

# Undecidability of Asynchronous Session Subtyping

Mario Bravetti

*University of Bologna, Department of Computer Science and Engineering / FOCUS INRIA  
Mura Anteo Zamboni 7, 40126 Bologna, Italy*

Marco Carbone

*Department of Computer Science, IT University of Copenhagen  
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark*

Gianluigi Zavattaro

*University of Bologna, Department of Computer Science and Engineering / FOCUS INRIA  
Mura Anteo Zamboni 7, 40126 Bologna, Italy*

---

## Abstract

Session types are used to describe communication protocols in distributed systems and, as usual in type theories, session subtyping characterizes substitutability of the communicating processes. We investigate the (un)decidability of subtyping for session types in asynchronously communicating systems. We first devise a core undecidable subtyping relation that is obtained by imposing limitations on the structure of types. Then, as a consequence of this initial undecidability result, we show that (differently from what stated or conjectured in the literature) the three notions of asynchronous subtyping defined so far for session types are all undecidable. Namely, we consider the asynchronous session subtyping by Mostrous and Yoshida [1] for binary sessions, the relation by Chen et al. [2] for binary sessions under the assumption that every message emitted is eventually consumed, and the one by Mostrous et al. [3] for multiparty session types. Finally, by showing that two fragments of the core subtyping relation are decidable, we evince that further restrictions on the structure of types make our core subtyping relation decidable.

*Keywords:* Session Types, Subtyping, Undecidability, Queue Machines

---

## 1. Introduction

Session types [4, 5] are types for describing the behaviour of communicating systems, and can be used as specifications of distributed protocols to be checked against implementations. Such check, done by means of a typing system, guarantees that communications at any endpoint of the implemented system are always matched by the corresponding intended partner. As a consequence, it is

ensured that communication errors, e.g., deadlock, will never occur. This approach provides a compositional way of checking the correctness of distributed systems.

As an example, consider a simple on-line shop: clients can buy a list of items by following the protocol expressed by the session type

$$S_{client} = \mu \mathbf{t}. \oplus \{ \text{add\_to\_cart} : \mathbf{t} , \text{pay} : \mathbf{end} \}$$

indicating a recursive behaviour, according to which the client decides whether to add an item and keep interacting with the store, or to pay and conclude the session. For the sake of simplicity we consider session types where (the type of) communicated data is abstracted away.

We call *output selection*<sup>1</sup> the construct  $\oplus \{ l_1 : T_1, \dots, l_n : T_n \}$ . 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 when to stop adding items to the cart and then move to the payment.

The symmetric behaviour of the shopping service is represented by the complementary session type

$$S_{service} = \mu \mathbf{t}. \& \{ \text{add\_to\_cart} : \mathbf{t} , \text{pay} : \mathbf{end} \}.$$

We call *input branching*<sup>2</sup> the construct  $\& \{ l_1 : T_1, \dots, l_n : T_n \}$ . 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 selection.

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 of the on-line shop, the client can be safely replaced by another one with session type

$$T_{client} = \oplus \{ \text{add\_to\_cart} : \oplus \{ \text{pay} : \mathbf{end} \} \}$$

indicating that only one item is added to the shopping cart before paying. But also the shopping service could be safely replaced by another one offering also the `remove_from_cart` functionality:

$$T_{service} = \mu \mathbf{t}. \& \{ \text{add\_to\_cart} : \mathbf{t} , \text{remove\_from\_cart} : \mathbf{t} , \text{pay} : \mathbf{end} \}.$$

---

<sup>1</sup>In session type terminology [4, 5], this construct is usually simply called *selection*; we call it output selection because we consider a simplified syntax for session types in which there is no specific separate construct for sending one output. Anyway, such an output type could be seen as an output selection with only one choice.

<sup>2</sup>In session type terminology this construct is simply called *branching*. We call it input branching for symmetric reasons w.r.t. those discusses in the previous footnote.

Formally, subtyping relations have been defined for session types to precisely capture this safe replacement notion.

Gay and Hole [6] are the first ones who studied subtyping for session types in a context where protocols involve only two peers (i.e. are binary) and communication is synchronous. Later, Mostrous et al. [3] extended this notion to multiparty session types with *asynchronous* communication. Both articles propose an algorithm for checking subtyping, but the one proposed by Mostrous et al. [3], differently from what stated therein, is not always terminating in the sense that there are cases in which it diverges and never gives an answer. An example of divergent execution is discussed in the *Remark* paragraph of §4.4.

Later work by Mostrous and Yoshida [1], Mostrous [7] and Chen et al. [2] addresses subtyping in variants of an asynchronous setting for binary sessions. In particular Chen et al. [2] focus on binary sessions in which messages sent by a partner are guaranteed to be eventually received. Such articles conjecture that an algorithm for checking asynchronous session subtyping exists, although, in his PhD thesis, Mostrous [7] expresses a few doubts about the decidability of asynchronous subtyping (pp. 178-180), because of the need for infinite simulations.

In this work, we prove that the subtyping relations defined by Mostrous and Yoshida [1], Chen et al. [2], and Mostrous et al. [3] are undecidable. We proceed by identifying a core asynchronous subtyping relation and show it is undecidable: all other undecidability results are obtained by reduction from this initial relation.

The core relation, denoted by  $\ll$ , is named asynchronous single-choice relation. Such a relation is obtained by first defining (following the approach by Mostrous and Yoshida [1]) a standard asynchronous subtyping  $\leq$  and then reduce it by imposing additional constraints:  $T$  and  $S$  are in single-choice relation, written  $T \ll S$ , if  $T \leq S$ , all output selections in  $T$  have a single choice (output selections are covariant<sup>3</sup>, thus  $S$  is allowed have output selections with multiple choices), all input branchings in  $S$  have a single choice (input branchings are contravariant<sup>4</sup>, thus  $T$  is allowed to have input branchings with multiple choices), and additionally both  $T$  and  $S$  do not have consecutive infinite output selections. This last condition is added to encompass the subtyping defined by Chen et al. [2] that, as discussed above, requires all messages to be eventually received: in fact, if consecutive infinite output selections are not allowed, it is not possible to indefinitely delay inputs.

For instance, considering the simple on-line shop example, we have:

$$T_{client} \not\ll S_{client}$$

because  $S_{client}$  has consecutive infinite output selections; and

$$T_{service} \not\ll S_{service}$$

---

<sup>3</sup>Covariant means that the bigger type has more choices than the smaller type.

<sup>4</sup>Contravariant means that the bigger type has less choices than the smaller type.

because  $S_{service}$  has input branchings with more than one choice.

If we consider a different behavior for the shopping service, where each input branching has single choice

$$S'_{service} = \&\{\text{add\_to\_cart} : \&\{\text{pay} : \text{end}\}\}.$$

we have, instead,  $T_{service} \ll S'_{service}$ .

The proof of undecidability of  $\ll$  is by reduction from the acceptance problem in queue machines. Queue machines are a Turing powerful computational model composed of a finite control that consumes and introduces symbols in a queue, i.e. a First-In First-Out (FIFO) structure. The input to a queue machine is given by a sequence of symbols, ended by a special delimiter \$, that is initially present in the queue. The finite control is defined by a finite set of states (one of which being the initial state) and a transition function that given the current state and the consumed symbol, i.e. the one taken from the beginning of the queue, returns the next state and a sequence of symbols to be added at the end of the queue. The input is accepted by the queue machine whenever queue is emptied. As an example, one can define a queue machine able to accept the strings  $a^n b^n$ , with  $n \geq 0$ , by considering a finite control with the following states:

- an initial state that expects to consume one among two possible symbols: an  $a$  and then move to the second state, or the delimiter \$ (thus accepting);
- the second state that cyclically consumes each of the remaining symbols  $a$ , re-introducing them at the end of the queue, and then moves to the third state by consuming the first  $b$ ;
- the third state that cyclically consumes each of the remaining symbols  $b$ , re-introducing them at the end of the queue, and then returns to the initial state by consuming the \$ and re-enqueueing it.

Even if queue machines are similar to pushdown-automata, they are strictly more powerful. For instance, it is trivial to extend the above queue machine with an additional state in order to accept strings  $a^n b^n c^n$ , with  $n \geq 0$ . Queue machines are not only more expressive than pushdown-automata, but they are Turing complete. Intuitively, this follows from the fact that by using a queue instead of a stack, it is possible to access any symbol in the structure without losing the symbols in front. In fact, it is sufficient to re-introduce such symbols at the end of the queue, as done in the above example e.g. with the symbols  $a$  and  $b$  in the second and the third state, respectively. This mechanism makes it possible to simulate the tape of a Turing machine by using the queue.

Being Turing powerful, acceptance of a string is undecidable for queue machines. We show (Theorem 3.1) that given a queue machine and an input string, it is always possible to define two session types  $T$  and  $S$  such that  $T \ll S$  if and only if the given input string is not accepted by the considered queue machine. From this we conclude that the  $\ll$  relation is also undecidable.

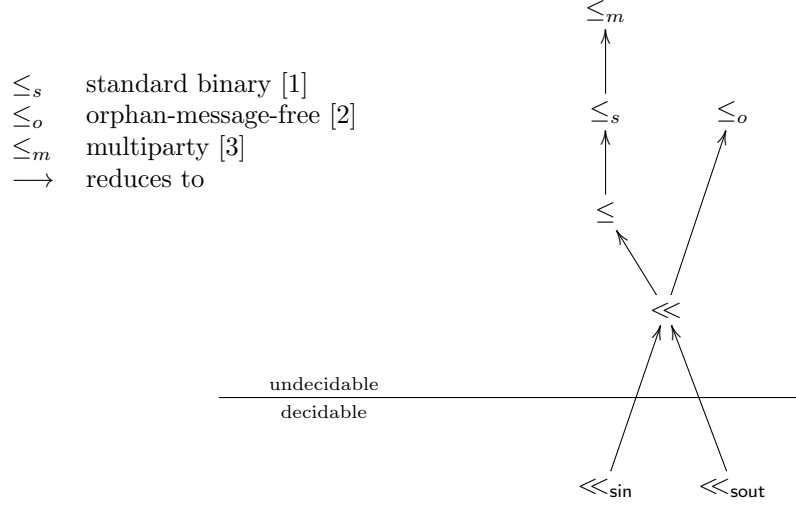


Figure 1: Lattice of the asynchronous subtyping relations considered in this paper.

This core undecidability result allows us to prove by reduction the undecidability of  $\leq$  as well as other more complex relations including the three asynchronous subtypings in the literature discussed above. Namely, we prove the undecidability of the following subtypings:  $\leq_s$  that includes also send and receive actions and corresponds with (a fragment of) the subtyping defined by Mostrous and Yoshida [1],  $\leq_o$  that disallows orphan messages and coincides with the subtyping defined by Chen et al. [2], and  $\leq_m$  that deals with multiparty session types and corresponds with the subtyping introduced by Mostrous et al. [3].

As an additional result, we show that further restrictions on the branching/selection structure of types make our core subtyping relation  $\ll$  decidable. In fact, by imposing any of two possible restrictions on  $\ll$  — namely, in both subtype and supertype all input branchings (or all output selections) have one choice only — the obtained relation turns out to be decidable. We thus define the subtyping relations  $\ll_{\text{sin}}$  (both types are single-choice on inputs) and  $\ll_{\text{sout}}$  (both types are single-choice on outputs) by considering the two above restrictions on the asynchronous single-choice relation  $\ll$  and prove that both  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$  are decidable. As a matter of fact, we prove decidability for larger relations w.r.t.  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$  where we do not impose the constraint about no consecutive infinite outputs.

Figure 1 depicts the relations discussed in this paper as a lattice representing a  $\preceq_1 \longrightarrow \preceq_2$  order.  $\preceq_1 \longrightarrow \preceq_2$  means that it is possible to algorithmically reduce the problem of deciding the relation  $\preceq_1$  into the problem of deciding  $\preceq_2$ . As discussed above,  $\leq_s$ ,  $\leq_o$  and  $\leq_m$  are taken from the literature, while  $\leq$ ,  $\ll$ ,  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$  are defined in this paper to characterize as tightly as possible

the boundary between decidability and undecidability for asynchronous session subtyping relations. Obviously, when a relation is undecidable all relations above it (it reduces to) are also undecidable, while when a relation turns out to be decidable all relations below it (that reduce to it) are decidable as well.

*Structure of the paper.* In §2 we introduce a core language of session types with only branching/selection and recursion, and define for it the asynchronous subtyping relation  $\leq$ . In §3 we restrict subtyping to  $\ll$  and we show that such relation is undecidable. In §4, we discuss how our undecidability result allows us to prove the undecidability of other asynchronous subtypings, namely  $\leq$ ,  $\leq_s$ ,  $\leq_o$ ,  $\leq_m$ , and subtyping for communicating finite state machines (CFSMs). In §5 we discuss the two decidable relations  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$ , obtained as further restrictions of  $\ll$ . Finally, in §6 we comment the related literature and draw some concluding remarks.

## 2. Asynchronous Subtyping

In this section, we give a definition of a core session type language and define the asynchronous subtyping relation  $\leq$  following the approach by Mostrous and Yoshida [1].

### 2.1. Session Types

We start by presenting a very simple session type language (with only branching/selection and recursion) which is sufficient to prove our undecidability result.

**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:*

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

In our session type language we simply consider session termination **end**, recursive definitions  $\mu \mathbf{t}.T$ , with  $\mathbf{t}$  being the recursion variable, output selection  $\oplus\{l_i : T_i\}_{i \in I}$  and input branching  $\&\{l_i : T_i\}_{i \in I}$ . Each possible choice is labeled by a label  $l_i$ , taken from the set of labels  $L$ , followed by a session continuation  $T_i$ . Labels in a branching/selection are pairwise distinct.

### 2.2. Subtyping

We consider a notion of asynchronous subtyping corresponding to the subtyping relation by Mostrous and Yoshida [1] applied to our language. In particular we formalize the property of *output anticipation* (which, as we will see, characterizes asynchronous subtyping) by using the notion of *input context* as in Chen et al. [2] and Mostrous and Yoshida [1]. In order to define subtyping, we first need to introduce  $n$ -unfolding and input contexts.

The  $n$ -unfolding function unfolds nested recursive definitions to depth  $n$ .

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

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

**Definition 2.3** (**Input Context**). An input context  $\mathcal{A}$  is a session type with multiple holes defined by the following syntax:

$$\mathcal{A} ::= []^n \quad | \quad \&\{l_i : \mathcal{A}_i\}_{i \in I}$$

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  $T_1, \dots, T_k$ , 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 contexts we will assume them to be well-formed.

For example, consider the input context

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

we have:

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

We are finally ready to define our notion of subtyping.

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

1. if  $T = \mathbf{end}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = \mathbf{end}$ ;
2. if  $T = \oplus\{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0, \mathcal{A}$  such that
  - $\text{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in \{1, \dots, m\}}$  for some  $J_k$ ,
  - $\forall k \in \{1, \dots, m\}. I \subseteq J_k$  and
  - $\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k \in \{1, \dots, m\}}) \in \mathcal{R}$ ;
3. if  $T = \&\{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = \&\{l_j : S_j\}_{j \in J}$ ,  $J \subseteq I$  and  $\forall j \in J. (T_j, S_j) \in \mathcal{R}$ ;
4. if  $T = \mu\mathbf{t}.T'$  then  $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$ .

We say that  $T$  is a subtype of  $S$ , written  $T \leq S$ , if there is a subtyping relation  $\mathcal{R}$  such that  $(T, S) \in \mathcal{R}$ .

An important characteristic of asynchronous subtyping (formalized by rule 2. above) is the following one. In a subtype output selections can be anticipated so to bring them before the input branchings that in the supertype occur in front of them. For example the type

$$S = \&\{l_{\text{but1}} : \oplus\{l_{\text{coffee}} : T_1\}, l_{\text{but2}} : \oplus\{l_{\text{coffee}} : T_2\}\}$$

has the following subtype

$$T = \oplus\{l_{\text{coffee}} : \&\{l_{\text{but1}} : T_1, l_{\text{but2}} : T_2\}\}$$

where the output selection with label  $l_{\text{coffee}}$  is anticipated w.r.t. the input branching with labels  $l_{\text{but1}}$  and  $l_{\text{but2}}$ . That is, since in the supertype the input branching with labels  $l_{\text{but1}}$  and  $l_{\text{but2}}$  occurs in front of the output selection with label  $l_{\text{coffee}}$  (which is present in *all* its input branches), such an output selection can be anticipated so to bring it before the  $l_{\text{but1}}/l_{\text{but2}}$  input branching.

It is, thus, immediate to verify that, according to Definition 2.4, we have  $T \leq S$ . In particular, in rule 2. the well-formed input context considered to express the output  $l$  anticipation is  $\mathcal{A} = \&\{l_{\text{but1}} : \Box^1, l_{\text{but2}} : \Box^2\}$ . By considering this context, the supertype  $S$  can be written as

$$\mathcal{A}[\oplus\{l_{\text{coffee}} : T_1\}]^{i \in \{1,2\}} = \&\{l_{\text{but1}} : \oplus\{l_{\text{coffee}} : T_1\}, l_{\text{but2}} : \oplus\{l_{\text{coffee}} : T_2\}\}$$

Notice that in general an output selection can be anticipated even if it occurs in a larger input context, such as, for example

$$\mathcal{A} = \&\{l_{\text{but1}} : \&\{l_{\text{but3}} : \Box^1\}, l_{\text{but2}} : \Box^2\}$$

Conceptually output anticipation reflects the fact that we are considering asynchronous communication protocols in which messages are stored in queues. In this setting, it is safe to replace a peer that follows a given protocol with another one following a modified protocol where outputs are anticipated: in fact, the difference is simply that such outputs will be stored earlier in the communication queue.

### 2.3. Examples

Consider the types

$$\begin{aligned} T &= \mu\mathbf{t}.\&\{l : \oplus\{l : \mathbf{t}\}\} \\ S &= \mu\mathbf{t}.\&\{l : \&\{l : \oplus\{l : \mathbf{t}\}\}\} \end{aligned}$$

We have  $T \leq S$  as the following infinite set of type pairs is a subtyping relation:

$$\begin{aligned} \{ & (T, S), (\&\{l : \oplus\{l : T\}\}, S), (\oplus\{l : T\}, \&\{l : \oplus\{l : S\}\}), \\ & (T, \&\{l : S\}), (\&\{l : \oplus\{l : T\}\}, \&\{l : S\}), (\oplus\{l : T\}, S), \\ & (T, \&\{l : \&\{l : S\}\}), \dots \\ & (T, \&\{l : \&\{l : \&\{l : S\}\}\}), \dots \\ & \dots \} \end{aligned}$$



Notice that the types on the r.h.s. ( $S$  and subsequent ones) can always mimic the initial actions of the corresponding type on the l.h.s. ( $T$  and subsequent ones). Pairs are presented above in such a way that: the second one is reached from the first one by rule 4. of Definition 2.4 (recursion), the third one is reached from the second one by rule 3. of Definition 2.4 (input), the first one in the subsequent line is reached from the third one by rule 2. of Definition 2.4 (output), and similarly (using the same rules) in the subsequent lines. Notice that, every time an output  $\oplus\{l : \_ \}$  must be mimicked, the r.h.s. must be unfolded, and the corresponding output is anticipated, since it is preceded by inputs only (i.e. the output fills an input context). The effect of the anticipation of the output is that a new input  $\&\{l : \_ \}$  is accumulated at the beginning of the r.h.s. It is worth to observe that every accumulated input  $\&\{l : \_ \}$  is eventually consumed in the simulation game, but the accumulated inputs grows unboundedly.

As another example consider

$$\begin{aligned}
T &= \mu t. \&\{ \\
&\quad l_{\mathbf{but1}} : \oplus\{l_{\mathbf{coffee}} : \mathbf{t}\}, \\
&\quad l_{\mathbf{but2}} : \oplus\{l_{\mathbf{tea}} : \mathbf{t}\} \\
&\quad \} \\
S &= \mu t. \&\{ \\
&\quad l_{\mathbf{but2}} : \oplus\{ \\
&\quad \quad l_{\mathbf{coffee}} : \mathbf{t}, \\
&\quad \quad l_{\mathbf{tea}} : \&\{l_{\mathbf{but1}} : \mathbf{t}, l_{\mathbf{but2}} : \mathbf{t}\} \\
&\quad \} \\
&\quad \}
\end{aligned}$$

We have  $T \leq S$  for the following reasons. Type  $T$  repeatedly alternates input and output, the input corresponding to an input branching with labels  $l_{\mathbf{but1}}$  and  $l_{\mathbf{but2}}$  (where **but** stands for “button”), and the output with only one label: either  $l_{\mathbf{coffee}}$  or  $l_{\mathbf{tea}}$ . Also  $S$  infinitely repeats input and output, but, depending on which of its inputs it performs, the corresponding input branching can have fewer choices than  $T$  (in the case of the input branching with just one label, i.e.  $l_{\mathbf{but2}}$ ). The output, instead, always corresponds to an output selection with labels  $l_{\mathbf{coffee}}$  and  $l_{\mathbf{tea}}$ . Such a difference between  $T$  and  $S$  is not problematic due to contravariance on input branchings and covariance on output selections. Type  $S$  also differs because after the input with label  $l_{\mathbf{but2}}$  and the output with label  $l_{\mathbf{tea}}$  the type  $\&\{l_{\mathbf{but1}} : S, l_{\mathbf{but2}} : S\}$  is reached, and this term (no matter which input is chosen) may perform two consecutive inputs, the second one available upon unfolding of  $S$ . Type  $T$  does not have two consecutive inputs because it always alternates input and output. Nevertheless, we still have  $T \leq \&\{l_{\mathbf{but1}} : S, l_{\mathbf{but2}} : S\}$  because, as discussed in the previous example, according to our notion of asynchronous subtyping (rule 2. of Definition 2.4) the output in the r.h.s. can be always anticipated to match the output actions of the l.h.s., if they are preceded by inputs only.

### 3. Core Undecidability Result

In this section we prove our core undecidability result for a restricted subtyping relation. This relation is called asynchronous single-choice subtyping and corresponds to the restriction of  $\leq$  to pairs of session types  $(T, S)$  such that  $T$  has output selections with one choice only,  $S$  has input branchings with one choice only, and both  $T$  and  $S$  cannot have infinite sequences of outputs.

#### 3.1. Asynchronous single-choice subtyping

In order to formally define such relation we need some preliminary definitions.

**Definition 3.1 (Session types with single outputs).** *Given a set of labels  $L$ , ranged over by  $l$ , the syntax of binary session types with single outputs is given by the following grammar:*

$$T^{\text{out}} ::= \oplus\{l : T^{\text{out}}\} \mid \&\{l_i : T^{\text{out}}_i\}_{i \in I} \mid \mu \mathbf{t}.T^{\text{out}} \mid \mathbf{t} \mid \mathbf{end}$$

Session types with single outputs are all those session types where inputs can have multiple choices while outputs must be singletons.

**Definition 3.2 (Session types with single inputs).** *Given a set of labels  $L$ , ranged over by  $l$ , the syntax of binary session types with single inputs is given by the following grammar:*

$$T^{\text{in}} ::= \oplus\{l_i : T^{\text{in}}_i\}_{i \in I} \mid \&\{l : T^{\text{in}}\} \mid \mu \mathbf{t}.T^{\text{in}} \mid \mathbf{t} \mid \mathbf{end}$$

Session types with single inputs, instead, are all those session types where inputs are singletons and outputs may have multiple choices.

**Definition 3.3 (Session types with input guarded recursion).** *The set of session types  $T^{\text{noinf}}$  is composed of the session types  $T$  that satisfy the following condition: for every subterm of  $T$  of the form  $\mu \mathbf{t}.R$ , for some  $\mathbf{t}$  and  $R$ , every occurrence of  $\mathbf{t}$  in  $R$  is inside a subterm of  $R$  of the form  $\&\{l_i : S_i\}_{i \in I}$ , for some set of labels  $l_i$  and session types  $S_i$ .*

Session types with input guarded recursion have an important property: there are no consecutive infinite outputs.

The asynchronous single-choice relation is defined as the subset of  $\leq$  where types on the left-hand side of the relation are with single outputs, types on the right-hand side are with single inputs, and both types have no infinite sequences of outputs.

**Definition 3.4 (Asynchronous Single-Choice Relation).** *The asynchronous single-choice relation  $\ll$  is defined as:*

$$\ll = \leq \cap (T^{\text{out}} \times T^{\text{in}}) \cap (T^{\text{noinf}} \times T^{\text{noinf}})$$

*Remark.* Note that the asynchronous single-choice relation is not reflexive. In fact, any type that has multiple (non single) choices is not related to itself, e.g.,  $\&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\} \not\leq \&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\}$  simply because the term on right-hand side has more than one input branch.

### 3.2. Queue Machines

The proof of undecidability of the asynchronous single-choice relation is by reduction from the acceptance problem for queue machines. Queue machines have been already informally presented in the Introduction, we now report their formal definition.

**Definition 3.5 (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;
- $\Gamma$  is a finite set denoting the queue alphabet (ranged over by  $A, B, C, X$ );
- $\$ \in \Gamma - \Sigma$  is the initial queue symbol;
- $s \in Q$  is the start state;
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma^*$  is the transition function.

In the Introduction we have informally described a queue machine that accepts the language  $a^n b^n$ , we now present its formal definition. Let  $M = (\{q_1, q_2, q_3, q_s\}, \{a, b\}, \{a, b, \$\}, \$, q_1, \delta)$  with  $\delta$  defined as follows:

- $\delta(q_1, a) = (q_2, \epsilon), \delta(q_1, \$) = (q_1, \epsilon), \delta(q_1, b) = (q_s, b);$
- $\delta(q_2, a) = (q_2, a), \delta(q_2, \$) = (q_s, \$), \delta(q_2, b) = (q_3, \epsilon);$
- $\delta(q_3, a) = (q_s, a), \delta(q_3, \$) = (q_1, \$), \delta(q_3, b) = (q_3, b);$
- $\delta(q_s, X) = (q_s, X)$  for  $X \in \{a, b, \$\}.$

Differently from the informal definition in the Introduction, here (since, according to Definition 3.5, the transition function  $\delta$  is expected to be total) we have to consider an additional sink state which is entered whenever an unexpected symbol is consumed from the queue. Once this state is entered, it will be no longer possible to leave it, and every consumed symbol will be simply re-added to the queue.

We now formally define queue machine computations.

**Definition 3.6 (Queue machine computation).** A configuration of a queue machine is an ordered pair  $(q, \gamma)$  where  $q \in Q$  is its current state and  $\gamma \in \Gamma^*$  is the queue ( $\Gamma^*$  is the Kleene closure of  $\Gamma$ ). The starting configuration on an input string  $x$  is  $(s, x\$)$ . The transition relation  $\rightarrow_M$  over configurations  $Q \times \Gamma^*$ , leading from a configuration to the next one, is defined as follows. For any  $p, q \in Q$ ,

$A \in \Gamma$  and  $\alpha, \gamma \in \Gamma^*$  we have  $(p, A\alpha) \rightarrow_M (q, \alpha\gamma)$  whenever  $\delta(p, A) = (q, \gamma)$ . A machine  $M$  accepts an input  $x$  if it eventually terminates on input  $x$ , i.e. it reaches a blocking configuration with the empty queue (notice that, as the transition relation is total, the unique way to terminate is by emptying the queue). Formally,  $x$  is accepted by  $M$  if  $(s, x\$) \rightarrow_M^* (q, \epsilon)$  where  $\epsilon$  is the empty string and  $\rightarrow_M^*$  is the reflexive and transitive closure of  $\rightarrow_M$ .

Going back to the queue machine  $M$  defined above, if we consider the input  $aabb$  we have the following computation:

$$\begin{aligned} (q_1, aabb\$) &\rightarrow_M (q_2, abb\$) \rightarrow_M (q_2, bb\$a) \rightarrow_M (q_3, b\$a) \rightarrow_M (q_3, \$ab) \rightarrow_M \\ (q_1, ab\$) &\rightarrow_M (q_2, b\$) \rightarrow_M (q_3, \$) \rightarrow_M (q_1, \$) \rightarrow_M (q_1, \epsilon) \end{aligned}$$

Hence, we can conclude that the string  $aabb$  is accepted by  $M$  (as any other string of type  $a^n b^n$ ).

Turing completeness of queue machines is discussed by Kozen [8] (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. Formally, given any Turing machine  $T$  we have that a string  $x$  is accepted by  $T$  if and only if  $x$  is accepted by the queuing machine  $M$  obtained as the encoding of  $T$ . The undecidability of acceptance of an input string  $x$  by a machine  $M$  follows directly from such encoding.

### 3.3. Modelling Queue Machines with Session Types

Our goal is to construct a pair of types, say  $T$  and  $S$ , from a given queue machine  $M$  and a given input  $x$ , such that:  $T \ll S$  if and only if  $x$  is not accepted by  $M$ . Intuitively, type  $T$  encodes the finite control of  $M$ , i.e., its transition function  $\delta$ , starting from its initial state  $s$ . And type  $S$  encodes the machine queue that initially contains  $x\$$ , where  $x$  is the input string  $x = X_1 \cdots X_n$  of length  $n \geq 0$ . The set of labels  $L$  for such types  $T$  and  $S$  is  $M$ 's queue alphabet  $\Gamma$ .

Formally, the queue of a machine is encoded into a session type as follows:

**Definition 3.7 (Queue Encoding).** Let  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$  be a queue machine and let  $C_1 \cdots C_m \in \Gamma^*$ , with  $m \geq 0$ . Then, the queue encoding function  $[C_1 \cdots C_m]$  is defined as:

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

Given a configuration  $(q, \gamma)$  of  $M$ , the encoding of the queue  $\gamma = C_1 \cdots C_m$  is thus defined as  $[C_1 \cdots C_m]$ .

Note that whenever  $m = 0$ , we have  $[\epsilon] = \mu\mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$ . Observe that we are using a slight abuse of notation: in both output selections and input branchings, labels  $l_A$ , with  $A \in \Gamma$ , are simply denoted by  $A$ .

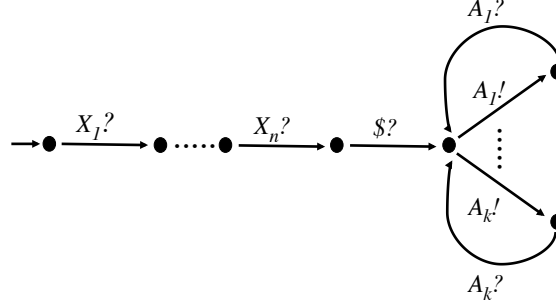


Figure 2: Labelled Transition System of a session type encoding the initial queue  $X_1 \dots X_n \$$

Figure 2 contains a graphical representation of the queue encoding with its initial content  $X_1 \dots X_n \$$ . In order to better clarify our development, we graphically represent session types as labeled transition systems (in the form of communicating automata [9]), where an output selection  $\oplus\{l_i : T_i\}_{i \in I}$  is represented as a choice among alternative output transitions labeled with “ $l_i!$ ”, and an input branching  $\&\{l_i : T_i\}_{i \in I}$  is represented as a choice among alternative input transitions labeled with “ $l_i?$ ”. Intuitively, we encode a queue containing symbols  $C_1 \dots C_m$  with a session type that starts with  $m$  inputs with labels  $C_1, \dots, C_m$ , respectively. Thus, in Figure 2, we have  $C_1 \dots C_m = X_1 \dots X_n \$$ . After such sequence of inputs, representing the current queue content, there is a recursive type representing the capability to enqueue new symbols. Such a type repeatedly performs an output selection with one choice for each symbol  $A_i$  in the queue alphabet  $\Gamma$  (with  $k$  being the cardinality of  $\Gamma$ ), followed by an input labeled with the same symbol  $A_i$ .

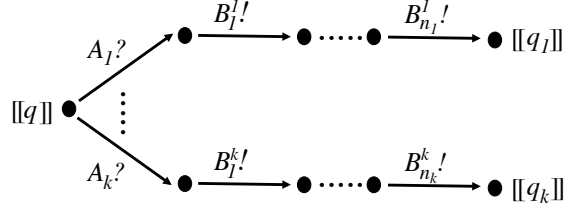
We now give the definition of the type modelling the finite control of a queue machine, i.e., the encoding of the transition function  $\delta$ .

**Definition 3.8 (Finite Control Encoding).** *Let  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$  be a queue machine and let  $q \in Q$  and  $\mathcal{S} \subseteq Q$ . Then,*

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

*The encoding of the transition function of  $M$  is then defined as  $[s]^0$ .*

The finite control encoding is a recursively defined term with one recursion variable  $\mathbf{q}$  for each state  $q \in Q$  of the machine. Above,  $[q]^{\mathcal{S}}$  is a function that, given a state  $q$  and a set of states  $\mathcal{S}$ , returns a type representing the possible behaviour of the queue machine starting from state  $q$ . Such behaviour consists of first reading from the queue (input branching on  $A \in \Gamma$ ) and then writing on the queue a sequence of symbols  $B_1^A, \dots, B_{n_A}^A$ . The parameter  $\mathcal{S}$  is necessary



(for  $\Gamma = \{A_i | i \leq k\}$  and  $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$  for every  $i$ .)

Figure 3: Labelled Transition System of a session type encoding a finite control.

for managing the recursive definition of this type. In fact, as the definition of the encoding function is itself recursive, this parameter keeps track of the states that have been already encoded (see example below). In Figure 3, we report a graphical representation of the Labelled Transition System corresponding to the session type that encodes the queue machine finite control, i.e. the transition function  $\delta$ . Each state  $q \in Q$  is mapped onto a state  $\llbracket q \rrbracket$  of a session type, which performs an input branching with a choice for each symbol in the queue alphabet  $\Gamma$  (with  $k$  being the cardinality of  $\Gamma$ ). Each of these choices represents a possible character that can be read from the queue. After this initial input branching, each choice continues with a sequence of outputs labeled with the symbols that are to be inserted in the queue (after the symbol labeling that choice has been consumed). This is done according to function  $\delta$ , assuming that  $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$ , with  $n_i \geq 0$ , for all  $i$  in  $\{1, \dots, k\}$ . After the insertion phase, state  $\llbracket q_i \rrbracket$  of the session type corresponding to state  $q_i$  of the queue machine is reached.

Notice that, queue insertion actually happens in the encoding because, when the encoding of the finite control performs an output of a  $B$  symbol, the encoding of the queue must mimic such an output, possibly by anticipating it. This has the effect of adding an input on  $B$  at the end of the sequence of initial inputs of the queue machine encoding.

Observe that our encodings generate terms that belong to the restricted syntax of session types introduced in the previous section, namely the queue encoding of Definition 3.7 produces types in  $T^{\text{in}}$ , while the finite control encoding of Definition 3.8 produces types in  $T^{\text{out}}$ .

*Example.* As an example, consider a queue machine with two states  $s$  and  $q$ , any non-empty input and queue alphabets  $\Sigma$  and  $\Gamma$ , and a transition relation defined as follows:  $\delta(s, A) = (q, A)$  and  $\delta(q, A) = (s, \epsilon)$ , for every queue symbol  $A \in \Gamma$ . We have that

$$\begin{aligned}
 [s]^\emptyset &= \mu s. \& \{A : \oplus \{A : [q]^{\{s\}}\}\}_{A \in \Gamma} \\
 &= \mu s. \& \{A : \oplus \{A : \mu q. \& \{A : [s]^{\{s, q\}}\}_{A \in \Gamma}\}\}_{A \in \Gamma} \\
 &= \mu s. \& \{A : \oplus \{A : \mu q. \& \{A : s\}_{A \in \Gamma}\}\}_{A \in \Gamma}
 \end{aligned}$$

### 3.4. Properties of the Encodings

We begin by proving that subtyping is preserved by reductions of queue machines (modulo our encoding), and then we exploit this property (Lemma 3.1) to prove our core undecidability result (Theorem 3.1).

**Lemma 3.1.** *Consider a queue machine  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ . If  $(q, \gamma) \rightarrow_M (q', \gamma')$  and  $[q]^\emptyset \ll [\gamma]$  then also  $[q']^\emptyset \ll [\gamma']$ .*

*Proof.* Assume that  $(q, \gamma) \rightarrow_M (q', \gamma')$  with  $\gamma = C_1 \cdots C_m$  and  $\delta(q, C_1) = (q', B_1^{C_1} \cdots B_{n_{C_1}}^{C_1})$ . Then we have that  $\gamma' = C_2 \cdots C_m B_1^{C_1} \cdots B_{n_{C_1}}^{C_1}$ .

Assume now  $[q]^\emptyset \ll [\gamma]$ ; this means that there exists an asynchronous subtyping relation  $\mathcal{R}$  s.t.  $([q]^\emptyset, [C_1 \cdots C_m]) \in \mathcal{R}$ . By item 4 of Definition 2.4, we also have

$$(\&\{A : \oplus\{B_1^A : \cdots \oplus \{B_{n_A}^A : [q]^\emptyset\}\}\}_{A \in \Gamma}, [C_1 \cdots C_m]) \in \mathcal{R}$$

where the l.h.s. has been unfolded once. By item 3 of Definition 2.4, the presence of the above pair in  $\mathcal{R}$  guarantees also that

$$(\oplus\{B_1^{C_1} : \oplus\{B_2^{C_1} : \cdots \oplus \{B_{n_{C_1}}^{C_1} : [q]^\emptyset\}\}\}, \text{unfold}^{n_0}(\&\{C_2 : \cdots \&\{C_m : Z\}\})) \in \mathcal{R}$$

for some  $n_0$ , and with  $Z = \mu\mathbf{t} . \oplus\{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$ . By item 2 of Definition 2.4, the presence of this last pair in  $\mathcal{R}$  guarantees also that

$$(\oplus\{B_2^{C_1} : \cdots \oplus \{B_{n_{C_1}}^{C_1} : [q]^\emptyset\}\}, \text{unfold}^{n_1}(\&\{C_2 : \cdots \&\{C_m : \&\{B_1^{C_1} : Z\}\}\})) \in \mathcal{R}$$

for some  $n_1$ . By repeating the same reasoning on  $B_2^{C_1}, \dots, B_{n_{C_1}}^{C_1}$  we conclude that also

$$([q]^\emptyset, \text{unfold}^{n_{C_1}}(\&\{C_2 : \cdots \&\{C_m : \&\{B_1^{C_1} : \cdots \&\{B_{n_{C_1}}^{C_1} : Z\}\}\}\})) \in \mathcal{R}$$

for some  $n_{C_1}$ .

We now observe that if an asynchronous subtyping relation  $\mathcal{R}$  contains the pair  $(T, \text{unfold}^n(S))$ , for some  $n$ , then we have that also the following is an asynchronous subtyping relation:  $\mathcal{R} \cup \{(T', S) \mid T' \in \text{topUnfold}(T)\}$ , where  $\text{topUnfold}(T)$  is the minimal set of types that contains  $T$  and such that  $\mu\mathbf{t}.R \in \text{topUnfold}(T)$  implies  $R\{\mu\mathbf{t}.R/\mathbf{t}\} \in \text{topUnfold}(T)$ .

In the light of this last observation we can now conclude that having

$$([q]^\emptyset, \text{unfold}^{n_{C_1}}(\&\{C_2 : \cdots \&\{C_m : \&\{B_1^{C_1} : \cdots \&\{B_{n_{C_1}}^{C_1} : Z\}\}\}\})) \in \mathcal{R}$$

in the asynchronous subtyping relation  $\mathcal{R}$  implies that

$$[q]^\emptyset \ll \&\{C_2 : \cdots \&\{C_m : \&\{B_1^{C_1} : \cdots \&\{B_{n_{C_1}}^{C_1} : Z\}\}\}\}$$

Notice that the r.h.s. corresponds to  $[\gamma']$ . Hence we have proved the thesis  $[q]^\emptyset \ll [\gamma']$ .  $\square$

We are now ready to prove our main theorem.

**Theorem 3.1.** *Given a queue machine  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ , an input string  $x$ , and the two types  $T = [s]^\emptyset$  and  $S = [x\$]$ , we have that  $M$  accepts  $x$  if and only if  $T \not\ll S$ .*

*Proof.* We prove the two directions separately.

(*Only if part*). We first observe that  $[q]^\emptyset \not\ll [\epsilon]$  for every possible state  $q$ . In fact,  $[q]^\emptyset$  is a recursive definition that upon unfolding begins with an input branching that implies (according to items 3. and 4. of Definition 2.4) that also  $[\epsilon]$  (once unfolded, if needed) should start with an input branching. But this is false, in that, by definition of the encoding we have  $[\epsilon] = \mu\mathbf{t} \cdot \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$ . We can conclude that  $[s]^\emptyset \not\ll [x\$]$  because otherwise, by repeated application of Lemma 3.1, we would have that the termination of the queue machine, i.e.  $(s, x\$) \rightarrow_M^* (q, \epsilon)$ , implies the existence of a state  $q$  such that  $[q]^\emptyset \ll [\epsilon]$ . But we have just proved that  $[q]^\emptyset \ll [\epsilon]$  does not hold, for every state  $q$ .

(*If part*). Our aim is to show that if  $T \not\ll S$  then  $M$  accepts  $x$ , which is equivalent to showing that  $T \ll S$ , assuming that  $M$  does not accept  $x$ . 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  $(s, x\$) \rightarrow_M (q_1, \gamma_1) \rightarrow_M \dots \rightarrow_M (q_i, \gamma_i) \rightarrow_M \dots$ . Let  $\mathcal{C}$  be the set of reachable configurations, i.e.  $\mathcal{C} = \{(q_i, \gamma_i) \mid i \geq 0\}$  where we assume  $(q_0, \gamma_0) = (s, x\$)$ . We now define a relation  $\mathcal{R}$  on types:

$$\begin{aligned} \mathcal{R} = & \{ \quad ([q]^\emptyset, [C_1 \dots C_m]), \\ & (\&\{A : \oplus \{B_1^A : \dots \oplus \{B_{n_A}^A : [q]^\emptyset\}\}_{A \in \Gamma}, [C_1 \dots C_m]), \\ & (\oplus \{B_1^{C_1} : \oplus \{B_2^{C_1} : \dots \oplus \{B_{n_{C_1}}^{C_1} : [q]^\emptyset\}\}, \&\{C_2 : \dots \&\{C_m : Z\}\}), \\ & (\oplus \{B_2^{C_1} : \dots \oplus \{B_{n_{C_1}}^{C_1} : [q]^\emptyset\}, \&\{C_2 : \dots \&\{C_m : \&\{B_1^{C_1} : Z\}\}\}), \\ & \dots \\ & ([q]^\emptyset, \&\{C_2 : \dots \&\{C_m : \&\{B_1^{C_1} : \dots \&\{B_{n_{C_1}}^{C_1} : Z\}\}\}) \\ & \mid (q, C_1 \dots C_m) \in \mathcal{C}, \delta(q, C_1) = (q', B_1^{C_1} \dots B_{n_{C_1}}^{C_1}), \\ & \quad Z = \mu\mathbf{t} \cdot \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma} \quad \} \end{aligned}$$

Notice that the type pairs listed in the above definition correspond to the pairs discussed in the proof of Lemma 3.1. We have that the above  $\mathcal{R}$  is a subtyping relation because, using a reasoning similar to the one reported in the proof of Lemma 3.1, it is immediate to see that each of the pairs satisfies the conditions in Definition 2.4 thanks to the presence of the subsequent pair. The unique pair without a subsequent pair is the last one, but this last pair corresponds to the first one of the pairs corresponding to the configuration  $(q', C_2 \dots C_m B_1^{C_1} \dots B_{n_{C_1}}^{C_1}) \in \mathcal{C}$  reached in the queue machine computation after  $(q, C_1 \dots C_m)$ , i.e.  $(q, C_1 \dots C_m) \rightarrow_M (q', C_2 \dots C_m B_1^{C_1} \dots B_{n_{C_1}}^{C_1})$ . We can conclude observing that  $T \ll S$  because  $T = [s]^\emptyset$ ,  $S = [x\$]$  and  $(s, x\$) \in \mathcal{C}$  implies that  $([s]^\emptyset, [x\$])$  belongs to the above subtyping relation  $\mathcal{R}$ .  $\square$

As a corollary, we have that the asynchronous single-choice relation is undecidable.



**Corollary 3.1.** *Asynchronous single-choice subtyping for binary session types  $\ll$  is undecidable.*

*Proof.* From Theorem 3.1 we know that given any queue machine  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ , an input string  $x$ , and the two types  $T = [s]^\emptyset$  and  $S = [x\$]$ , we have that  $M$  does not accept  $x$  if and only if  $T \ll S$ . From the undecidability of acceptance for queue machines we can conclude the undecidability of  $\ll$ .  $\square$

#### 4. Impact of the Undecidability Result

Starting from our core undecidability result we prove the undecidability of other subtyping relations starting from  $\leq$ . In the following subsections, for the sake of simplicity, we denote different session type languages with the same letters  $T, S, \dots$  and the actual language will be made clear by the context.

##### 4.1. Undecidability of Asynchronous Subtyping

The undecidability of the asynchronous single-choice relation can be exploited to show that asynchronous subtyping is undecidable. Intuitively, this follows by the fact that the encoding of a queue machine into  $\ll$  is also a valid encoding into  $\leq$ .

**Corollary 4.1.** *Asynchronous subtyping for binary session types  $\leq$  is undecidable.*

*Proof.* A direct consequence of the undecidability of  $\ll$ , i.e.  $\leq$  restricted to the pairs of types belonging to  $(T^{\text{out}} \times T^{\text{in}}) \cap (T^{\text{noinf}} \times T^{\text{noinf}})$ .  $\square$

##### 4.2. Standard Binary Session Types with Asynchronous Subtyping

The syntax proposed in Definition 2.1 allows for the definition of session types with only output selections and input branchings, and recursion. Standard binary session types [4] also feature classic send/receive types, containing the type of the communicated message, dubbed *carried type*. Carried types can be primitive types such as `bool`, `nat`, `...` or a session type  $T$  (modelling delegation). We define standard binary session types as follows:

**Definition 4.1 (Standard Session Types).** *Standard binary session types are defined as*

$$\begin{aligned} T &::= \dots \text{ as in Definition 2.1 } \dots \mid !\langle U \rangle.T \mid ?(U).T \\ U &::= T \mid \text{bool} \mid \text{nat} \mid \dots \end{aligned}$$

In order to extend subtyping to standard binary session types, we need to adapt the notion of unfolding. The  $n$ -unfolding function is extended as follows:

$$\begin{aligned} &\dots \text{ as in Definition 2.2 } \dots \\ \text{unfold}^1(!\langle U \rangle.T) &= !\langle U \rangle.\text{unfold}^1(T) \\ \text{unfold}^1(? (U).T) &= ?(U).\text{unfold}^1(T) \end{aligned}$$

Moreover, the input context definition becomes:

$$\mathcal{A} ::= \dots \text{as in Definition 2.3} \dots \quad | \quad ?(U).\mathcal{A}$$

Finally, the asynchronous subtyping relation for standard binary session types is given by the following:

**Definition 4.2. (Asynchronous Subtyping for Standard Session Types,  $\leq_s$ )** *Asynchronous subtyping for standard session types is defined as in Definition 2.4 where we consider a relation  $\mathcal{R}$  on both session and primitive types that satisfies, besides the items in that definition, also the following ones (with an abuse of notation, we use  $T$  and  $S$  to range over both session and primitive types):*

5. if  $T \in \{\text{bool} \mid \text{nat} \mid \dots\}$  then  $S = T$ ;
6. if  $T = !\langle U \rangle.T'$  then  $\exists n \geq 0, \mathcal{A}$  such that
  - $\text{unfold}^n(S) = \mathcal{A}[\langle V_k \rangle.S_k]^{k \in \{1, \dots, m\}},$
  - $\forall k \in \{1, \dots, m\}. (V_k, U) \in \mathcal{R}$  and
  - $(T, \mathcal{A}[S_k]^{k \in \{1, \dots, m\}}) \in \mathcal{R};$
7. if  $T = ?(U).T'$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = ?(V).S'$  and  $(U, V) \in \mathcal{R}$  and  $(T', S') \in \mathcal{R}.$

The undecidability proof applies to this extended setting:

**Corollary 4.2.** *Asynchronous subtyping  $\leq_s$  for standard binary session types is undecidable.*

*Proof.* The core session type language in Definition 2.1 on which  $\leq$  is defined is a fragment of the standard session type language defined in the present section. The thesis follows from the fact that on this fragment the two subtyping relations  $\leq$  and  $\leq_s$  coincide.  $\square$

*Remark.* It is worth to observe that the asynchronous subtyping relation  $\leq_s$  defined above corresponds to the one by Mostrous and Yoshida [1], with the only difference that the various rules for the different kinds of carried types considered in that paper are replaced by the simpler rule 5., that considers primitive types only (carried session types are managed by the rules 6. and 7.). Hence our undecidability result applies also to the subtyping by Mostrous and Yoshida [1].

#### 4.3. Carried Types in Selection/Branching and no Orphan Messages

Chen et al. [2] propose a variant of standard binary session types where messages of some type can also be communicated together with a choice performed within an output selection/input branching. This is defined by the following syntax.

**Definition 4.3** (Session types with carried types on choices [2]).

$$\begin{aligned} T &::= \oplus\{l_i\langle U_i \rangle : T_i\}_{i \in I} \mid \&\{l_i(U_i) : T_i\}_{i \in I} \mid \mu t.T \mid \mathbf{t} \mid \mathbf{end} \\ U &::= T \mid \mathbf{bool} \mid \mathbf{nat} \mid \dots \end{aligned}$$

This syntax corresponds with the one we have considered in Definition 2.1, where  $\oplus\{l_i : T_i\}_{i \in I}$  and  $\&\{l_i : T_i\}_{i \in I}$  are replaced by  $\oplus\{l_i\langle U_i \rangle : T_i\}_{i \in I}$  and  $\&\{l_i(U_i) : T_i\}_{i \in I}$ , respectively. The definition of  $n$ -unfolding can be updated accordingly.

Chen et al. [2] also propose a different definition of subtyping that does not allow to have orphan messages, i.e., inputs on the right-hand side of the subtyping relation cannot be indefinitely delayed. We reformulate the subtyping by Chen et al. [2] (defined on infinite trees) as follows.

**Definition 4.4** (Asynchronous Orphan-Message-Free Subtyping,  $\leq_o$ ).

*Asynchronous subtyping for orphan-message-free session types is defined as in Definition 2.4 where we consider a relation  $\mathcal{R}$  on both session and primitive types that satisfies the items in that definition, with items 2. and 3. replaced by the following corresponding ones, plus the additional rule 5. (also in this case, we use  $T$  and  $S$  to range over both session and primitive types):*

2. if  $T = \oplus\{l_i\langle U_i \rangle : T_i\}_{i \in I}$  then  $\exists n \geq 0, \mathcal{A}$  such that
  - $\exists j \in I$  s.t.  $T_j$  does not contain input branchings implies  $\mathcal{A} = []^1$ ,
  - $\text{unfold}^n(S) = \mathcal{A}[\oplus\{l_j\langle V_{kj} \rangle : S_{kj}\}_{j \in J_k}]^{k \in \{1, \dots, m\}}$ ,
  - $\forall k \in \{1, \dots, m\}. I \subseteq J_k$  and
  - $\forall i \in I, k \in \{1, \dots, m\}. (V_{ki}, U_i) \in \mathcal{R}$ ;
  - $\forall i \in I. (T_i, \mathcal{A}[S_{ki}]^{k \in \{1, \dots, m\}}) \in \mathcal{R}$ ;
3. if  $T = \&\{l_i(U_i) : T_i\}_{i \in I}$  then  $\exists n \geq 0$  such that
  - $\text{unfold}^n(S) = \&\{l_j(V_j) : S_j\}_{j \in J}, J \subseteq I$  and  $\forall j \in J. (U_j, V_j), (T_j, S_j) \in \mathcal{R}$ ;
5. if  $T \in \{\mathbf{bool} \mid \mathbf{nat} \mid \dots\}$  then  $S = T$ .

The key point in the definition above is rule 2., first item, that guarantees that if the r.h.s. does not start with the output needed to be mimicked in the simulation game, and then such output must be anticipated, then all possible continuations in the l.h.s. must contain at list an input. This implies that the input in the r.h.s. that have been delayed due to the anticipation, will be eventually involved in the simulation game, i.e. they will be not delayed indefinitely. It is worth to notice that this is guaranteed already by the core relation  $\ll$  because it avoids processes from having consecutive infinite outputs.

Also in this case the undecidability proof applies to this extended setting:

**Corollary 4.3.** *Asynchronous subtyping  $\leq_o$  for binary session types with carried types in output selections/input branchings and asynchronous orphan-message-free subtyping is undecidable.*

*Proof.* First of all, we observe that our core session language has a one-to-one correspondence with a fragment of the language given in Definition 4.3, under the assumption that only one given primitive type can be carried (e.g. `bool`). Then, we observe (as already remarked above) that if we consider terms without consecutive infinite outputs the new additional first item of rule 2. in Definition 4.4 can be omitted without changing the defined relation. In fact, we show that it is implied by the other conditions. If during the simulation game an output is anticipated w.r.t. some inputs in the r.h.s. (i.e. we use  $A \neq []^1$  in the application of rule 2.) then the continuations of the simulation game could be either finite or infinite. For the finite continuations we have that the inputs in front of the r.h.s. must be eventually consumed otherwise the pair  $(\mathbf{end}, \mathbf{end})$  cannot be reached; hence at least one input should be present in the l.h.s. In the infinite continuations, the fact that the l.h.s. has no consecutive infinite outputs guarantees the presence in such term of at least one input.

Hence we can conclude that  $\ll$ , which is defined on terms belonging to  $T^{\text{noinf}}$ , is isomorphic to  $\leq_o$  restricted to terms without consecutive infinite output. The undecidability of  $\leq_o$  thus directly follows from the undecidability of  $\ll$ .  $\square$

*Remark.* It is immediate to conclude that also the subtyping relation by Chen et al. [2] is undecidable.

#### 4.4. Multiparty session types

We now investigate how our undecidability result can be applied to a version of multiparty session types given by Mostrous et al. [3]. Multiparty session types are an extension of binary session types that allow to describe protocols between several parties. Protocols, specified as *global types* [5], can then be projected into *local types*, formally defined as follows.

**Definition 4.5 (Local Types).**

$$\begin{aligned} T &::= k!\langle U \rangle.T \mid k?(U).T \mid k \oplus \{l_i : T_i\}_{i \in I} \mid k \& \{l_i : T_i\}_{i \in I} \mid \mu \mathbf{t}.T \mid \mathbf{t} \mid \mathbf{end} \\ U &::= T \mid \mathbf{bool} \mid \mathbf{nat} \mid \dots \end{aligned}$$

Local types are a generalisation of the standard binary session types seen in Definition 4.1, where communications can now be performed on different channels, e.g., a process involved in a session with type  $k!\langle U \rangle.k'?(U').T$  first outputs something of type  $U$  on channel  $k$ , and, then, inputs something of type  $U'$  from channel  $k'$ .

Before introducing our definition of subtyping for local types, we note that the definition of  $n$ -unfolding can be trivially adapted to terms in the definition above from our initial definition. Moreover, we need to redefine input contexts for local types: such contexts now contain also outputs, under the assumption that those outputs are on different channels. This reflects the fact that ordering is guaranteed to be preserved only by messages sent on the same channel. Technically, we use a parameterized notion of input context  $\mathcal{A}^k$  where  $k$  is assumed to be the channel of the output to be anticipated.

**Definition 4.6 (Multiparty input context).** A multiparty input context  $\mathcal{A}^k$  is a session type with multiple holes defined by the following syntax:

$$\mathcal{A}^k ::= []^n \mid k'?(U).\mathcal{A}^k \mid k' \& \{l_i : \mathcal{A}^k_i\}_{i \in I} \mid k''!\langle U \rangle.\mathcal{A}^k \mid k'' \oplus \{l_i : \mathcal{A}^k_i\}_{i \in I}$$

where we assume that  $k'$  is any possible channel while  $k'' \neq k$ .

We are now ready to define the subtyping relation for local types. The definition of subtyping we propose is inspired by the one initially proposed by Mostrous et al. [3]. Unlike the binary case, outputs on a channel can be anticipated over inputs and outputs on different channels. Moreover, unlike Mostrous et al. [3], we allow outputs to be anticipated over inputs on the same channel, and we do not allow inputs over different channels to be swapped. The former point carries the same intuition as the output anticipation for the binary case. The latter is a restriction that guarantees that subtyping preserves the ordering of observable events (input actions) given by the corresponding global type specification of the protocol. The definition of asynchronous subtyping is then given as follows:

**Definition 4.7 (Multiparty Asynchronous Subtyping,  $\leq_m$ ).**

Asynchronous subtyping for multiparty session types is defined as in Definition 2.4 where we consider a relation  $\mathcal{R}$  on both session and primitive types that satisfies the items in that definition, with items 2. and 3. replaced by the following corresponding ones, plus the additional items 5.–7. (also in this case, we use  $T$  and  $S$  to range over both session and primitive types):

Asynchronous subtyping for multiparty session types is defined as in Definition 4.2, by replacing the items about selection, branching, output and input with:

2. if  $T = k \oplus \{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0, \mathcal{A}^k$  such that
  - $\text{unfold}^n(S) = \mathcal{A}^k[k \oplus \{l_j : S_{h_j}\}_{j \in J_h}]^{h \in \{1, \dots, m\}}$ ,
  - $\forall h \in \{1, \dots, m\}. I \subseteq J_h$  and
  - $\forall i \in I. (T_i, \mathcal{A}^k[S_{h_i}]^{h \in \{1, \dots, m\}}) \in \mathcal{R}$ ;
3. if  $T = k \& \{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = k \& \{l_j : S_j\}_{j \in J}$ ,  $J \subseteq I$  and  $\forall j \in J. (T_j, S_j) \in \mathcal{R}$ ;
5. if  $T \in \{\text{bool} \mid \text{nat} \mid \dots\}$  then  $S = T$ ;
6. if  $T = k!\langle U \rangle.T'$  then  $\exists n \geq 0, \mathcal{A}^k$  such that
  - $\text{unfold}^n(S) = \mathcal{A}^k[k!\langle V_h \rangle.S_h]^{h \in \{1, \dots, m\}}$ ,
  - $\forall h \in \{1, \dots, m\}. (V_h, U) \in \mathcal{R}$  and
  - $(T', \mathcal{A}^k[S_h]^{h \in \{1, \dots, m\}}) \in \mathcal{R}$ ;
7. if  $T = k?(U).T'$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = k?(V).S'$  and  $(U, V) \in \mathcal{R}$  and  $(T', S') \in \mathcal{R}$ .

As for the other cases, subtyping is undecidable:

**Corollary 4.4.** *Asynchronous subtyping for multiparty session types is undecidable.*

*Proof.* Similarly to the case for session types with carried types in branching/selection, standard session types defined in Definition 4.1 have a one-to-one correspondence with a fragment of local types where only one single channel, e.g.,  $k$ , is used. Then, the thesis follows from the fact that on such fragment the two subtyping relations  $\leq_s$  and  $\leq_m$  are isomorphic.  $\square$

*Remark.* The proof of the previous Lemma does not directly work for the definition of asynchronous subtyping used by Mostrous et al. [3]. This is because their subtyping relation does not allow to anticipate outputs over inputs on the same channel, e.g.,  $k!\langle U \rangle.k?(U').T$  is not a subtype of  $k?(U').k!\langle U \rangle.T$ . However, output anticipation is possible over inputs on different channels, e.g.,  $k!\langle U \rangle.k'(U').T$  is a subtype of  $k'(U').k!\langle U \rangle.T$ , assuming  $k \neq k'$ . The subtyping algorithm proposed by Mostrous et al. [3] correctly checks cases like the above two examples, but fails to terminate when there is an unbounded accumulation of inputs as in the first example that we have discussed in §2.3. Rephrasing that example in the syntax of local types, we have that  $T = \mu t.k'(U').k!\langle U \rangle.t$  and  $S = \mu t.k'(U').k'(U').k!\langle U \rangle.t$  are in subtyping relation, even for Mostrous et al. [3]. Nevertheless, the algorithm proposed in that paper does not terminate because, in this case, it is expected to check infinitely many different pairs  $(T, k'(U').S)$ ,  $(T, k'(U').k'(U').S)$ ,  $\dots$ . In the light of our undecidability result, we can even conclude the impossibility to check algorithmically the subtyping relation by Mostrous et al. [3]. Consider the proof of our Theorem 3.1: given a queue machine, we can change both encodings of its finite control and its queue so that all inputs are on some special channel  $k$  and all outputs are on some special channel  $k'$ , with  $k \neq k'$ . As discussed above, outputs on  $k'$  can be anticipated w.r.t. inputs on a different channel  $k$ , hence the two encodings will be in a subtyping relation, also for the subtyping by Mostrous et al. [3], if and only if the encoded machine does not terminate.

#### 4.5. Communicating automata

A Communicating Finite State Machine (CFSM) [9], or more simply a communicating automaton, is defined as a finite automaton  $(Q, q_0, \Sigma, \delta)$ , where

- $Q$  is a finite set of states
- $q_0 \in Q$  is the initial state
- $\Sigma$  is a finite alphabet, and
- $\delta \subseteq Q \times \Sigma \times \{!, ?\} \times Q$  is a transition set.

We use “?” to represent inputs and “!” to represent outputs (in CFSMs [9] “+” and “−”, respectively, are used, instead).

A CFSM is a labeled transition system that can be employed to graphically represent a session type (see, e.g., Figures 2 and 3). Note that in general a CFSM may express more behaviours than the ones described by session types: it can include non-deterministic and mixed choices, i.e. choices including both inputs and outputs.

Let  $\mathcal{T}$  be the set of all session types  $T$  and  $L$  the alphabet of session types, we define a transition relation  $\longrightarrow \subseteq \mathcal{T} \times L \times \{!, ?\} \times \mathcal{T}$ , as the least transition set satisfying the following rules

$$\begin{aligned} \oplus \{l_i : T_i\}_{i \in I} &\xrightarrow{l_i!} T_i \quad i \in I \\ \&\{l_i : T_i\}_{i \in I} &\xrightarrow{l_i?} T_i \quad i \in I \\ \frac{T\{\mu\mathbf{t}.T/\mathbf{t}\} &\xrightarrow{\alpha} T'}{\mu\mathbf{t}.T \xrightarrow{\alpha} T'} \end{aligned}$$

with  $\alpha$  ranging over  $L \times \{!, ?\}$ .

Given a session type  $T$  we define  $CFSM(T)$  as being the communicating automaton  $(Q_T, T, L, \delta_T)$ , where:  $L$  is the alphabet of session types,  $Q_T$  is the set of terms  $T'$  which are reachable from  $T$  according to  $\longrightarrow$  relation and  $\delta_T$  is defined as the restriction of  $\longrightarrow$  to  $Q_T \times L \times \{!, ?\} \times Q_T$ . For example Figure 2 depicts  $CFSM(S)$  with  $S$  being the session type defined in Definition 3.7 (assuming  $\Gamma = \{A_i \mid i \leq k\}$ ).

We, thus, get the following result as a consequence of undecidability of  $\leq$ . Any relation  $\preceq$  over communicating automata (usually called *refinement* relation in this context) that is such that  $CFSM(T) \preceq CFSM(T')$  if and only if  $T \leq T'$ , i.e. it reduces to our subtyping definition for the subclass of communicating automata not including non-deterministic and mixed choices, is undecidable.

## 5. Decidable Fragments of Asynchronous Single-Choice Relation

We now show that we cannot further reduce (w.r.t. branching/selection structure) the core undecidable fragment: if we consider single-output selection only or single-input branching only, we obtain a decidable relation.

### Definition 5.1 (Asynchronous Single-Choice Output Relation).

The asynchronous single-choice output relation  $\ll_{\text{sout}}$  is defined as:

$$\ll_{\text{sout}} = \ll \cap (T^{\text{out}} \times T^{\text{out}})$$

### Definition 5.2 (Asynchronous Single-Choice Input Relation).

The asynchronous single-choice input relation  $\ll_{\text{sin}}$  is defined as:

$$\ll_{\text{sin}} = \ll \cap (T^{\text{in}} \times T^{\text{in}})$$

$$\begin{array}{c}
\overline{\Sigma, (T, S) \vdash T \leq_a S} \text{ Asmp} \qquad \overline{\Sigma \vdash \mathbf{end} \leq_a \mathbf{end}} \text{ End} \\
\\
\frac{\forall n. I \subseteq J_n \quad \forall i \in I. \Sigma \vdash T_i \leq_a \mathcal{A}[S_{ni}]^n}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j \in J_n}]^n} \text{ Out} \quad \frac{J \subseteq I \quad \forall j \in J. \Sigma \vdash T_j \leq_a S_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq_a \&\{l_j : S_j\}_{j \in J}} \text{ In} \\
\\
\frac{\Sigma, (\mu\mathbf{t}.T, S) \vdash \text{unfold}^1(\mu\mathbf{t}.T) \leq_a S}{\Sigma \vdash \mu\mathbf{t}.T \leq_a S} \text{ RecL} \\
\\
\frac{T = \mathbf{end} \vee T = \&\{l_i : T_i\}_{i \in I} \quad \Sigma, (T, \mu\mathbf{t}.S) \vdash T \leq_a \text{unfold}^1(\mu\mathbf{t}.S)}{\Sigma \vdash T \leq_a \mu\mathbf{t}.S} \text{ RecR}_1 \\
\\
\frac{n = \text{depth}(S, \emptyset) \quad n \geq 1 \quad \Sigma, (\oplus\{l_i : T_i\}_{i \in I}, S) \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \text{unfold}^n(S)}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a S} \text{ RecR}_2
\end{array}$$

Figure 4: A Procedure for Checking Subtyping

As a matter of fact, we prove decidability for larger relations w.r.t.  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$  where we do not impose the constraint about no consecutive infinite outputs.

In order to define an algorithm for deciding the two relations above, we first adapt to our setting the procedure defined in Mostrous et. al [3] and then improve it to precisely characterize the two relations. The initial procedure is defined for the unrestricted syntax of session types, i.e. the subtyping  $\leq$ ; while the improved version assumes to work on types restricted according to the single-choice assumptions, i.e. the two new relations  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$ . Actually, in order to have a more general decidability result, we show that it is not necessary to consider the constraint about no consecutive infinite outputs.

The procedure is defined by the rules reported in Figure 4. In the rules, the environment  $\Sigma$  is a set of pairs  $(T, S)$  used to keep track of previously visited pairs of types. The procedure successfully terminates either by applying rule **Asmp** (which has priority over the other rules) or by applying rule **End**. In the former case, a previously visited pair is being visited again, and therefore, there is no need to proceed further. Rules **In** and **Out** are straightforward. The procedure always unfolds on the left-hand side when necessary (rule **RecL**). If this is not the case, but it is necessary to unfold on the right-hand side, it is possible to apply either **RecR<sub>1</sub>** or **RecR<sub>2</sub>**, depending on whether the left-hand side type is an input (or an **end**) or an output, respectively. In the first case, a single unfolding is sufficient. However, we may need to unfold several times in the case we need an output. The partial function **depth** (we write  $\text{depth}(S) = \perp$  if **depth** is undefined on  $S$ ) measures the number of unfoldings necessary for



anticipating such output. The function is inductively defined as:

$$\begin{aligned} \text{depth}(\mathbf{end}, \Gamma) &= \perp & \text{depth}(\oplus\{l_i : T_i\}_{i \in I}, \Gamma) &= 0 \\ \text{depth}(\&\{l_i : T_i\}_{i \in I}, \Gamma) &= \max\{\text{depth}(T_i, \Gamma) \mid i \in I\} \\ \text{depth}(\mu\mathbf{t}.T, \Gamma) &= \begin{cases} \perp & \text{if } \mathbf{t} \in \Gamma \\ 1 + \text{depth}(T\{\mu\mathbf{t}.T/\mathbf{t}\}, \Gamma + \{\mathbf{t}\}) & \text{otherwise} \end{cases} \end{aligned}$$

In the definition of  $\text{depth}$ , we assume that  $\max\{\text{depth}(T_i, \Gamma) \mid i \in I\} = \perp$  if  $\text{depth}(T_i, \Gamma) = \perp$  for some  $i \in I$ . Similarly,  $1 + \perp = \perp$ .

The subtyping procedure, when it has to check whether  $T \leq S$ , applies the rules from bottom to top starting from the judgement  $\emptyset \vdash T \leq_a S$ . We write  $\Sigma \vdash T \leq_a S \rightarrow \Sigma' \vdash T' \leq_a S'$  if  $\Sigma \vdash T \leq_a S$  matches the consequences of one of the rules, and  $\Sigma' \vdash T' \leq_a S'$  is produced by the corresponding premises.  $\Sigma \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$  is the reflexive and transitive closure of such relation. We write  $\Sigma \vdash T \leq_a S \rightarrow_{\text{err}}$  to mean that no rule can be applied to the judgement  $\Sigma \vdash T \leq_a S$ . We give priority to the application of the rule **Asmp** to have a deterministic procedure: this is sufficient because all the other rules are alternative, i.e., given a judgement  $\Sigma \vdash T \leq_a S$  there are no two rules that can be both applied.

We now prove that the above procedure is a semi-algorithm for checking whether two types are not in subtyping relation.

**Lemma 5.1.** *Given the types  $T$  and  $S$  we have  $\exists \Sigma', T', S'. \emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$  if and only if  $T \not\leq S$ .*

*Proof.* We prove the two implications separately. We start with the *if* part and proceed by contraposition. Assume that it is not true that  $\exists \Sigma', T', S'. \emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ . Consider now the relation

$$\mathcal{R} = \{(T', S') \mid \exists \Sigma'. \emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'\}$$

We show that  $\mathcal{R}$  is a subtyping relation. Let  $(T', S') \in \mathcal{R}$ . Then, it is possible to apply at least one rule to  $\Sigma' \vdash T' \leq_a S'$  for the environment  $\Sigma'$  such that  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$ . We proceed by cases on  $T'$ . In the following we use  $\text{nrec}(R)$  to denote the number of unguarded –not prefixed by some branching/selection– occurrences of recursions  $\mu\mathbf{t}.R'$  in  $R$  for any  $R', \mathbf{t}$ .

- If  $T' = \mathbf{end}$  then item 1 of Definition 2.4 for pair  $(T', S')$  is shown by induction on  $k = \text{nrec}(S')$ .
  - Base case  $k = 0$ . The only rule applicable to  $\Sigma' \vdash T' \leq_a S'$  is **End**, that immediately yields the desired pair of  $\mathcal{R}$ .
  - Induction case  $k > 0$ . The only rules applicable to  $\Sigma' \vdash T' \leq_a S'$  are **Asmp** and **RecR<sub>1</sub>**. In the case of **Asmp** we have that  $(T', S') \in \Sigma'$ , hence there exists  $\Sigma''$  with  $(T', S') \notin \Sigma''$  such that  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma'' \vdash T' \leq_a S' \rightarrow^* \Sigma' \vdash T' \leq_a S'$  and rule **RecR<sub>1</sub>** has been applied to  $\Sigma'' \vdash T' \leq_a S'$ . So for some  $\Sigma''' (= \Sigma' \text{ or } = \Sigma'')$  we have

that the procedure applies rule  $\text{RecR}_1$  to  $\Sigma''' \vdash T' \leq_a S'$ . Hence  $\Sigma''' \vdash T' \leq_a S' \rightarrow \Sigma'''' \vdash T' \leq_a \text{unfold}^1(S')$ . Since  $\text{nrec}(\text{unfold}^1(S')) = k - 1$ , by induction hypothesis item 3 of Definition 2.4 holds for pair  $(T', \text{unfold}^1(S'))$ , 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  $\text{depth}(S, \emptyset) = 0$  then the only rule applicable to  $\Sigma' \vdash T' \leq_a S'$  is **Out**, that immediately yields the desired pairs of  $\mathcal{R}$ .
  - If  $\text{depth}(S, \emptyset) \geq 1$  then the only rules applicable to  $\Sigma' \vdash T' \leq_a S'$  are **Asmp** and **RecR<sub>2</sub>**. In the case of **Asmp** we have that  $(T', S') \in \Sigma'$ , hence there exists  $\Sigma''$  with  $(T', S') \notin \Sigma''$  such that  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma'' \vdash T' \leq_a S' \rightarrow^* \Sigma' \vdash T' \leq_a S'$  and rule **RecR<sub>2</sub>** has been applied to  $\Sigma'' \vdash T' \leq_a S'$ . So for some  $\Sigma'''$  ( $= \Sigma'$  or  $= \Sigma''$ ) we have that the procedure applies rule **RecR<sub>2</sub>** to  $\Sigma''' \vdash T' \leq_a S'$ . Hence  $\Sigma''' \vdash T' \leq_a S' \rightarrow \Sigma'''' \vdash T' \leq_a \text{unfold}^k(S')$ , taking  $k = \text{depth}(S, \emptyset)$ . Since  $\text{depth}(\text{unfold}^k(S')) = 0$ , we end up in the previous case. Therefore item 3 of Definition 2.4 holds for pair  $(T', \text{unfold}^k(S'))$ , 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  $k = \text{nrec}(S')$ :
  - Base case  $k = 0$ . The only rule applicable to  $\Sigma' \vdash T' \leq_a S'$  is **In**, that immediately yields the desired pairs of  $\mathcal{R}$ .
  - Induction case  $k > 0$ . The only rules applicable to  $\Sigma' \vdash T' \leq_a S'$  are **Asmp** and **RecR<sub>1</sub>**. In the case of **Asmp** we have that  $(T', S') \in \Sigma'$ , hence there exists  $\Sigma''$  with  $(T', S') \notin \Sigma''$  such that  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma'' \vdash T' \leq_a S' \rightarrow^* \Sigma' \vdash T' \leq_a S'$  and rule **RecR<sub>1</sub>** has been applied to  $\Sigma'' \vdash T' \leq_a S'$ . So for some  $\Sigma'''$  ( $= \Sigma'$  or  $= \Sigma''$ ) we have that the procedure applies rule **RecR<sub>1</sub>** to  $\Sigma''' \vdash T' \leq_a S'$ . Hence  $\Sigma''' \vdash T' \leq_a S' \rightarrow \Sigma'''' \vdash T' \leq_a \text{unfold}^1(S')$ . Since  $\text{nrec}(\text{unfold}^1(S')) = k - 1$ , by induction hypothesis item 3 of Definition 2.4 holds for pair  $(T', \text{unfold}^1(S'))$ , hence it holds for pair  $(T', S')$ .
- If  $T' = \mu t.T'$  then item 4 of Definition 2.4 for pair  $(T', S')$  holds because the only rule applicable to  $\Sigma' \vdash T' \leq_a S'$  is **RecL** that immediately yields the desired pair of  $\mathcal{R}$ .

We now prove the *only if* part and proceed by contraposition. Assume that there exists a relation  $\mathcal{R}$  that is a subtyping relation such that  $(T, S) \in \mathcal{R}$ .

We say that  $\Sigma \vdash T \leq_a S \rightarrow_w \Sigma' \vdash T' \leq_a S'$  if  $\Sigma \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$  and: the last rule applied is one of **Out**, **In** or **RecL** rules; while all previous ones are **RecR<sub>1</sub>** or **RecR<sub>2</sub>** rules.

We start by showing that if  $\exists \Sigma. \emptyset \vdash T \leq_a S \rightarrow_w^* \Sigma \vdash T' \leq_a S'$  implies  $\exists n. (T', \text{unfold}^n(S')) \in \mathcal{R}$ . The proof is by induction on the length of such

computation  $\rightarrow_w^*$  of the procedure. 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 holds for all computations of a length  $k$  and we show it holds for all computations of length  $k + 1$ , by considering all judgements  $\Sigma' \vdash T'' \leq_a S''$  such that  $\Sigma \vdash T' \leq_a S' \rightarrow_w \Sigma' \vdash T'' \leq_a S''$ . This is shown by first considering the case in which rule **Asmp** applies to  $\Sigma \vdash T' \leq_a S'$ : 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  $\Sigma \vdash \mathbf{end} \leq_a S' \rightarrow^* \Sigma''' \vdash \mathbf{end} \leq_a \mathbf{end}$  (by applying **RecR<sub>1</sub>** rules) and rule **End** applies to  $\Sigma''' \vdash \mathbf{end} \leq_a \mathbf{end}$ . Also in this case there is no such a judgement  $\Sigma' \vdash T'' \leq_a S''$  and there is nothing to prove. Finally, we proceed by an immediate verification that judgements  $\Sigma' \vdash T'' \leq_a S''$  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<sub>2</sub>** applications followed by **Out** application),  $T' = \&\{l_i : T_i\}_{i \in I}$  ( $\rightarrow_w$  is a possibly empty sequence of **RecR<sub>1</sub>** applications followed by **In** application) or  $T' = \mu t.T'$  ( $\rightarrow_w$  is simply **RecL** application).

We finally observe that, considered any judgement  $\Sigma \vdash T' \leq_a S'$  such that  $\exists n. (T', \text{unfold}^n(S')) \in \mathcal{R}$ , we have:

- either rule **Asmp** applies to  $\Sigma \vdash T' \leq_a S'$ , or
- $T' = \mathbf{end}$  and, by item 1 of Definition 2.4, there exists  $\Sigma'$  such that  $\Sigma \vdash \mathbf{end} \leq_a S' \rightarrow^* \Sigma' \vdash \mathbf{end} \leq_a \mathbf{end}$  (by applying **RecR<sub>1</sub>** rules) and rule **End** is the unique rule applicable to  $\Sigma' \vdash \mathbf{end} \leq_a \mathbf{end}$ , with **RecR<sub>1</sub>** being the unique rule applicable to intermediate judgements, or
- by items 2,3 and 4 of Definition 2.4, there exist  $\Sigma', T'', S''$  such that  $\Sigma \vdash T' \leq_a S' \rightarrow_w^* \Sigma' \vdash T'' \leq_a S''$ , with each intermediate judgement having a unique applicable rule. In particular this holds for  $T' = \oplus\{l_i : T_i\}_{i \in I}$  ( $\rightarrow_w$  is a possibly empty sequence of **RecR<sub>2</sub>** applications followed by **Out** application),  $T' = \&\{l_i : T_i\}_{i \in I}$  ( $\rightarrow_w$  is a possibly empty sequence of **RecR<sub>1</sub>** applications followed by **In** application) or  $T' = \mu t.T'$  ( $\rightarrow_w$  is simply **RecL** application).  $\square$

The above procedure is not guaranteed to terminate when  $T \leq S$ , in particular in those cases in which an infinite subtyping relation is needed to prove that they are in subtyping. For instance, this happens for the cases discussed in the examples reported in Section 2.3. We now show how to amend the procedure to terminate at least in the restricted cases discussed above. The new algorithm is defined by the same rules as before, plus a pair of rules presented below, where we use judgements  $\Sigma \vdash T \leq_t S$  instead of  $\Sigma \vdash T \leq_a S$ .

Moreover, the new version of the algorithm requires to distinguish among different instances of the same input branching. More precisely, due to multiple unfoldings we could have that the same input choice appears more than once. For instance, given  $\mu t. \&\{l : t\}$ , if we apply unfolding twice, we obtain  $\&\{l : \&\{l : \mu t. \&\{l : t\}\}\}$ . To distinguish different instances of the same input branching, we assume to have an extended syntax in which such choices are

annotated:  $\&^\alpha\{l : S\}$  denotes an input annotated with a symbol  $\alpha$  taken from a countable set of annotations. Annotations in the same term are assumed to be pairwise distinct. In the example above, the decorated version of the unfolded term is  $\&^\alpha\{l : \&^{\alpha'}\{l : \mu\mathbf{t}.\&\{l : \mathbf{t}\}\}\}$ , for a pair of distinct annotations  $\alpha$  and  $\alpha'$ . As multiple unfoldings can be applied only to the r.h.s. terms of the judgements  $\Sigma \vdash T \leq_t S$ , we will adopt the syntax extended with annotations only for such terms.

Algorithmically, in order to have the guarantee that all the annotations are pairwise distinct, we assume that they are all different in the initial term. Namely, when we want to check whether  $T \ll_{\text{sout}} S$  or  $T \ll_{\text{sin}} S$  we assume an annotation function  $\text{ann}(S)$  that annotates all the input choices in  $S$  with pairwise distinct symbols, and we start the algorithm from the judgement  $\emptyset \vdash T \leq_t \text{ann}(S)$ . Moreover, every time an unfolding is applied to the r.h.s. of a judgement, the algorithm annotates with fresh symbols each added input choice.

Concerning annotations, we omit them when they are irrelevant. For instance, the rules in Figure 4 do not contain annotations, but are used any way to define also the new algorithm (upon replacement of  $\Sigma \vdash T \leq_a S$  with  $\Sigma \vdash T \leq_t S$ ). The omission of the annotations means that the rules in that figure can be applied to any annotated judgement  $\Sigma \vdash T \leq_t S$ , to obtain a new judgement  $\Sigma' \vdash T' \leq_t S'$  where the new annotated term  $S'$  inherits the annotations of  $S$  and in case new input branchings are present in  $S'$  (due to unfolding in rules  $\text{RecR}_1$  and  $\text{RecR}_2$ ) these will be freshly annotated as discussed above.

We assume a function  $\text{unann}(S)$  that returns the same term but without annotations. We overload the function  $\text{unann}(\_)$  to apply it also to an environment  $\Sigma$ :  $\text{unann}(\Sigma)$  is the environment obtained by removing all annotations from the r.h.s. of all of its pairs.

We are now ready to present the two additional rules, having the same higher priority of **Asmp**:

$$\frac{l_1 \cdots l_m = \gamma^j \cdot (l_1 \cdots l_s) \quad j > i \quad s < |\gamma| \quad l_1 \cdots l_n = \gamma^i \cdot (l_1 \cdots l_s) \quad S \in \{\oplus, \mu\} \quad \& \in T \quad \text{unann}(S) = \text{unann}(S')}{\Sigma, (T, \&\{l_1 : \dots \&\{l_r : \dots \&\{l_n : S\} \dots\} \dots\}) \vdash T \leq_t \&^\alpha\{l_1 : \dots \&\{l_m : S'\} \dots\}} \text{Asmp2}$$

$$\frac{S \in \{\oplus, \mu\} \quad \& \notin T \quad n < m \quad \text{unann}(S) = \text{unann}(S')}{\Sigma, (T, \&\{l_1 : \dots \&\{l_n : S\} \dots\}) \vdash T \leq_t \&\{l_1 : \dots \&\{l_m : S'\} \dots\}} \text{Asmp3}$$

Above  $S \in \{\oplus, \mu\}$  means that  $S$  starts with either an output selection or a recursive definition, while  $\& \in T$  requires the occurrence of at least one input branching anywhere in the term  $T$ .

Intuitively, **Asmp2** allows the algorithm to terminate when, after the judgement  $\Sigma \vdash T \leq_t R$ , another judgement  $\Sigma' \vdash T \leq_t R'$  is reached such that:

- both  $R$  and  $R'$  start with a sequence of input branchings (respectively labeled with  $l_1 \cdots l_n$  and  $l_1 \cdots l_m$ );

- the label sequences are repetitions of the same pattern, with the second one strictly longer than the first one (namely, there exist  $\gamma$ , a proper prefix  $l_1 \cdots l_s$  of  $\gamma$  and  $j > i$  such that  $l_1 \cdots l_n = \gamma^i \cdot (l_1 \cdots l_s)$  and  $l_1 \cdots l_m = \gamma^j \cdot (l_1 \cdots l_s)$ );
- there exists  $l_r$  in the initial input sequence of  $R$  having an annotation  $\alpha$  that coincides with the annotation of  $l_1$  in the initial input sequence of  $R'$  (hence  $l_1 = l_r$ ,  $n \geq 1$  and  $m \geq 1$ );
- after such initial input branchings, both  $R$  and  $R'$  continue with the same term up to annotations ( $S$  and  $S'$  such that  $\text{unann}(S) = \text{unann}(S')$ ).

The algorithm can terminate in this case because otherwise it would continue indefinitely by repeating the same steps performed between the judgements  $\Sigma \vdash T \leq_t R$  and  $\Sigma' \vdash T \leq_t R'$ . During these steps the l.h.s. term  $T$  performs a cycle including: the non-empty sequence of inputs  $l_1 \cdots l_{r-1}$  (due to the requirement above about the  $\alpha$  annotation and the constraint  $\& \in T$  in the rule premise) that is a repetition of  $\gamma$ , which is matched by the initial part of the input sequence in the r.h.s.; and a sequence of outputs, which is matched by the final term  $S$  of the r.h.s. that, itself, performs a cycle. While performing this cycle,  $S$  generates new input branchings that strictly extend the initial input sequence.

The rule **Asmp3** allows the algorithm to terminate in the case the potential infinite repetition includes output selections only. In this case it is sufficient to check that the l.h.s. cycles without consuming any inputs and that, after a number of such cycles: the amount of initial accumulated inputs in the r.h.s. strictly increases and the term after such accumulation performs, itself, a cycle.

Also for the new algorithm we use  $\Sigma \vdash T \leq_t S \rightarrow^* \Sigma' \vdash T' \leq_t S'$  to denote the application of one rule on the former judgement that generates the latter, and  $\Sigma \vdash T \leq_t S \rightarrow_{\text{err}}$  to denote that no rule can be applied to the judgement  $\Sigma \vdash T \leq_t S$ .

We now prove that the new algorithm terminates.

**Lemma 5.2.** *Given two types  $T \in T^{\text{out}} \cap T^{\text{in}}$  (resp.  $T \in T^{\text{out}}$ ) and  $S \in T^{\text{in}}$  (resp.  $S \in T^{\text{in}} \cap T^{\text{out}}$ ), the algorithm applied to initial judgement  $\emptyset \vdash T \leq_t \text{ann}(S)$  terminates.*

*Proof.* In this proof, we abstract away from the annotations of input actions, i.e., we denote two types that differ only in the annotations with the same term.

We proceed by contraposition. Assume that there exist  $T$  and  $S$  such that the algorithm starting from the initial judgement  $\emptyset \vdash T \leq_t S$  does not terminate. Hence, there must exist an infinite sequence of rule applications  $\Sigma \vdash T \leq_t S \rightarrow \Sigma_1 \vdash T_1 \leq_t S_1 \rightarrow \dots \rightarrow \Sigma_n \vdash T_n \leq_t S_n \rightarrow \dots$ . In this sequence, infinitely many unfoldings of recursive definitions will be performed on infinitely many traversed judgements  $\Sigma' \vdash T' \leq_t S'$ . All these pairs  $(T', S')$  must be pairwise distinct otherwise rule **Asmp** will be applied to terminate the sequence. This implies that the environment  $\Sigma$  grows unboundedly in order to contain all these infinitely many distinct pairs  $(T', S')$ .

We now prove that all such pairs  $(T', S')$  are of the following form:  $(T_f, \&\{l_1 : \dots \&\{l_n : S_f\} \dots\})$  where  $T_f$  and  $S_f$  belong to a finite set of terms. The first term  $T'$  is obtained from the initial term  $T$  by means of consumptions of initial inputs or outputs (rules **In** and **Out**), or single unfoldings of top level recursive definitions (rules **RecL**). The set of terms that can be obtained in this way from the finite initial term  $T$  is clearly finite. On the contrary, the second group of terms  $S'$  could be obtained by application of different transformations: in particular **Out** that allows for the anticipation of outputs prefixed by an arbitrary sequence of inputs, and **RecR<sub>2</sub>** that can unfold more than one recursive definition at a time. Concerning the multiple unfoldings of **RecR<sub>2</sub>**, we have that the **depth**( $\_$ ) function guarantees that recursive definitions guarded by an output operation are never unfolded. In this way a subterm of  $S'$  prefixed by an output is taken from a finite set of terms. Obviously, also the set of possible subterms starting with a recursive definition is finite as well. As  $S'$  has only input single-choices, we can conclude that it is of the form  $\&\{l_1 : \dots \&\{l_n : S_f\} \dots\}$  because it starts with a (possibly empty) sequence of inputs followed by a term  $S_f$ , guarded by an output or a recursive definition, taken from a finite set.

As infinitely many distinct pairs  $(T', S')$  are introduced in  $\Sigma$ , there are infinitely many distinct pairs having the same  $T_f$  and  $S_f$ . This guarantees the existence of an infinite sequence

$$\begin{aligned} &(T_f, \&\{l_1^1 : \dots \&\{l_{n_1}^1 : S_f\} \dots\}) \\ &(T_f, \&\{l_1^2 : \dots \&\{l_{n_2}^2 : S_f\} \dots\}) \\ &\dots \\ &(T_f, \&\{l_1^v : \dots \&\{l_{n_v}^v : S_f\} \dots\}) \\ &\dots \end{aligned}$$

of pairs that are introduced in  $\Sigma$  in this order, and for which  $n_v$  is strictly growing.

If  $T_f$  does not contain input actions (i.e.  $\& \notin T_f$ ) we have that the rule **Asmp3** could be applied when the second pair in the above sequence  $(T_f, \&\{l_1^2 : \dots \&\{l_{n_2}^2 : S_f\} \dots\})$  was introduced in  $\Sigma$ . As **Asmp3** has priority, it is necessary to apply this rule, thus terminating successfully the sequence of rule applications. This contradicts the initial assumption about the infinite sequence of rule applications.

Now, consider  $T_f$  that contains at least one input action. We separate this case in two subcases:  $T_f \in T^{\text{out}} \cap T^{\text{in}}$  and  $T_f \in T^{\text{out}}$ .

In the first subcase, both inputs and outputs in  $T_f$  are single-choices. This means that, by cycling,  $T_f$  performs indefinitely always the same sequence of input branchings: let  $\gamma$  be the sequence of the corresponding labels. The inputs accumulated on the r.h.s. term should be consistent, i.e. must have labels of the form  $l_1^v \dots l_{n_v}^v = \gamma^i \cdot \iota$ , with  $\iota$  prefix of  $\gamma$ . Consider now the final part  $\iota$  of the sequences  $l_1^v \dots l_{n_v}^v$  that, as discussed above, are all prefixes of  $\gamma$ ; as the different prefixes of  $\gamma$  are finite, we have that there exists an infinite subsequence of pairs  $(T_f, \&\{l_1^v : \dots \&\{l_{n_v}^v : S_f\} \dots\})$  all having the same final  $\iota$ . Consider now *maxIn* as the maximal number of inputs between two subsequent

outputs in  $S_f$  (or its unfolding). We select from this infinite subsequence a pair  $(T_f, \&\{l_1^k : \dots \&\{l_{n_k}^k : S_f\} \dots\})$  with  $n_k > \max In$ . We now consider the subsequent pair in the subsequence  $(T_f, \&\{l_1^w : \dots \&\{l_{n_w}^w : S_f\} \dots\})$  and the corresponding sequence of rule applications:

$$\Sigma_k \vdash T_f \leq_t \&\{l_1^k : \dots \&\{l_{n_k}^k : S_f\} \dots\} \rightarrow^*$$

$$\Sigma_w \vdash T_f \leq_t \&\{l_1^w : \dots \&\{l_{n_w}^w : S_f\} \dots\}$$

Let  $l'_1 \dots l'_h$  be the labels of the outputs involved in applications of the rule **Out**. Such outputs are present in  $S_f$  (and its unfoldings). Let  $\&\{l''_1 : \dots \&\{l''_g : S'\} \dots\}$  (assuming  $S' \in \{\oplus, \mu\}$ ) be the term obtained from  $S_f$  by (unfolding the term and) consuming the outputs labeled with  $l'_1 \dots l'_h$ . We have that  $S' = S_f$ . We conclude considering two possible cases:  $g \leq |\iota|$  and  $g > |\iota|$ .

- If  $g \leq |\iota|$ , we have that all the applications of the rule **In** involve inputs that are already present in the initial sequence of inputs of the r.h.s. in  $(T_f, \&\{l_1^k : \dots \&\{l_{n_k}^k : S_f\} \dots\})$ . This guarantees that it is possible to apply on the judgement  $\Sigma_f \vdash T_f \leq_t \&\{l_1^w : \dots \&\{l_{n_w}^w : S_f\} \dots\}$  the rule **Asmp2**. As **Asmp2** has priority it is necessary to apply this rule thus terminating successfully the sequence of rule applications. This contradicts the initial assumption about the infinite sequence of rule applications.

- If  $g > |\iota|$ , we have that  $l''_1 \dots l''_g = \iota' \cdot \gamma^y \cdot \iota$  with  $\iota \cdot \iota' = \gamma$ . From the infinite subsequence we select a pair  $(T_f, \&\{l''_1 : \dots \&\{l''_{n_r} : S_f\} \dots\})$  such that  $n_r - |\iota|$  is greater than the number of applications of the rule **In** in the sequence of rule applications:

$$\Sigma_k \vdash T_f \leq_t \&\{l_1^k : \dots \&\{l_{n_k}^k : S_f\} \dots\} \rightarrow^*$$

$$\Sigma_w \vdash T_f \leq_t \&\{l_1^w : \dots \&\{l_{n_w}^w : S_f\} \dots\}$$

Consider now the same sequence of rule applications starting from the judgement  $\Sigma_r \vdash T_f \leq_t \&\{l''_1 : \dots \&\{l''_{n_r} : S_f\} \dots\}$  that introduced  $(T_f, \&\{l''_1 : \dots \&\{l''_{n_r} : S_f\} \dots\})$  in the environment. Let

$$\Sigma_q \vdash T_f \leq_t \&\{l''_1 : \dots \&\{l''_{n_q} : S''\} \dots\}$$

be the reached judgement. We have that on this judgement it is possible to apply **Asmp2** because  $S'' = S_f$  and  $l''_1 \dots l''_{n_q} = \gamma^u \cdot \iota \cdot \iota' \cdot \gamma^y \cdot \iota$ . As discussed in the previous case this contradicts the initial assumption on the infinite sequence of rule applications.

It remains the final subcase  $T_f \in T^{\text{out}}$ . In this case we have  $S_f \in T^{\text{in}} \cap T^{\text{out}}$ . As  $T_f$  has outputs with single-choice, the rule **In** will be applied at least once in the sequence of rule applications between every pair and the subsequent one. For this reason, it is not restrictive to assume that in all the pairs  $(T_f, \&\{l_1^v : \dots \&\{l_{n_v}^v : S_f\} \dots\})$  the inputs labeled with  $l_1^v \dots l_{n_v}^v$  are produced by previous unfoldings of the same term  $S_f$ . As in  $S_f$  all the inputs and outputs are single-choice, there is a predefined sequence of inputs within a cycle from  $S_f$  to  $S_f$  again. Let  $\eta$  be the sequence of the labels corresponding to such inputs. Then we have that the inputs accumulated on the l.h.s. term, labeled with the sequence of labels  $l_1^v \dots l_{n_v}^v$ , will be such that  $l_1^v \dots l_{n_v}^v = \rho \eta^i$ , for some  $\rho$  suffix of  $\eta$ . We consider a suffix  $\rho$  because during the execution of the algorithm some previously accumulated input will be consumed to mimick inputs of the l.h.s.

term, and  $\rho$  represents the part of inputs that have been left from a sequence  $\rho$  that has been only partially consumed. As the suffixes of  $\eta$  are finite, we can extract an infinite subsequence of pairs that always consider the same suffix  $\rho$ . Namely, for all the pairs, the second term has an initial sequence of inputs with a sequence of labels belonging to  $\rho\eta^*$ , for the same  $\rho$ . But as  $\rho$  is a suffix of  $\eta$  there exists a rotation  $\gamma'$  of  $\eta$  such that all the sequences belong to  $\gamma'^*\rho'$  with  $\rho \cdot \rho' = \eta$ . Hence all the sequences  $l_1^v \cdots l_{n_v}^v$  are of the form  $\gamma'^i \cdot \rho'$ . By considering  $\gamma = \gamma'$  and  $\iota = \rho'$  we can now conclude with the same arguments used in the previous subcase.  $\square$

We now move to the proof of soundness of the algorithm.

**Lemma 5.3.** *Given two types  $T \in T^{out} \cap T^{in}$  (resp.  $T \in T^{out}$ ) and  $S \in T^{in}$  (resp.  $S \in T^{in} \cap T^{out}$ ), we have that  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{err}$  if and only if  $\emptyset \vdash T \leq_t \text{ann}(S) \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{err}$ .*

*Proof.* We consider the two implications separately starting from the *if* part. Assume that  $\emptyset \vdash T \leq_t \text{ann}(S) \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{err}$ . In this sequence of rule applications, the new rules **Asmp2** and **Asmp3** are never used otherwise the sequence terminates successfully by applying such rules. Hence, by applying the same sequence of rules, we have  $\emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$  with  $T'' = T'$ ,  $\text{unann}(S'') = S'$  and  $\text{unann}(\Sigma'') = \Sigma'$ . We have that  $\Sigma' \vdash T' \leq_a S' \rightarrow_{err}$ , otherwise if a rule could be applied to this judgement, the same rule could be applied also to  $\Sigma'' \vdash T'' \leq_t S''$  thus contradicting the assumption  $\Sigma'' \vdash T'' \leq_t S'' \rightarrow_{err}$ .

We now move to the *only if* part. Assume the existence of the sequence of rule applications  $\rho_a = \emptyset \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{err}$ . As the algorithm for  $\emptyset \vdash T \leq_t \text{ann}(S)$  considers a superset of rules, we have two possible cases: either  $\emptyset \vdash T \leq_t \text{ann}(S) \rightarrow^* \Sigma'' \vdash T'' \leq_t S'' \rightarrow_{err}$  by applying the same sequence of rules, or during the application of this sequence of rules, starting from  $\emptyset \vdash T \leq_t \text{ann}(S)$ , a judgement is reached on which one of the additional rules **Asmp2** or **Asmp3** can be applied. Namely, there exists a sequence  $\rho_t = \emptyset \vdash T \leq_t \text{ann}(S) \rightarrow^* \Sigma_e \vdash T_e \leq_t S_e$ , corresponding to a prefix of  $\rho_a$ , such that either **Asmp2** or **Asmp3** can be applied on the judgement  $\Sigma_e \vdash T_e \leq_t S_e$ . We conclude the proof by showing that this second case never occurs. We discuss only **Asmp2**, as the case for **Asmp3** is treated similarly.

If  $\emptyset \vdash T \leq_t S \rightarrow^* \Sigma_e \vdash T_e \leq_t S_e$  and **Asmp2** can be applied to  $\Sigma_e \vdash T_e \leq_t S_e$ , there exists an intermediary judgement  $\Sigma_s \vdash T_s \leq_t S_s$ , traversed during such sequence of rule applications, that introduces in the environment the pair  $(T_s, S_s)$  used in the above application of the rule **Asmp2**. Hence  $\rho_t$  has a suffix  $\Sigma_s \vdash T_s \leq_t S_s \rightarrow^* \Sigma_e \vdash T_e \leq_t S_e$  with:

- $T_s = T_e$ ,
- $S_s = \&\{l_1 : \dots \&\{l_n : R\} \dots\}$  and  $S_e = \&\{l_1 : \dots \&\{l_m : R'\} \dots\}$  with  $\text{unann}(R) = \text{unann}(R')$ ,  $l_1 \cdots l_n = \gamma^i \cdot (l_1 \cdots l_s)$ ,  $l_1 \cdots l_m = \gamma^j \cdot (l_1 \cdots l_s)$  for  $i < j$  and  $s < |\gamma|$ ;



- during the entire sequence  $\Sigma_s \vdash T_s \leq_a S_s \rightarrow^* \Sigma_e \vdash T_e \leq_a S_e$  only a prefix  $l_1 \cdots l_{r-1}$  of the input actions  $l_1 \cdots l_n$  is consumed from  $S_s$ .

Let  $\Sigma'_s = \text{unann}(\Sigma_s)$ ,  $S'_s = \text{unann}(S_s)$ ,  $\Sigma'_e = \text{unann}(\Sigma_e)$ , and  $S'_e = \text{unann}(S_e)$ . As  $\rho_t$  corresponds to a prefix of  $\rho_a$  we have that  $\rho_a = \emptyset \vdash T \leq_a S \rightarrow^* \Sigma'_s \vdash T_s \leq_a S'_s \rightarrow^* \Sigma'_e \vdash T_e \leq_a S'_e \rightarrow^* \Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ . This is not possible because we now show that after the sequence  $\Sigma'_s \vdash T_s \leq_a S'_s \rightarrow^* \Sigma'_e \vdash T_e \leq_a S'_e$ , it is not possible to reach any judgement  $\Sigma' \vdash T' \leq_a S'$  such that  $\Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ . On the basis of the observations listed above we have:

- $T_s = T_e$ ,
- $S'_s = \&\{l_1 : \dots \&\{l_n : R\} \dots\}$ ,  $S'_e = \&\{l_1 : \dots \&\{l_m : R\} \dots\}$  with  $l_1 \cdots l_n = \gamma^i \cdot (l_1 \cdots l_s)$ ,  $l_1 \cdots l_m = \gamma^j \cdot (l_1 \cdots l_s)$  for  $i < j$  and  $s < |\gamma|$ ;
- during the entire sequence  $\Sigma'_s \vdash T_s \leq_a S'_s \rightarrow^* \Sigma'_e \vdash T_e \leq_a S'_e$  only a prefix  $l_1 \cdots l_{r-1}$  of the input actions  $l_1 \cdots l_n$  is consumed from  $S'_s$ .

We have that the sequence of rules applied in  $\Sigma'_s \vdash T_s \leq_a S'_s \rightarrow^* \Sigma'_e \vdash T_e \leq_a S'_e$  is the unique one that can be applied also to the ending judgement  $\Sigma'_e \vdash T_e \leq_a S'_e$ . This is guaranteed by the fact that we are considering single-choice subtyping, the correspondence of the terms  $T_s = T_e$ , the availability of the input actions labeled with  $l_1 \cdots l_{r-1}$  in  $S'_e$ , and the fact that both  $S'_s$  and  $S'_e$  have the same term  $R$  at the end of their initial input actions. Consider now the judgement  $\Sigma''_e \vdash T'_e \leq_a S''_e$  reached at the end of such sequence, i.e.  $\Sigma'_e \vdash T_e \leq_a S'_e \rightarrow^* \Sigma''_e \vdash T'_e \leq_a S''_e$ . We have that the same properties listed above hold also for the new sequence  $\Sigma'_e \vdash T_e \leq_a S'_e \rightarrow^* \Sigma''_e \vdash T'_e \leq_a S''_e$ :

- $T_e = T'_e$ , because the same transformations are applied to the l.h.s. terms by the rules that are applied.
- $S'_e = \&\{l_1 : \dots \&\{l_m : R\} \dots\}$  and  $S''_e = \&\{l_v : \dots \&\{l_m : R\} \dots\}$  with  $l_1 \cdots l_m = \gamma^j \cdot (l_1 \cdots l_s)$  and  $l_1 \cdots l_v = \gamma^{(j+(j-i))} \cdot (l_1 \cdots l_s)$ , where  $i$  is the number of the repetitions of  $\gamma$  in the initial r.h.s. term  $S'_s$ , hence  $j - i$  is the number of new repetitions of  $\gamma$  added during the sequence of rule applications. From the previous properties we have  $j > i$ , hence  $j < j + (j - i)$ , and  $s < |\gamma|$ .
- During the entire sequence  $\Sigma'_e \vdash T_e \leq_a S'_e \rightarrow^* \Sigma''_e \vdash T'_e \leq_a S''_e$  only a prefix  $l_1 \cdots l_{r-1}$  of the input actions  $l_1 \cdots l_m$  is consumed from  $S'_e$ .

As these properties continue to hold, we have that the sequence of rules applied in  $\Sigma'_e \vdash T_e \leq_a S'_e \rightarrow^* \Sigma''_e \vdash T'_e \leq_a S''_e$  can be continued to be applied indefinitely, hence it is not possible to reach any judgement  $\Sigma' \vdash T' \leq_a S'$  such that  $\Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$ .  $\square$

We can finally conclude with the following theorem that states decidability for more general versions of  $\ll_{\text{sin}}$  and  $\ll_{\text{sout}}$ , where we do not impose related types to belong to  $T^{\text{noinf}}$ .

**Theorem 5.1 (Algorithm Correctness).** *Given two types  $T \in T^{\text{out}} \cap T^{\text{in}}$  (resp.  $T \in T^{\text{out}}$ ) and  $S \in T^{\text{in}}$  (resp.  $S \in T^{\text{in}} \cap T^{\text{out}}$ ), we have that  $\emptyset \vdash T \leq_t \text{ann}(S)$  if and only if  $T \leq S$ .*

*Proof.* It follows immediately from Lemma 5.1, Lemma 5.2 and Lemma 5.3.  $\square$

As an obvious consequence of algorithm correctness, we have that the two relations  $\ll_{\text{sout}}$  and  $\ll_{\text{sin}}$  are decidable.

**Corollary 5.1.** *The asynchronous single-choice output relation  $\ll_{\text{sout}}$  and the asynchronous single-choice input relation  $\ll_{\text{sin}}$  are decidable.*

*Proof.* In order to verify whether  $T \ll_{\text{sin}} S$  it is sufficient to check that  $T \in T^{\text{in}} \cap T^{\text{out}} \cap T^{\text{noinf}}$  and  $S \in T^{\text{in}} \cap T^{\text{noinf}}$  and then verify whether  $T \leq S$ , which is decidable for terms belonging to these sets as proved in Theorem 5.1. For  $T \ll_{\text{sout}} S$  it is possible to proceed in the same way with the only difference that the check is whether  $T \in T^{\text{out}} \cap T^{\text{noinf}}$  and  $S \in T^{\text{in}} \cap T^{\text{out}} \cap T^{\text{noinf}}$ .  $\square$

Notice that with respect to the general case, when our algorithm is applied to the restricted case of  $\ll_{\text{sout}}$  and  $\ll_{\text{sin}}$ , rule **Asmp3** and premise  $\& \in T$  of rule **Asmp2** become useless. This follows from the fact that types in  $T^{\text{noinf}}$  never satisfy the premises of **Asmp3** and always satisfy the premise  $\& \in T$  of rule **Asmp2**.

## 6. Conclusion and Related Work

*Related Work.* Lange and Yoshida [10] have independently and simultaneously provided an undecidability result for a class of communicating automata called asynchronous duplex systems (which are shown to correspond to a class of binary session types). They prove that automata compatibility (checking whether two automata in parallel can safely interact) is undecidable and then show that such result makes also asynchronous session types subtyping undecidable. Their proof consists of an encoding of the termination problem for Turing machines (rather than our simpler and direct encoding based on queue machines) into deciding automata compatibility. Most importantly, in order to prove undecidability of a subtyping  $\preceq$ , a translation  $\mathcal{M}$  from types to automata is used which exploits the following result (Theorem 5.1, [10]):

Given two types  $T$  and  $S$ , we have that  $T \preceq S$  if and only if  $\mathcal{M}(T)$  is compatible with  $\mathcal{M}(\overline{S})$ .

Above,  $\overline{S}$  is the dual of  $S$ , the type obtained from  $S$  by inverting inputs (outputs) with outputs (inputs). As compatibility between communicating automata is a symmetric relation, we have that this approach to prove undecidability of subtyping can be applied only to dual closed relations  $\preceq$ , i.e.,  $T \preceq S$  if and only if  $\overline{S} \preceq \overline{T}$ . Among all the subtyping relations in the literature (discussed in details in Section 4), this property holds only for the definition of subtyping by Chen et al. [2], where orphan messages are not admitted. For instance, we have that the type  $\mu t. \oplus \{l : t\}$  is a subtype of  $\mu t. \& \{l' : \oplus \{l : t\}\}$  but obviously

$\mu\mathbf{t}.\oplus\{l' : \&\{l : \mathbf{t}\}\}$  is not a subtype of  $\mu\mathbf{t}.\&\{l : \mathbf{t}\}$ . Moreover, differently from Lange and Yoshida [10] we show that, to get undecidability, it is sufficient to consider a restricted version of asynchronous binary session subtyping ( $\ll$  relation) that is much less expressive and that cannot be further simplified by imposing limitations on the branching/selection structure of types (otherwise it becomes decidable). As shown, this has one plain advantage: it allowed us to easily show undecidability of various existing subtyping relations.

Concerning decidability, Lange and Yoshida [10] independently and simultaneously proved a result similar to ours (Theorem 5.1). They present an algorithm for deciding subtyping between types  $T$  and  $S$ , with one between  $T$  and  $S$  being a single-choice session type (i.e. a type where every branching/selection has a single choice) and the other one being a general (any) session type. In particular, they consider a dual closed subtyping relation defined following the orphan-message free approach by Chen et al. [2]. On the other hand, we show (Theorem 5.1)  $T \leq S$  to be decidable when: one between  $T$  and  $S$  is a single-choice session type (as for Lange and Yoshida) and the other one is either a type with single outputs, if it is  $T$ , or a type with single inputs, if it is  $S$ . We first notice that our  $\leq$  relation is more general with respect to the one used by Lange and Yoshida in that it does not include the orphan message free constraint. Moreover, although it may seem that Lange and Yoshida can effectively relate types that do not fall under our (more restricted) syntactical characterization, covariance and contravariance prevent any type containing at least one non-single input branch and one non-single output selection (both reachable in the subtyping simulation game) to be related with a single-choice type. We finally observe that since, differently from the relation used by Lange and Yoshida, our  $\leq$  relation is not dual closed, we need to explicitly carry out two separate proofs for the two cases of  $T \leq S$ : one where the single-choice session type is  $T$  and another one where it is, instead,  $S$ .

*Conclusion.* We have proven that asynchronous subtyping for session types is undecidable. Moreover, we have shown that subtyping becomes decidable if we put some restrictions on the branching/selection structure. As future work, we plan to search for alternative subtyping relations that enjoy properties similar to  $\leq$ , but are decidable.

## References

## References

- [1] D. Mostrous, N. Yoshida, Session typing and asynchronous subtyping for the higher-order  $\pi$ -calculus, *Inf. Comput.* 241 (2015) 227–263.
- [2] 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.

- [3] 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.
- [4] 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.
- [5] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, *J. ACM* 63 (1) (2016) 9.
- [6] S. J. Gay, M. Hole, Subtyping for session types in the pi calculus, *Acta Inf.* 42 (2-3) (2005) 191–225.
- [7] D. Mostrous, Session types in concurrent calculi: Higher-order processes and objects, Ph.D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine (2009).
- [8] D. Kozen, Automata and computability, Springer, New York, 1997.
- [9] D. Brand, P. Zafiropulo, On communicating finite-state machines, *Journal of the ACM* 30 (2) (1983) 323–342.
- [10] J. Lange, N. Yoshida, On the undecidability of asynchronous session subtyping, in: 20th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2017), Vol. 10203 of Lecture Notes in Computer Science, 2017, pp. 441–457.