequence $\Sigma \vdash T \leq_{\mathsf{a}} S \to \Sigma_1 \vdash T_1 \leq_{\mathsf{a}} S_1 \to^* \Sigma_i \vdash T_i \leq_{\mathsf{a}} S_i \to^*$. Along this infinite sequence infinitely many distinct pairs (T, S) will be added to Σ. As only finitely many distinct terms can be reached as first element of the pairs, there will be infinitely many distinct terms as second element. Such terms will have unbounded depth, but this is not possible due to the constraint added to rule Out that impose the use of k-bounded input contexts.

We now prove that, given the types T and S, there exist Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}}^{\mathsf{k}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \to_{\mathsf{err}}$ if and only if $T \not\leq_{\mathsf{k}} S$.

We start with the *if* part and proceed by contraposition. We assume that it is not true that $\exists \Sigma', T', S'.\ \emptyset \vdash T \leq_{\mathsf{a}}^{\mathsf{k}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{a}}^{\mathsf{k}} S' \to_{\mathsf{err}}$ and we build a relation $\mathcal{R}$ that we show to be a k-bounded Asynchronous Subtyping relation. The relation $\mathcal{R}$ is built from the judgments $\Sigma'' \vdash T'' \leq_{\mathsf{a}}^{\mathsf{k}} S''$ exactly as we did for the $\leq_{\mathsf{a}}$ subtyping procedure in the first part (the *if* part) of the proof of Proposition 3.4. In such a proof we show $\mathcal{R}$ to be an orphan-message-free subtyping relation, hence we just have to show it to be k-bounded. It is immediate to observe that, since when applying rule Out to a judgment $\Sigma'' \vdash T'' \leq_{\mathsf{a}}^{\mathsf{k}} S''$ we require the input context $\mathcal{A}$ to be k-bounded, we may include in $\mathcal{R}$ only pairs (T'', S'') that satisfy the same constraint in item 2 of k-bounded Asynchronous Subtyping relation definition (Definition 2.11), because otherwise we would have $\Sigma'' \vdash T'' \leq_{\mathsf{a}}^{\mathsf{k}} S'' \to^* \Sigma''' \vdash T''' \leq_{\mathsf{a}}^{\mathsf{k}} S''' \to_{\mathsf{err}}$ by possibly applying $\mathsf{RecR}_1/\mathsf{RecR}_2$ rules. Hence, as justified in Proposition 3.4 this would lead to violating the assumption that the algorithm does not reach an error. The justification provided there still holds because judgments $\Sigma'' \vdash T'' \leq_{\mathsf{a}}^{\mathsf{k}} S_1''$ and $\Sigma'' \vdash T'' \leq_{\mathsf{a}}^{\mathsf{k}} S_2''$, with S_1'' and S_2'' that just differ for the level of internal unfoldings, behave equivalently with respect to errors due to k-boundedness violations. This because the k-boundedness of context $\mathcal{A}$ is established by the Out rule after unfolding in S''_1/S''_2 all recursions occurring before the first output of every possible branch by means of the $\mathsf{RecR}_1/\mathsf{RecR}_2$ rules.

We now prove the *only if* part and proceed by contraposition. We assume that $T\leq_{\mathsf{k}}S$ and show that there exist no Σ', T', S', such that $\emptyset \vdash T \leq^{\mathsf{k}}_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq^{\mathsf{k}}_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$. If $T\leq_{\mathsf{k}}S$ then also $T\leq S$. So we can assume the existence of a relation $\mathcal{R}$ that is an orphan-message-free subtyping relation such that $(T, S) \in \mathcal{R}$. We then use exactly the same proof as that of the second part (the *only if* part) of the proof of Proposition 3.4 to establish a correspondance between judgements $\Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$, such that $\emptyset \vdash T \leq^{\mathsf{k}}_{\mathsf{a}} S \rightarrow^*_w \Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$, and pairs in $\mathcal{R}$ (see the construction of the corresponding pair in the proof of Proposition 3.4). Since $\mathcal{R}$ includes only pairs that satisfy the constraint in item 2 of k-bounded Asynchronous Subtyping relation definition (Definition 2.11) requiring context $\mathcal{A}$ to be k-bounded; and since any judgment $\Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$ such that $\emptyset \vdash T \leq^{\mathsf{k}}_{\mathsf{a}} S \rightarrow^*_w \Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$ implies there is in $\mathcal{R}$ a corresponding pair (T'', S''_1), with S''_1 differing from S'' just for the level of internal unfoldings, we have that reachable judgments $\Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$ cannot be such that: $\Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S'' \rightarrow^* \Sigma''' \vdash T''' \leq^{\mathsf{k}}_{\mathsf{a}} S'''$, by possibly applying $\mathsf{RecR}_1/\mathsf{RecR}_2$ rules, and $\Sigma''' \vdash T''' \leq^{\mathsf{k}}_{\mathsf{a}} S''' \rightarrow_{\mathsf{err}}$ due to not satisfying the requirement about the input context $\mathcal{A}$ to be k-bounded in the rule Out. This because the difference in unfolding levels between S'' and S''_1 (inside judgment $\Sigma'' \vdash T'' \leq^{\mathsf{k}}_{\mathsf{a}} S''$ and the corresponding pair (T'', S''_1) in $\mathcal{R}$) is not significant: the k-boundedness of context $\mathcal{A}$ is established both in the rule Out and in item 2 of $\leq_{\mathsf{k}}$ definition after unfolding all recursions occurring before the first output of every possible branch.

This observation makes it possible to carry out the proof as in Proposition 3.4, hence to show that there exist no Σ', T', S', such that $\emptyset \vdash T \leq^{\mathsf{k}}_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq^{\mathsf{k}}_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$. □

Appendix B.3. Proof of Proposition 3.10

We start by providing the proof of the first part of Proposition 3.10, i.e. finiteness of $\mathsf{reach}(T)$ for single-out session types T, as a separate preliminary lemma, then we move to proving the second part, i.e. decidability of $\mathsf{antOutInf}(T)$.

Lemma Appendix B.1. *Given a single-out session type T, $\mathsf{reach}(T)$ is finite.*

Proof. We now define a finite set of session types $\mathsf{fin}(T)$, and then we prove that it satisfies all the constraints $1., \ldots, 4.$ in Definition 3.9. Hence $\mathsf{reach}(T) \subseteq \mathsf{fin}(T)$ by definition, from which finiteness of $\mathsf{reach}(T)$ follows.

It is not restrictive to assume that all the recursion variables of T are distinct: let $\mathbf{x_1}, \ldots, \mathbf{x_n}$ be such variables. We consider the rewriting variables $X_1, \ldots, X_n$. Let T_i be such that $\mu\mathbf{x_i}.T_i$ occurs in T; let T' be T with X_i that replaces $\mu\mathbf{x_i}.T_i$; and similarly let T'_i be T_i with X_j that replaces each occurrence of $\mu\mathbf{x_j}.T_j$ and $\mathbf{x_j}$. We now consider the rewriting rules $X_i \rightarrow^1_i T'_i$ and $X_i \rightarrow^2_i \mathbf{x_i}$. Given one of the above term S containing rewriting variables, we denote with $\mathsf{close}(S)$ the session type obtained by repeated application of the rewriting rules in the following way: if X_i occurs inside a subterm $\mu\mathbf{x_i}.S'$ apply $\rightarrow_i^2$, otherwise apply $\rightarrow_i^1$. We now define another closure function on sets of terms $\mathcal{S}$: $\mathsf{subterms}(\mathcal{S}) = \{S' | S' \text{ is a subterm of } S \in \mathcal{S}\}$. Consider finally $\mathsf{fin}(T) = \{\mathsf{close}(S) | S \in \mathsf{subterms}(\{T', T_1', \ldots, T_n'\})\}$. We have that $\mathsf{fin}(T)$ is finite and it satisfies all the constraints $1., \ldots, 4.$ in Definition 3.9. □

We now report some definitions and preliminary results used in the proof of Proposition 3.10 concerning the second part about decidability of $\mathsf{antOutInf}(T)$ for single-out session types T. We introduce the relation antEq_T: intuitively, $(T', T'') \in \mathsf{antEq}_T$ if T' and T'' are terms in $\mathsf{reach}(T)$ capable of anticipating the same infinite sequence of outputs. For instance, assuming that the following $T' = \mu\mathbf{t}. \oplus \{l : \&\{l_1 : \oplus\{l' : \mathbf{t}\}, l_2 : \oplus\{l' : \mathbf{t}\}\}\}$ and $T'' = \mu\mathbf{t}.\&\{l_1 : \oplus\{l : \oplus\{l' : \mathbf{t}\}\}\}$ belong to $\mathsf{reach}(T)$, for some session type T, we have that $(T', T'') \in \mathsf{antEq}_T$ because they can anticipate the sequence of outputs l and l', indefinitely.

Definition Appendix B.2. *Let T be a single-out session type. A relation $\mathcal{R}$ over $\mathsf{reach}(T)$ is an antEq_T relation if $(T', T'') \in \mathcal{R}$ implies: there exist $l, \mathcal{A}', \mathcal{A}''$ such that $\mathsf{outUnf}(T') = \mathcal{A}'[\oplus\{l \colon T_i'\}]^{i\in\{1,\ldots,n\}}$ and $\mathsf{outUnf}(T'') = \mathcal{A}''[\oplus\{l \colon T_j''\}]^{j\in\{1,\ldots,m\}}$, with $(T_i', T_j'') \in \mathcal{R}$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. We say that $T' \,\mathsf{antEq}_T\, T''$ if there is an antEq_T relation $\mathcal{R}$ such that $(T', T'') \in \mathcal{R}$.*

Notice that antEq_T itself is an antEq_T relation because, obviously, the union of two antEq_T relations is an antEq_T relation and $\mathsf{reach}(T)$ is finite. Moreover notice that, given a term $T' \in \mathsf{reach}(T)$, all terms T_i' (with $i \in \{1, \ldots, n\}$) for which $\mathsf{outUnf}(T') = \mathcal{A}'[\oplus\{l \colon T_i'\}]^{i\in\{1,\ldots,n\}}$ are always such that $T_i' \in \mathsf{reach}(T)$ as well (because $\mathsf{outUnf}(T')$ never unfolds recursions occurring inside terms T_i'). Finally, notice that antEq_T is decidable in that it is a relation over $\mathsf{reach}(T)$, which is a finite set.

Definition Appendix B.3. *antSet_T is the field of antEq_T, that is the set of session types $T' \in \mathsf{reach}(T)$ such that there exists T'' with $(T', T'') \in \mathsf{antEq}_T$ or $(T'', T') \in \mathsf{antEq}_T$.*

Lemma Appendix B.4. *antEq_T is an equivalence relation on antSet_T.*

Proof. The reflexive, symmetric and transitive closure of an antEq_T relation is an antEq_T relation, hence this holds true for antEq_T as well. □

Lemma Appendix B.5. *Let $T' \in \mathsf{reach}(T)$. We have that $\mathsf{antOutInf}(T')$ if and only if $T' \in \mathsf{antSet}_T$.*

Proof. We prove the two implications separately, starting from the *if* part, e.g. by assuming $T' \in \mathsf{antSet}_T$. By Lemma Appendix B.4 we have $T' \,\mathsf{antEq}_T\, T'$. We now prove by induction on m that for every m there exists $l_{i_1} \cdots l_{i_m}$ such that $\mathsf{antOut}(T', l_{i_1} \cdots l_{i_m})$ is defined. If $m = 1$ it is sufficient to consider $l_{i_1} = l$ where $\mathsf{outUnf}(T') = \mathcal{A}'[\oplus\{l \colon T_i'\}]^{i\in\{1,\ldots,n\}}$ (with $\mathcal{A}'$ and T_i' that exist by Definition Appendix B.2). Consider now that $T'' = \mathsf{antOut}(T', l_{i_1} \cdots l_{i_{m-1}})$ is defined.

By Definition 3.8, we have $T'' = \mathcal{A}[T_k]^k$ with $\mathsf{outUnf}(\mathsf{antOut}(T, l_{i_1} \cdots l_{i_{m-2}})) = \mathcal{A}[\oplus\{l_{i_{m-1}} : T_k\}]^k$. As $T' \,\mathsf{antEq}_T\, T'$, we can apply $m-1$ times Definition Appendix B.2 to conclude that $T_i \,\mathsf{antEq}_T\, T_j$, for every $i, j \in 1 \ldots k$. This guarantees the existence of the input contexts $\mathcal{A}^k$, session types T^k_r, and label l such that such that $\mathsf{outUnf}(T_k) = \mathcal{A}^k[\oplus\{l : T^k_r\}]^r$. This implies that it is possible to define $\mathsf{antOut}(T'', l)$ hence also $\mathsf{antOut}(T', l_{i_1} \cdots l_{i_m})$ by taking $l_{i_m} = l$.

We now move to the *only if* part assuming that there exists an infinite label sequence $l_{i_1} \cdots l_{i_n} \cdots$ such that, for every n, $\mathsf{antOut}(T', l_{i_1} \cdots l_{i_n})$ is defined. Let $\mathcal{R}$ be the minimal relation such that $(T', T') \in \mathcal{R}$ and: $\mathsf{outUnf}(\mathsf{antOut}(T', l_{i_1} \cdots l_{i_{n-1}})) = \mathcal{A}[\oplus\{l_{i_n} : T_k\}]^{k \in \{1 \ldots m_n\}}$, for any $n \geq 1$, implies $\forall i, j \in \{1 \ldots m_n\}. (T_i, T_j) \in \mathcal{R}$. We now show that $\mathcal{R}$ above is an antEq_T relation. Considered any (T'', T''') in $\mathcal{R}$, we have that there exists h, with $h \geq 1$, such that, for some $\mathcal{A}'$, $\mathcal{A}''$, we have: $\mathsf{outUnf}(T'') = \mathcal{A}'[\oplus\{l_{i_h} : T'_i\}]^{i \in \{1,\ldots,m'\}}$ and $\mathsf{outUnf}(T''') = \mathcal{A}''[\oplus\{l_{i_h} : T''_j\}]^{j \in \{1,\ldots,m''\}}$, with $(T'_i, T''_j) \in \mathcal{R}$ for all $i \in \{1, \ldots, m'\}$ and $j \in \{1, \ldots, m''\}$. This holds, according to the definition of $\mathcal{R}$: for $(T'', T''') = (T', T')$ by taking $h = 1$ and by observing that pairs $(T'_i, T''_j) \in \mathcal{R}$ because they are added to $\mathcal{R}$ in the case $n = 1$; for any (T'', T''') added to $\mathcal{R}$ in the case n, by taking $h = n + 1$ and by observing that pairs $(T'_i, T''_j) \in \mathcal{R}$ because they are among the pairs that are added to $\mathcal{R}$ in the case $n + 1$. □

Proposition 3.10. *Given a single-out session type T, $\mathsf{reach}(T)$ is finite and it is decidable whether $\mathsf{antOutInf}(T)$.*

Proof. Direct consequence of Lemmas Appendix B.1, Lemma Appendix B.5 and the finiteness of antSet_T . □

Appendix B.4. Proof of Theorem 3.11

Theorem 3.11 states that, for single-out session types T and S, the algorithm provided by $T \leq_{\mathsf{t}} S$ indeed terminates. The proof is based on characterizing terms S'' such that $(T'', S'') \in \Sigma'$ for any judgment $\Sigma' \vdash T' \leq_{\mathsf{t}} S'$ reached by the algorithm. In particular, we show that any such term S'' can be obtained by anticipating a sequence of output labels γ in a term R belonging to the finite set $\mathsf{reach}(S)$. This preliminary result is proved by means of the following lemma and corollary.

Lemma Appendix B.6 states that given a type S' at the right hand side of a judgement reachable during the execution of our algorithm, i.e. $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ for some initial types T and S, the type S' (and the types in $\mathsf{reach}(S')$) can be obtained from the initial type S (or one of the types in $\mathsf{reach}(S)$) by means of anticipation of a sequence of output labels, i.e. for all $Q \in \mathsf{reach}(S')$ there exist $R \in \mathsf{reach}(S)$ and a sequence of labels γ such that $Q = \mathsf{antOut}(R, \gamma)$. There is only one case in which this property is not guaranteed to hold, namely, when the last applied rule in $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ is RecR_2. As a counter-example, consider for instance

$$\begin{array}{lll}
\emptyset & \vdash & \mu\mathbf{t}. \oplus \{l : \oplus\{l : \&\{l' : \mathbf{t}\}\}\} \leq_{\mathsf{t}} \mu\mathbf{t}.\&\{l' : \oplus\{l : \mathbf{t}\}\} \rightarrow^* \\
\Sigma' & \vdash & \oplus\{l : \&\{l' : \mu\mathbf{t}. \oplus \{l : \oplus\{l : \&\{l' : \mathbf{t}\}\}\}\}\} \leq_{\mathsf{t}} \\
 & & \&\{l' : \&\{l' : \oplus\{l : \mu\mathbf{t}.\&\{l' : \oplus\{l : \mathbf{t}\}\}\}\}\}
\end{array}$$

where $\to^*$ denotes application of the sequence of rules RecL, RecR_2, Out and RecR_2. We have that the last type $\&\{l' : \&\{l' : \oplus\{l : \mu\mathbf{t}.\&\{l' : \oplus\{l : \mathbf{t}\}\}\}\}\}$ is different from $\mathsf{antOut}(R, \gamma)$, for every $R \in \mathsf{reach}(\mu\mathbf{t}.\&\{l' : \oplus\{l : \mathbf{t}\}\})$ and every sequence of labels γ.

Lemma Appendix B.6. *Consider two single-out session types T and S. Given a judgement $\Sigma' \vdash T' \leq_{\mathsf{t}} S'$ such that $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{t}} S'$, in such a way that the final rule applied is not RecR_2, we have that for all $Q \in \mathsf{reach}(S')$ there exist $R \in \mathsf{reach}(S)$ and a sequence of labels γ such that $Q = \mathsf{antOut}(R, \gamma)$.*

Proof. By induction on the length of the sequence of rule applications $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{t}} S'$. In the base case we have $S' = S$. Consider now $Q \in \mathsf{reach}(S')$. Obviously $Q = \mathsf{antOut}(Q, \epsilon)$ with $Q \in \mathsf{reach}(S)$ because $\mathsf{reach}(S) = \mathsf{reach}(S')$.

In the inductive case we proceed by case analysis on the last rule application $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \to \Sigma' \vdash T' \leq_{\mathsf{t}} S'$. We have two possible cases:

- We can apply the induction hypotheses on the judgement $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$. Hence for all $Q'' \in \mathsf{reach}(S'')$ there exist $R \in \mathsf{reach}(S)$ and a sequence of labels γ such that $Q'' = \mathsf{antOut}(R, \gamma)$. Consider now $Q \in \mathsf{reach}(S')$. We proceed by cases on the applied rule.

  For the rules In, RecR_1 and Out with $\mathcal{A} = [\,]^1$ we have that $S' \in \mathsf{reach}(S'')$ hence also $Q \in \mathsf{reach}(S'')$ because if $S' \in \mathsf{reach}(S'')$ then $\mathsf{reach}(S') \subseteq \mathsf{reach}(S'')$ by definition of $\mathsf{reach}(\_)$.

  If the rule is Out with $\mathcal{A} \neq [\,]^1$ we have that $S' = \mathsf{antOut}(R, \gamma \cdot l)$ with $R \in \mathsf{reach}(S)$ and γ such that $S'' = \mathsf{antOut}(R, \gamma)$ and l is the label of the anticipated output. We limit our analysis to the case in which $Q \notin \mathsf{reach}(S'')$ (in the other cases we can proceed as above). This happens if Q is obtained by applying rule 2. of Definition 3.9 to remove some but not all the inputs in front of one of the output anticipated in S''. Consider now the term V being like Q but without the l output anticipation. Formally, V is defined as follows. Denoted S'' with $\mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^k$ we know that for all k there exists a j_k such that $l_{j_k} = l$. This means that S' is $\mathcal{A}[S_{kj_k}]^k$. Hence, being Q reachable from S' by consuming some inputs of the input context $\mathcal{A}$ only, we have that there exists $\mathcal{A}'$ such that Q is $\mathcal{A}'[S'_{hj'_h}]^h$, where, considered the hole k corresponding to the hole h, we have that $j'_h = j_k$ and $S'_{hj'_h} = S_{kj_k}$. Therefore, the previously mentioned term V is $\mathcal{A}'[\oplus\{l_j : S'_{hj}\}_{j \in J_h}]^h$, where, considered k corresponding to h, we have that $J_h = J_k$. We conclude by observing that $V \in \mathsf{reach}(S'')$, hence there exist $R' \in \mathsf{reach}(S)$ and γ' such that $V = \mathsf{antOut}(R', \gamma')$. But $Q = \mathsf{antOut}(R', \gamma' \cdot l)$, hence proving the thesis.

- We cannot apply the induction hypotheses on the judgement $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ because the rule used to obtain $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ is RecR_2. As RecR_2 cannot be applied in sequence, it is surely possible to apply the induction hypothesis on the previous judgement $\Sigma''' \vdash T''' \leq_{\mathsf{t}} S'''$ such that $\Sigma''' \vdash T''' \leq_{\mathsf{t}} S''' \to$

$\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$. Then we have that for all $Q''' \in \mathsf{reach}(S''')$ we have $Q''' = \mathsf{antOut}(R, \gamma)$ with $R \in \mathsf{reach}(S)$ and a sequence of labels γ. We also have that the rule applied in $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \to \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ is Out, which is the only rule that can applied after RecR_2. Let l be the label of the output involved in the application of the Out rule. Consider now $Q \in \mathsf{reach}(S')$. We consider two possible cases:

- Q is obtained from S' by consuming inputs present in the input context $\mathcal{A}$ used in the last application of the rule Out. Consider now Q''' obtained from S''' by consuming the same inputs and performing the needed unfoldings. Obviously $Q''' \in \mathsf{reach}(S''')$: hence, by induction hypothesis, $Q''' = \mathsf{antOut}(R, \gamma)$ with $R \in \mathsf{reach}(S)$. We have $Q = \mathsf{antOut}(R, \gamma \cdot l)$ hence proving the thesis.
- Q is obtained from S' by consuming strictly more than a sequence of inputs present in the input context $\mathcal{A}$ used in the last application of the rule Out. This means that $Q \in \mathsf{reach}(W)$ where W is a term starting with an output that populates one of the holes of $\mathcal{A}$ in S''. But the terms starting with an output that can occur in S'', assuming $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$, are already in $\mathsf{reach}(S)$. In fact the rules do not perform transformations under outputs, excluding those strictly performed by top level unfoldings. Hence $W \in \mathsf{reach}(S)$, which implies $Q \in \mathsf{reach}(S)$ from which the thesis trivially follows (because $Q = \mathsf{antOut}(Q, \epsilon)$). $\square$

Corollary Appendix B.7. *Consider two single-out session types T and S. Given a judgement $\Sigma' \vdash T' \leq_{\mathsf{t}} S'$ such that $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{t}} S'$ and a pair $(T'', S'') \in \Sigma'$, we have that $S'' = \mathsf{antOut}(R, \gamma)$ for some $R \in \mathsf{reach}(S)$ and a sequence of labels γ.*

Proof. Let $(T'', S'') \in \Sigma'$ and consider the sequence of rule applications $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ that preceeds the application of the rule that introduces (T'', S'') in Σ'. Such rule must be one of RecL, RecR_1 or RecR_2: hence on the judgement $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ it is possible to apply one of these three rules. Since after the application of a rule RecR_2 the uniqe applicable rule is Out, we have the guarantee that the last rule in $\emptyset \vdash T \leq_{\mathsf{t}} S \to^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ is not RecR_2. Hence it is possible to apply Lemma Appendix B.6, from which the thesis directly follows. $\square$

Theorem 3.11. *Given two single-out session types T and S, the algorithm applied to the initial judgement $\emptyset \vdash T \leq_{\mathsf{t}} S$ terminates.*

Proof. Assume by contraposition that there exists single-out session types T and S such that the algorithm applied to the initial judgement $\emptyset \vdash T \leq_{\mathsf{t}} S$ does not terminate. This means that there exists an infinite sequence of rule applications $\emptyset \vdash T \leq_{\mathsf{t}} S \to \Sigma_1 \vdash T_1 \leq_{\mathsf{t}} S_1 \to^* \Sigma_i \vdash T_i \leq_{\mathsf{t}} S_i \to^*$. Within this infinite sequence, there are infinitely many applications of the unfolding rules RecL, RecR_1 or RecR_2, that implies the existence of infinitely many distinct pairs (T_j, S_j) that are introduced in the environment (assuming that j ranges over the instances of application of such rules). All these pairs are distinct, otherwise the precedence of the Asmp rule would have blocked the algorithm. It is obvious that the distinct r.h.s. T_j are finitely many, because every $T_j \in \mathsf{reach}(T)$, which is a finite set. On the contrary, the distinct S_j are infinitely many, but Corollary Appendix B.7 guarantees that for each of them, there exists $S'_j \in \mathsf{reach}(S)$ and a sequence of labels γ_j such that $S_j = \mathsf{antOut}(S'_j, \gamma_j)$.

Due to the finiteness of the possible T_j and S'_j, there exists T'' and S'' such that there exists an infinite subsequence of $(T_{j_1}, S_{j_1}), (T_{j_2}, S_{j_2}), \ldots, (T_{j_k}, S_{j_k}), \ldots$ such that $T_{j_i} = T''$ and $S_{j_i} = \mathsf{antOut}(S'', \gamma_{j_i})$. It is not restrictive to consider $j_h < j_{h+1}$ for every h. The presence of infinitely many distinct γ_{j_i} for which $\mathsf{antOut}(S'', \gamma_{j_i})$ is defined, guarantees $\mathsf{antOutInf}(S'')$. Moreover, this guarantees also the possibility to define an infinite subsequence $(T_{j_{l_1}}, S_{j_{l_1}}), (T_{j_{l_2}}, S_{j_{l_2}}), \ldots,$ $(T_{j_{l_k}}, S_{j_{l_k}}), \ldots$ such that $|\gamma_{j_{l_i}}| < |\gamma_{j_{l_{i+1}}}|$. We now consider the leaf sets $\mathsf{leafSet}(S_{j_{l_i}})$. These sets are defined on a finite domain because the subterms of such types starting with a recursive definition or an output, and preceded by inputs only, are taken from $\mathsf{reach}(S)$. This because the algorithm does not apply transformations under recursive definitions or outputs, excluding the effect of the standard top level unfolding of previous recursive definitions, which is considered in the definition of $\mathsf{reach}(S)$. Hence there are only finitely many distinct $\mathsf{leafSet}(S_{j_{l_i}})$, that guarantees the existence of $v < w$ such that $\mathsf{leafSet}(S_{j_{l_v}}) = \mathsf{leafSet}(S_{j_{l_w}})$. Consider now the judgement $\Sigma_{j_{l_w}} \vdash T_{j_{l_w}} \leq_{\mathsf{t}} S_{j_{l_w}}$. We know that $(T_{j_{l_v}}, S_{j_{l_v}}) \in \Sigma_{j_w}$, $T_{j_{l_v}} = T_{j_{l_w}}$, $S_{j_{l_v}} = \mathsf{antOut}(S'', \gamma_{j_{l_v}})$, $S_{j_{l_w}} = \mathsf{antOut}(S'', \gamma_{j_{l_w}})$, $S'' \in \mathsf{reach}(S)$, and $|\gamma_{j_{l_w}}| < |\gamma_{j_{l_v}}|$. Hence it is possible to apply to such judgement the rule Asmp_2. As Asmp_2 has priority, it should be applied on this judgement thus blocking the sequence of rule applications. But this contradicts the initial assumption of non termination of the algorithm. $\square$

Appendix B.5. Proof of Theorem 3.12

The soundness Theorem 3.12 states that the $\leq_{\mathsf{t}}$ algorithm reaches $\rightarrow_{\mathsf{err}}$ if and only if the $\leq_{\mathsf{a}}$ procedure does so. This is proved in the following by resorting to some preliminary definitions and results.

In Definition Appendix B.2 we have defined the relation antEq_T among types that have the same infinite sequence of outputs that can be anticipated. But, in order to have a decidable relation, we had to limit to types belonging to the set $\mathsf{reach}(T)$. Now, we define a more general relation $\mathsf{extAntEq}_T$ applicable to types having (once unfolded) the following shape: any possible input context with holes filled with single outputs having a continuation belonging to $\mathsf{reach}(T)$. This extension of the antEq_T is necessary because the execution of the subtyping algorithm can generate new terms (as a consequence of output anticipations) having this specific shape.

Definition Appendix B.8. *Let T', T'' be single-out session types. We say that $T' \,\mathsf{extAntEq}_T\, T''$ if there exist $l, \mathcal{A}', \mathcal{A}''$ such that $\mathsf{outUnf}(T') = \mathcal{A}'[\oplus\{l : T'_i\}]^{i\in\{1,\ldots,n\}}$ and $\mathsf{outUnf}(T'') = \mathcal{A}''[\oplus\{l : T''_j\}]^{j\in\{1,\ldots,m\}}$, with $T'_i \,\mathsf{antEq}_T\, T''_j$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.*

Moreover, $\mathsf{extAntSet}_T$ *is the field of* $\mathsf{extAntEq}_T$ *.*

Notice that, all terms T'_i, with $i \in \{1, \dots, n\}$ and T''_j, with $j \in \{1, \dots, m\}$, are in $\mathsf{antSet}_T \subseteq \mathsf{reach}(T)$. Moreover, notice that $\mathsf{extAntEq}_T$ is obviously an equivalence relation on $\mathsf{extAntSet}_T$.

Lemma Appendix B.9. *Let $T' \in \mathsf{antSet}_T$ and $T'' = \mathsf{antOut}(T', \gamma)$ for some γ. We have that $T'' \in \mathsf{extAntSet}_T$.*

Proof. We have to show that there exist $l, \mathcal{A}$ for which we have $\mathsf{outUnf}(\mathsf{antOut}(T', \gamma)) = \mathcal{A}[\oplus\{l \colon T_i\}]^{i \in \{1,\dots,m\}}$, with $T_i \, \mathsf{antEq}_T \, T_j$ for all $i, j \in \{1, \dots, m\}$. We denote $\gamma l = l_{i_1} \cdots l_{i_h}$, with $h \geq 1$. For any n, with $1 \leq n \leq h$, considered $\mathcal{A}'$ and terms T_k with $k \in \{1 \dots m_n\}$ such that $\mathsf{outUnf}(\mathsf{antOut}(T', l_{i_1} \cdots l_{i_{n-1}})) = \mathcal{A}'[\oplus\{l_{i_n} \colon T_k\}]^{k \in \{1 \dots m_n\}}$, we have that $\forall i, j \in \{1 \dots m_n\}.\, T_i \, \mathsf{antEq}_T \, T_j$. This is easily shown by induction on n, applying the definition of antEq_T (the base case is directly derived from $T' \, \mathsf{antEq}_T \, T'$). The case $n = h$ yields the desired result. $\square$

Lemma Appendix B.10. *Let $T', T'' \in \mathsf{extAntSet}_T$ and $\mathsf{leafSet}(T') = \mathsf{leafSet}(T'')$. We have that $T' \, \mathsf{extAntEq}_T \, T''$.*

Proof. It is easy to see that $\mathsf{leafSet}(T') = \mathsf{leafSet}(T'')$ implies $\mathsf{leafSet}(\mathsf{outUnf}(T')) = \mathsf{leafSet}(\mathsf{outUnf}(T''))$. This because $\mathsf{outUnf}()$ causes a leaf T''' belonging to both $\mathsf{leafSet}(T')$ and $\mathsf{leafSet}(T'')$ to yield the same new set of leaves $\mathsf{leafSet}(T''')$ in both T' and T''. By definition of $\mathsf{extAntSet}_T$ we have that exist $l', \mathcal{A}'$ such that $\mathsf{outUnf}(T') = \mathcal{A}'[\oplus\{l' \colon T'_i\}]^{i \in \{1,\dots,n\}}$, with $T'_i \, \mathsf{antEq}_T \, T'_j$ for all $i, j \in \{1, \dots, n\}$. Similarly, there exist $l'', \mathcal{A}''$ such that $\mathsf{outUnf}(T'') = \mathcal{A}''[\oplus\{l'' \colon T''_j\}]^{j \in \{1,\dots,m\}}$, with $T''_i \, \mathsf{antEq}_T \, T''_j$ for all $i, j \in \{1, \dots, m\}$. From the fact that $\mathsf{leafSet}(\mathsf{outUnf}(T')) = \mathsf{leafSet}(\mathsf{outUnf}(T''))$ we have that $l' = l''$ and that: for all T'_i, with $i \in \{1, \dots, n\}$, there exists T''_j, with $j \in \{1, \dots, m\}$, such that $T'_i = T''_j$; and, vice versa, for all T''_j, with $j \in \{1, \dots, m\}$, there exists T'_i, with $i \in \{1, \dots, n\}$, such that $T''_j = T'_i$. Therefore we conclude that $T' \, \mathsf{extAntEq}_T \, T''$. $\square$

In order to prove Theorem 3.12 we also need to consider a simplified subtyping procedure, whose judgments are denoted by $\leq_{\mathsf{sa}}$, and the notions of *IO steps*, *blocking judgments* and *blocking paths*.

Simplified Subtyping Procedure. We here denote by $\leq_{\mathsf{sa}}$ the judgements of the subtyping procedure that is defined exactly as our procedure (defined in Section 3.1 and based on applications of the rules therein over judgments of the form $\Sigma \vdash T \leq_{\mathsf{a}} S$) with the only difference that the Asmp rule is removed (i.e. the subtyping procedure whose transitions were denoted by $\rightarrow_{noAsmp}$ in the proof of Proposition 3.4). Since, in the absence of the Asmp rule the content of environment Σ is never accessed for reading, it has no actual effect on the procedure (on rule applications) and can be removed as well, together with updates on such environment made by the rules. As a consequence we will denote $\leq_{\mathsf{sa}}$ judgments just by $\vdash T \leq_{\mathsf{sa}} S$ for some T and S. Here, differently from the $\rightarrow_{noAsmp}$ notation used in the proof of Proposition 3.4, since we adopt a new notation for judgements, we will simply use: $\vdash T \leq_{\mathsf{sa}} S \rightarrow \vdash T' \leq_{\mathsf{sa}} S'$ to denote that the latter can be obtained from the former by one rule application. Finally, as usual, $\vdash T \leq_{\mathsf{sa}} S \rightarrow_{\mathsf{err}}$ denotes that there is no rule that can be applied to the judgement $\vdash T \leq_{\mathsf{sa}} S$.

Definition Appendix B.11. *A* blocking judgment $\vdash T \leq_{\mathsf{sa}} S$, *denoted by* $\vdash T \leq_{\mathsf{sa}} S \rightarrow_{\mathsf{blk}}$, *is a judgment such that, for some* T', S' *we have:* $\vdash T \leq_{\mathsf{sa}} S \rightarrow^* \vdash T' \leq_{\mathsf{sa}} S' \rightarrow_{\mathsf{err}}$ *by applying rules* RecL, RecR_1 *and* RecR_2 *only.*

Definition Appendix B.12. *An* IO step a, *denoted by* $\xrightarrow{a}_{\mathsf{io}}$, *with* $a \in \{\&_l, \oplus_l \mid l \in L\}$ *is a sequence of* $\leq_{\mathsf{sa}}$ *rule applications* $\rightarrow^*$ *such that the last applied rule is an* In *(in the case* $a = \&_l$*, where* l *is the input label singling out which of the rule premises we consider), or an* Out *rule (in the case* $a = \oplus_l$*, where* l *is the output label singling out which of the rule premises we consider) and all other rule applications concern* RecL, RecR_1 *and* RecR_2 *rules only.*

Definition Appendix B.13. $a_1 \dots a_n$, *with* $n \geq 0$, *is a* blocking path *for judgment* $\vdash T \leq_{\mathsf{sa}} S$ *if there exist* T', S' *such that* $\vdash T \leq_{\mathsf{sa}} S \xrightarrow{a_1}_{\mathsf{io}} \dots \xrightarrow{a_n}_{\mathsf{io}} \vdash T' \leq_{\mathsf{sa}} S' \rightarrow_{\mathsf{blk}}$ *(where* $T' = T$ *and* $S' = S$ *in the case* $n = 0$*).*

Lemma Appendix B.14. *Let* $S \in \mathsf{reach}(Z)$ *and* $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma)$, $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \beta)$ *be such that:* $|\gamma| < |\beta|$ *and* $\mathsf{antOut}(S, \beta) \,\mathsf{extAntEq}_Z\, \mathsf{antOut}(S, \gamma)$*. If* $a_1 \dots a_n$, *with* $n \geq 0$, *is a blocking path for* $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \beta)$ *then there exists an* m *long prefix of* $a_1 \dots a_n$, *with* $0 \leq m \leq n$, *that is a blocking path for* $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma)$.

Proof.

The proof is by induction on $n \geq 0$.

We start by proving the base case $n = 0$. That is $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \rightarrow_{\mathsf{blk}}$, i.e. for some T', S' we have: $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \rightarrow^* \vdash T' \leq_{\mathsf{sa}} S' \rightarrow_{\mathsf{err}}$ by applying rules RecL, RecR_1 and RecR_2 only.

We first observe that $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \xrightarrow{a}_{\mathsf{io}}$ is not possible for any $a \in \{\&_l, \oplus_l \mid l \in L\}$. This because: if we had $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \xrightarrow{\&_l}_{\mathsf{io}}$ for some $l \in L$, then $\mathsf{antOut}(S, \beta) = \&\{l_i : T_i\}_{i \in I}$ with $l = l_i$ for some $i \in I$, hence we would have that also $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \beta) \xrightarrow{\&_l}_{\mathsf{io}}$; and if we had $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \xrightarrow{\oplus_l}_{\mathsf{io}}$ for some $l \in L$, then, since $\mathsf{antOut}(S, \beta) \,\mathsf{extAntEq}_Z\, \mathsf{antOut}(S, \gamma)$, we would have that also $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \beta) \xrightarrow{\oplus_l}_{\mathsf{io}}$.

Therefore, given that it is not possible that $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \rightarrow^* \vdash \mathbf{end} \leq_{\mathsf{sa}} \mathbf{end}$ by applying rules RecL, RecR_1 and RecR_2 only (because otherwise $\mathsf{antOut}(S, \beta)$ would not be defined), we conclude $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma) \rightarrow_{\mathsf{blk}}$ (notice that the number of times a RecL, RecR_1 or RecR_2 is applicable to a judgment is finite because we do not have unguarded recursion and RecR_2 cannot be consecutively applied for more than one time).

We now consider the induction case for blocking path $a_1 \dots a_n$ of length $n \geq 1$.

We first consider the case $a_1 = \&_l$ for some $l \in L$. Given that $\mathsf{antOut}(S, \beta)$ is defined and that $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S, \beta) \xrightarrow{\&_l}_{\mathsf{io}}$, we deduce that $\mathsf{antOut}(S, \gamma)$

is: either $\oplus\{l' : T'\}$ (possibly preceded by some recursion operators), for some l', T'; or $\&\{l_i : T_i\}_{i\in I}$ (possibly preceded by some recursion operators), for some terms T_i and labels l_i such that $l = l_i$ for some $i \in I$. In the first case we have $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma) \rightarrow_{\mathsf{blk}}$, hence the the lemma trivially holds; in the second case we have $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma) \xrightarrow{\&_l}_{\mathsf{io}}$ and we proceed with the proof. We have that there exist T', S', σ such that $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma) \xrightarrow{\&_l}_{\mathsf{io}} T' \leq_{\mathsf{sa}} \mathsf{antOut}(S',\gamma')$ and $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\beta) \xrightarrow{\&_l}_{\mathsf{io}} T' \leq_{\mathsf{sa}} \mathsf{antOut}(S',\beta')$, with $\gamma = \sigma\gamma'$ and $\beta = \sigma\beta'$. In particular S' is obtained from S by removing all its initial (single-)outputs (and intertwined recursions, that are unfolded) until the first input $\&\{l_i : T_i\}_{i\in I}$ is reached, which is also removed, thus yielding $S' = T_i$ for the $i \in I$ such that $l = l_i$. This corresponds, in the definition of $\mathsf{reach}(Z)$ (Definition 3.9), to repeatedly applying, starting from $S \in \mathsf{reach}(Z)$, rules 3 and 4 and finally rule 2, thus yielding $S' \in \mathsf{reach}(Z)$. Notice that σ is the sequence of labels of the initial outputs that were removed during this procedure and that, obviously, $|\gamma| < |\beta|$.

Now, in order to be able to apply the induction hypothesis we have also to show that $\mathsf{antOut}(S,\beta')\,\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\gamma')$. We observe that $\mathsf{antOut}(S,\gamma')$ $\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\gamma)$. This holds because $\mathsf{antOut}(S,\gamma)$ is a $\&\{l_i : T_i\}_{i\in I}$ term, with $l = l_i$ for some $i \in I$, possibly preceded by some recursion operators, and from the following observations: obviously, for any t, T'', it holds $\mu\mathbf{t}.T''\,\mathsf{extAntEq}_Z\,T''\{\mu\mathbf{t}.T''/\mathbf{t}\}$; and $\mathsf{leafSet}(T_i) \subseteq \mathsf{leafSet}(\&\{l_i : T_i\}_{i\in I})$. In the same way, we have $\mathsf{antOut}(S,\beta')\,\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\beta)$.

It is therefore possible to apply the induction hypothesis to $T' \leq_{\mathsf{sa}} \mathsf{antOut}(S',\gamma')$ and $T' \leq_{\mathsf{sa}} \mathsf{antOut}(S',\beta')$ that possesses the shorter blocking path $a_2 \dots a_n$.

Finally, we consider the case $a_1 = \oplus_l$ for some $l \in L$. Since $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\beta)$ $\xrightarrow{\oplus_l}_{\mathsf{io}}$ and $\mathsf{antOut}(S,\beta)\,\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\gamma)$, we have that also $\vdash T \leq_{\mathsf{sa}}$ $\mathsf{antOut}(S,\gamma) \xrightarrow{\oplus_l}_{\mathsf{io}}$. In particular, we have that there exists T' such that $\vdash$ $T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma) \xrightarrow{\oplus_l}_{\mathsf{io}} T' \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma l)$ and $\vdash T \leq_{\mathsf{sa}} \mathsf{antOut}(S,\beta) \xrightarrow{\oplus_l}_{\mathsf{io}} T'$ $\leq_{\mathsf{sa}} \mathsf{antOut}(S,\beta l)$, where, obviously, $|\gamma l| < |\beta l|$. Moreover, since $\mathsf{antOut}(S,\beta)$ $\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\gamma)$ it is immediate to show (by applying the definitions of antOut, $\mathsf{extAntEq}$ and antEq) that also $\mathsf{antOut}(S,\beta l)\,\mathsf{extAntEq}_Z\,\mathsf{antOut}(S,\gamma l)$.

It is therefore possible to apply the induction hypothesis to $T' \leq_{\mathsf{sa}} \mathsf{antOut}(S,\gamma l)$ and $T' \leq_{\mathsf{sa}} \mathsf{antOut}(S,\beta l)$ that possesses the shorter blocking path $a_2 \dots a_n$. $\square$

Theorem 3.12. *Given two single-out session types T and S, we have that there exist Σ', T', S' such that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ if and only if there exist Σ'', T'', S'' such that $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$.*

Proof. We consider the two implications separately starting from the *if* part. Assume that $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$. In this sequence of rule applications, the new rule $\mathsf{Asmp2}$ is never used otherwise the sequence would terminate successfully by applying such a rule. Hence, by applying the same sequence of rules, we have $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ with $T'' = T'$, $S'' = S'$ and $\Sigma'' = \Sigma'$. We have that $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$, otherwise if a rule could be applied to this judgement, the same rule could be applied also to $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S''$ thus contradicting the assumption $\Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$.

We now move to the *only if* part. Assume that $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ and that, by contradiction, $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$ does not hold.

From $\emptyset \vdash T \leq_{\mathsf{a}} S \rightarrow^* \Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ (since in this sequence of rule applications the Asmp rule is never used, otherwise the sequence would terminate successfully by applying such a rule), by applying the same sequence of rules, we have $\vdash T \leq_{\mathsf{sa}} S \rightarrow^* \vdash T' \leq_{\mathsf{sa}} S'$.

We now observe that, since we assumed (by contradiction) that we do not get the error when using the $\leq_{\mathsf{t}}$ procedure, there must exist at least a triple Σ''', T''', S''' such that: $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma''' \vdash T''' \leq_{\mathsf{t}} S'''$ (and correspondingly $\vdash T \leq_{\mathsf{sa}} S \rightarrow^* \vdash T''' \leq_{\mathsf{sa}} S'''$ because the Asmp and $\mathsf{Asmp2}$ rules, that would have led to successful termination, cannot have been applied), $\Sigma''' \vdash T''' \leq_{\mathsf{t}} S'''$ successfully terminates by applying the Asmp or $\mathsf{Asmp2}$ rule, and $\vdash T''' \leq_{\mathsf{sa}} S'''$ has a blocking path.

Let us now consider one of such triples Σ''', T''', S''' (possessing the above stated properties) that has a blocking path of minimal length, i.e. there is no other Σ''', T''', S''' triple of the kind above such that $\vdash T''' \leq_{\mathsf{sa}} S'''$ has a shorter blocking path. Let $a_1 \ldots a_n$ be such a path. Since the Asmp or $\mathsf{Asmp2}$ rule is applied to $\Sigma''' \vdash T''' \leq_{\mathsf{t}} S'''$, we have $S''' = \mathsf{antOut}(S, \beta)$ (in the case of Asmp this is obtained by Corollary Appendix B.7).

We now consider γ such that $(T''', \mathsf{antOut}(S, \gamma)) \in \Sigma'''$ was used in the premise of Asmp or $\mathsf{Asmp2}$ rule: $\gamma = \beta$ in the case of the Asmp rule, $|\gamma| < |\beta|$ in the case of the $\mathsf{Asmp2}$ rule. Moreover, let us also consider Σ_γ to be the environment such that $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma_\gamma \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \gamma)$, where $\Sigma_\gamma \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \gamma)$ is the judgment to which the rule that caused $(T''', \mathsf{antOut}(S, \gamma))$ to be inserted in the environment was applied.

We now observe that there exists a m long prefix of $a_1 \ldots a_n$, with $0 \leq m \leq n$, that is a blocking path for $\vdash T''' \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma)$. This is obvious in the case $\gamma = \beta$; it is due to Lemma Appendix B.14 in the case $|\gamma| < |\beta|$: we obtain $\mathsf{antOut}(S, \beta)\, \mathsf{extAntEq}_Z\, \mathsf{antOut}(S, \gamma)$ as needed by such a Lemma from the statements in the premise of rule $\mathsf{Asmp2}$ and by applying Lemmas Appendix B.5, Appendix B.9 and Appendix B.10.

Since we assumed (by contradiction) that $\emptyset \vdash T \leq_{\mathsf{t}} S \rightarrow^* \Sigma'' \vdash T'' \leq_{\mathsf{t}} S'' \rightarrow_{\mathsf{err}}$ does not hold, this would be possible only if there existed a triple Σ'''', T'''', S'''' such that: there is a sequence of rule applications $\Sigma_\gamma \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \gamma) \rightarrow^* \Sigma'''' \vdash T'''' \leq_{\mathsf{t}} S''''$ that is a prefix of the sequence of rule applications of the blocking path for $\vdash T''' \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma)$; and $\Sigma'''' \vdash T'''' \leq_{\mathsf{t}} S''''$ successfully terminates by applying the Asmp or $\mathsf{Asmp2}$ rule. Notice that such a sequence $\Sigma_\gamma \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \gamma) \rightarrow^* \Sigma'''' \vdash T'''' \leq_{\mathsf{t}} S''''$ should necessarily include the application of, at least, an In rule (causing the algorithm to branch), because otherwise (given that $\Sigma_\gamma \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \gamma) \rightarrow^* \Sigma''' \vdash T''' \leq_{\mathsf{t}} \mathsf{antOut}(S, \beta)$) we could not have that $\Sigma'''' \vdash T'''' \leq_{\mathsf{t}} S''''$ successfully terminates by applying the Asmp or $\mathsf{Asmp2}$ rule.

However the existence of such a triple Σ'''', T'''', S'''' is not possible, because $\vdash T'''' \leq_{\mathsf{sa}} S''''$ would have a k long blocking path with $k < n$ (being such a path strictly shorter than that of $\vdash T''' \leq_{\mathsf{sa}} \mathsf{antOut}(S, \gamma)$), thus violating the minimality assumption about the blocking path length of the Σ''', T''', S''' triple. □

Appendix C. Proofs of Section 4

Appendix C.1. Proof of Lemma 4.4 and Theorem 4.5

We here prove Lemma 4.4 and Theorem 4.5 that show undecidability of $\leq_{\mathsf{bound}}$ by reduction from the bounded non termination problem.

Lemma 4.4. *Given a queue machine M and an input x, it is undecidable whether M does not terminate and is bound on x.*

Proof. We first prove that boundedness is undecidable. If, by contraposition, boundedness was decidable, termination could be decided by first checking boundedness, and then perform a finite state analysis of the queue machine behaviour. More precisely, termination on bounded queue machines can be decided by forward exploration of the reachable configurations until a terminating configuration is found, or a cycle is detected by reaching an already visited configuration.

We now conclude by observing that given a queue machine M and the input x, it is not possible to decide whether M *does not terminate and is bound on* x. Assume by contraposition one could decide the above property of queue machines. Then boundedness could be decided as follows: transform M in a new machine M' that behaves like M plus an additional special symbol # which is enqueued every time it is dequeued; boundedness of M on input x can be decided by checking the above property on M' and input $\#x$ (in fact M' never terminates and is bound on $\#x$ if and only if M is bound on x). □

Theorem 4.5. *Given a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string x, we have that $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$ if and only if M does not terminate and is bound on x.*

Proof. We need a preliminary result: given $(q, \gamma) \to_M (q', \gamma')$, if $[\![q]\!]^{\emptyset} \leq [\![\gamma]\!]$ then we also have that $[\![q']\!]^{\emptyset} \leq [\![\gamma']\!]$. In fact, assuming $\gamma = C_1 \cdots C_m$ and $\delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1})$, we have $\gamma' = C_2 \cdots C_m B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}$. Having $[\![q]\!]^{\emptyset} \leq [\![\gamma]\!]$, by one application of item 4. of Definition 2.4, one application of item 3., and n_{C_1} applications of item 2., we can conclude that $[\![q']\!]^{\emptyset} \leq [\![\gamma']\!]$.

We now observe that if M is not bound on x we have that it is not possible to have $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$. Assume by contraposition that $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$. From the previous preliminary result, we have that also $[\![q']\!]^{\emptyset} \leq_{\mathsf{bound}} [\![\gamma']\!]$ for each reachable configuration (q', γ'). But due to unboundedness of M on x we have that, for every k, there is an enqueue operation that is executed when the queue is longer than k. Assume this happens when the configuration (q', γ') performs its computation action. In order to relate $[\![q']\!]^{\emptyset}$ and $[\![\gamma']\!]$, we need a relation that contains pairs with the l.h.s. starting with an output and the r.h.s. with an input context of depth greater than k. But this cannot hold if we fix a maximal depth smaller than k to the input context.

Now we observe that $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$ if and only if M does not terminate and is bound on x. Following the *(Only if part)* of the proof of Theorem 3.1 [10] stating the undecidability of $\ll$, we prove that if $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$ then M does not terminate. Moreover, we also have that M is bound on x in the light of the previous observation.

Consider now that M does not terminate. As in the *(If part)* of the same proof mentioned above, we define $\mathcal{C} = \{(q_i, \gamma_i) \mid (s, x\$) = (q_0, \gamma_0) \rightarrow_M (q_1, \gamma_1) \rightarrow_M \cdots \rightarrow_M (q_i, \gamma_i), i \geq 0\}$ and the following relation $\mathcal{R}$ on types:

$$
\begin{array}{ll}
\mathcal{R} & = \\
\{ & \big([\![q]\!]^{\emptyset}, [\![C_1 \cdots C_m]\!] \big), \\
& \big(\&\{A \colon \oplus\{B_1^A \colon \cdots \oplus \{B_{n_A}^A \colon [\![q']\!]^{\emptyset}\}\}\}_{A \in \Gamma} \ , \ [\![C_1 \cdots C_m]\!] \big), \\
& \big(\oplus \{B_1^{C_1} \colon \oplus\{B_2^{C_1} \colon \cdots \oplus \{B_{n_{C_1}}^{C_1} \colon [\![q']\!]^{\emptyset}\}\}\} \ , \ \&\{C_2 \colon \cdots \&\{C_m \colon Z\}\} \big), \\
& \big(\oplus \{B_2^{C_1} \colon \cdots \oplus \{B_{n_{C_1}}^{C_1} \colon [\![q']\!]^{\emptyset}\}\} \ , \ \&\{C_2 \colon \cdots \&\{C_m \colon \&\{B_1^{C_1} \colon Z\}\}\} \big), \\
& \cdots \\
& \big([\![q']\!]^{\emptyset} \ , \ \&\{C_2 \colon \cdots \&\{C_m \colon \&\{B_1^{C_1} \colon \cdots \&\{B_{n_{C_1}}^{C_1} \colon Z\}\}\}\} \big) \\
| & (q, C_1 \cdots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}), \\
& Z = \mu \mathbf{t}. \oplus \{A \colon \&\{A \colon \mathbf{t}\}\}_{A \in \Gamma} \quad \}
\end{array}
$$

Following the proof of Theorem 3.1 [10] we show that this relation is an asynchronous subtyping relation. Moreover boundedness of M on x guarantees boundedness on the length of the reachable queue contents $C_1 \cdots C_m$, that implies boundedness of the depth of the input contexts of the r.h.s. of all the pairs in $\mathcal{R}$. This proves that $[\![s]\!]^{\emptyset} \leq_{\mathsf{bound}} [\![x\$]\!]$. $\square$

Appendix C.2. Proof of Theorem 4.8

Theorem 4.8 states that, given a single consuming queue machine M and an input x, termination of M on x is undecidable. The theorem is proved by resorting to Turing completeness of queue machines. In order to do this we preliminarily provide an encoding $[\![M]\!]$ from a queue machine M into a single-consuming queue machine and two lemmas that guarantee that, given a queue machine M and an input x, M terminates on x if and only if the single-consuming queue machine $[\![M]\!]$ terminates on x.

Definition Appendix C.1. *Let $M = (\{q_1, .., q_n\}, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $\#$ be a special character not in Γ. We denote with $[\![M]\!]$ the following single-consuming queue machine $(\{q_1, .., q_n, q'_1, .., q'_n\}, \Sigma, \Gamma \cup \{\#\}, \$, s, \delta')$ with δ' defined as follows:*

- $\delta'(q_i, a) = (q'_j, \epsilon)$ *if* $\delta(q_i, a) = (q_j, \epsilon)$
- $\delta'(q_i, a) = (q_j, \gamma)$ *if* $\delta(q_i, a) = (q_j, \gamma)$ *with* $\gamma \neq \epsilon$
- $\delta'(q_i, \#) = (q'_i, \epsilon)$
- $\delta'(q'_i, a) = (q_j, \#)$ *if* $\delta(q_i, a) = (q_j, \epsilon)$
- $\delta'(q'_i, a) = (q_j, \gamma)$ *if* $\delta(q_i, a) = (q_j, \gamma)$ *with* $\gamma \neq \epsilon$
- $\delta'(q'_i, \#) = (q_i, \#)$

Given a configuration (q, γ) of $[\![M]\!]$, we denote with $\{\!\{(q,\gamma)\}\!\}$ the configuration (z, β) where $z = q$, if $q \in \{q_1, .., q_n\}$, or $z = q_i$, if $q = q'_i$, while β is obtained from γ by removing each instance of the special symbol $\#$.

Example Appendix C.2. *We now comment the construction $[\![M]\!]$ that, given a queue machine M, returns a single-consuming queue machine. As an example, consider $M = (\{q_1, q_2\}, \{a\}, \{a, \$\}, \$, q_1, \delta)$ with δ such that $\delta(q_1, a) = (q_2, \epsilon)$, $\delta(q_1, \$) = (q_1, \$)$, $\delta(q_2, a) = (q_2, a)$, and $\delta(q_2, \$) = (q_2, \epsilon)$. This machine accepts the input string "a" by consuming in sequence a and then \$. Consider now $[\![M]\!] = (\{q_1, q_2, q'_1, q'_2\}, \{a\}, \{a, \$, \#\}, \$, q_1, \delta')$ with δ' such that $\delta'(q_1, a) = (q'_2, \epsilon)$, $\delta'(q_1, \$) = (q_1, \$)$, $\delta'(q_1, \#) = (q'_1, \epsilon)$, $\delta'(q_2, a) = (q_2, a)$, $\delta'(q_2, \$) = (q'_2, \epsilon)$, $\delta'(q_2, \#) = (q'_2, \epsilon)$, $\delta'(q'_1, a) = (q_2, \#)$, $\delta'(q'_1, \$) = (q_1, \$)$, $\delta'(q'_1, \#) = (q_1, \#)$, $\delta'(q'_2, a) = (q_2, a)$, $\delta'(q'_2, \$) = (q'_2, \#)$, and $\delta'(q'_2, \#) = (q_2, \#)$. This new queue machine also accepts the input string "a" but it does simply consume a and \$ in sequence, but when \$ is dequeued the special symbol \# is enqueued (which is subsequently consumed thus emptying the queue). Notice that the queue machine $[\![M]\!]$ cannot consume two symbol in sequence because after the first one is consumed, it enters in one of the primed state q'_i that always enqueue some symbol.*

Lemma Appendix C.3. *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $x \in \Sigma^*$. If $(s, x\$) \rightarrow^*_M (q, \gamma)$ then there exists a configuration (q', γ') such that $(s, x\$) \rightarrow^*_{[\![M]\!]} (q', \gamma')$ with $\{\!\{(q', \gamma')\}\!\} = (q, \gamma)$.*

Proof. By induction on the number of steps in the sequence $(s, x\$) \rightarrow^*_M (q, \gamma)$. The base case is trivial. In the inductive case we perform a case analysis. The unique non trivial case is when the configuration reached by $[\![M]\!]$ according to the inductive hypothesis has the queue starting with the special symbol #. In this case, $[\![M]\!]$ must perform more transitions, first to consume all the instances of # in front of the queue and then to mimick the new transition of M. □

Lemma Appendix C.4. *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $x \in \Sigma^*$. If $(s, x\$) \rightarrow^*_{[\![M]\!]} (q, \gamma)$ then $(s, x\$) \rightarrow^*_M \{\!\{(q, \gamma)\}\!\}$.*

Proof. By induction on the number of steps in the sequence $(s, x\$) \rightarrow^*_{[\![M]\!]} (q, \gamma)$. The base case is trivial. In the inductive case we perform a case analysis. The unique non trivial case is when γ starts with the special symbol #. In this case, M does not perform any new transition as if (q', γ') is the new configuration we have that $\{\!\{(q, \gamma)\}\!\} = \{\!\{(q', \gamma')\}\!\}$. □

Theorem 4.8. *Given a single consuming queue machine M and an input x, the termination of M on x is undecidable.*
Proof. The thesis directly follows from the Turing completeness of queue machines, and the two above Lemmas that guarantee that given a queue machine M and an input x, M terminates on x if and only if the single-consuming queue machine $[\![M]\!]$ terminates on x. This is guaranteed by the fact that if $[\![M]\!]$ reaches a configuration with the queue containing only instances of #, it is guaranteed to eventually terminate by emptying the queue. □

Appendix C.3. Proof of Theorem 4.11

Theorem 4.11 states that a single-consuming queue machine does not terminate if and only if the types obtained by the encoding of Figure 4 are in the $\leq_{\mathsf{tin},\mathsf{tout}}$ relation. The proof is done by separately showing, as preliminary lemmas, both implications (one in each direction) to hold.

Concerning such lemmas and their proof, we need to introduce some preliminary notation. Given a sequence of queue symbols γ, we denote with $[\![\gamma]\!]_u$ the set of session types that can be obtained from $[\![\gamma]\!]$ by independently replacing each occurrence, inside it, of the term T'' defined in Figure 4 with $\mathsf{antOut}(T'', l_{i_1} \ldots l_{i_n})$, for some sequence of labels $l_{i_1} \ldots l_{i_n}$ with $n \geq 0$ (distinguished label sequences can be considered for replacing different occurrences of T'' inside $[\![\gamma]\!]$). Observe that $[\![\gamma]\!]_u$ is well defined because T'' can anticipate every possible sequence of outputs. Moreover, for simplicity, we will consider the asynchronous subtyping relation $\leq$ instead of $\leq_{\mathsf{tin},\mathsf{tout}}$. Nevertheless, we will apply such relation on types that have all their choices labeled on the same set of labels, hence the two relations obviously coincide on such types.

Lemma Appendix C.5. *Given a single-consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, if $[\![s]\!]^{\emptyset} \leq [\![x\$]\!]$ then M does not terminate on x.*

Proof. We need a preliminary result: given $(q, \gamma) \rightarrow_M (q', \gamma')$ and a term $S \in [\![\gamma]\!]_u$, if $[\![q]\!]^{\emptyset} \leq S$ then there exists $S' \in [\![\gamma']\!]_u$ such that $[\![q']\!]^{\emptyset} \leq S'$. In fact, assuming $\gamma = C_1 \cdots C_m$ and $\delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1})$, we have $\gamma' = C_2 \cdots C_m B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}$. Consider now $S \in [\![\gamma]\!]_u$. Having $[\![q]\!]^{\emptyset} \leq S$, by one application of item 4. of Definition 2.4, one application of item 3., and n_A applications of item 2., we can conclude that there exists $S' \in [\![\gamma']\!]_u$ such that $[\![q']\!]^{\emptyset} \leq S'$.

We now prove the thesis by showing that assuming that M accepts x we have $[\![s]\!]^{\emptyset} \not\leq [\![x\$]\!]$. By definition of queue machines, we have that: M accepts x implies $(s, x\$) \rightarrow_M^* (q, \epsilon)$. Assume now, by contraposition, that $[\![s]\!]^{\emptyset} \leq [\![x\$]\!]$. As $(s, x\$) \rightarrow_M^* (q, \epsilon)$, by repeated application of the above preliminary result we have that exists $S' \in [\![\epsilon]\!]_u$ such that $[\![q]\!]^{\emptyset} \leq S'$. But this cannot hold because $[\![q]\!]^{\emptyset}$ is a recursive definition that upon unfolding begins with an input that implies (according to items 4. and 3. of Definition 2.4) that also S' (once unfolded) starts with an input. But this is false, in that, by definition of the queue encoding $[\![\epsilon]\!] = \mu \mathbf{t} \oplus \Big\{A : \&\Big(\{A : \mathbf{t}\} \uplus \{A' : T''\}_{A' \in \Gamma \setminus \{A\}}\Big)\Big\}_{A \in \Gamma}$. $\square$

Lemma Appendix C.6. *Given a single-consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, if M does not terminate on x then $[\![s]\!]^{\emptyset} \leq [\![x\$]\!]$.*

Proof. Assuming that M does not accept x we show that $[\![s]\!]^{\emptyset} \leq [\![x\$]\!]$. When a queue machine does not accept an input, the corresponding computation never ends. In our case, this means that there is an infinite sequence $(s, x\$) =$

$(q_0,\gamma_0) \to_M (q_1,\gamma_1) \to_M \cdots \to_M (q_i,\gamma_i) \to_M \cdots$. Let $\mathcal{C}$ be the set of reachable configurations, i.e. $\mathcal{C} = \{(q_i,\gamma_i) \mid i \geq 0\}$. We now define a relation $\mathcal{R}$ on types, where T' and T'' are as in Figure 4, $T_0 = \oplus\{A : T''\}_{A\in\Gamma}$ and $T_n = \&\{A : T_{n-1}\}_{A\in\Gamma}$:

$$
\begin{array}{lll}
\mathcal{R} & = & \\
& \{ & (\llbracket q \rrbracket^{\emptyset} \ , \ S_{C_1\cdots C_m} \ ), (\ \&\{A : \{\!\{B_1^A \cdots B_{n_A}^A\}\!\}_{q'}^{\emptyset}\}_{A\in\Gamma} \ , \ S_{C_1\cdots C_m} \ ), \\
& & (\ \{\!\{B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset} \ , \ S_{C_2\cdots C_m} \ ), (\ \{\!\{B_2^{C_1} \cdots B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset} \ , \ S_{C_2\cdots C_m B_1^{C_1}} \ ), \\
& & \cdots \\
& & (\ \{\!\{B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset} \ , \ S_{C_2\cdots C_m B_1^{C_1}\cdots B_{n_{C_1}-1}^{C_1}} \ ) \\
& | & (q, C_1 \cdots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}), S_\gamma \in \llbracket \gamma \rrbracket_u \quad \} \\
\bigcup & & \\
& \{ & (\llbracket q \rrbracket^{\emptyset} \ , \ T_n \ ), (\ \&\{A : \{\!\{B_1^A \cdots B_{n_A}^A\}\!\}_{q'}^{\emptyset}\}_{A\in\Gamma} \ , \ T_n \ ), \\
& & (\ \{\!\{B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset} \ , \ T_m \ ), (\ \{\!\{B_2^{C_1} \cdots B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset} \ , \ T_m \ ), \\
& & \cdots \\
& & (\ \{\!\{B_{n_{C_1}}^{C_1}\}\!\}_{q'}^{\emptyset}, T_m \ ) \\
& | & (q, C_1 \cdots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}), \\
& & \text{if } \exists q'', C \text{ s.t. } \delta(q, C) = (q'', \epsilon) \text{ then } n \geq 2 \text{ else } n \geq 1, m \geq 0 \quad \} \\
\bigcup & & \\
& \{ & (T' \ , \ T_n), (\&\{A_1 : \oplus\{A_2 : T'\}_{A_2\in\Gamma}\}_{A_1\in\Gamma} \ , \ T_n), (\oplus \{A_2 : T'\}_{A_2\in\Gamma} \ , \ T_m) \\
& | & n \geq 1, m \geq 0 \quad \} \\
\bigcup & & \\
& \{ & (T' \ , \ S_\gamma), (\&\{A_1 : \oplus\{A_2 : T'\}_{A_2\in\Gamma}\}_{A_1\in\Gamma} \ , \ S_\gamma), (\oplus \{A_2 : T'\}_{A_2\in\Gamma} \ , \ S_\gamma) \\
& | & \gamma \in \Gamma^*, S_\gamma \in \llbracket \gamma \rrbracket_u \quad \}
\end{array}
$$

We have that the above $\mathcal{R}$ is an asynchronous subtyping relation because each of the pairs satisfies the conditions in Definition 2.4 thanks to the presence of other pairs in $\mathcal{R}$. We can conclude observing that $(s, x\$) \in \mathcal{C}$ implies that $(\llbracket q \rrbracket^{\emptyset}, \llbracket x\$ \rrbracket)$ belongs to the above asynchronous subtyping relation $\mathcal{R}$, hence $\llbracket q \rrbracket^{\emptyset} \leq \llbracket x\$ \rrbracket$. □

Theorem 4.11. *Given a single consuming queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ and an input string $x \in \Sigma^*$, we have $\llbracket s \rrbracket^{\emptyset} \leq_{\mathsf{tin},\mathsf{tout}} \llbracket x\$ \rrbracket$ if and only if M does not terminate on x.*

Proof. Direct consequence of Lemmas Appendix C.5 and Appendix C.6. □