Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Session Types for Asynchronous Communication: a New Subtyping and Its Implementation

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Correlatore:
Prof. Julien Lange

Presentata da:
Teresa Signati

III Sessione
Anno Accademico 2019/2020

Abstract

Session types are a promising way to describe communication protocols directly through the type system, allowing to check the correctness of a system at compile time. This thesis represents a study of session subtyping, i.e., the substitutability of components, starting from the work by Bravetti, Lange, and Zavattaro, who presented a definition of fair asynchronous subtyping [3]. Their subtyping allows the anticipation of messages emissions without the restrictions imposed by previous results on this field. Since covariance of outputs is not allowed by the original version of subtyping, we try to address this problem and to introduce covariance into the original proposal of fair asynchronous subtyping [3]. Finally, we describe the integration process of the new definition of subtyping within the tool [4] for the check of subtyping relation.

Contents

Chapter 1

An Overview of Session Types

The always growing importance of network computing and of programming based on communication among processes led to the introduction of communication primitives and the birth of new programming languages and formalism to deal with the need of a readable and performant way to represent interactions between several processes. Concurrent and distributed communications have to face several issues, such as disagreements between senders and receivers, deadlocks and orphan messages. The increasing importance of distributed systems led to the search for a solution, that can be found in session types.

1.1 The Idea of Session Types

The importance of type systems comes from the possibility of verifying that a program is well-behaved by checking that it is well-typed. Traditionally, type systems have been focused on checking the possible outcome of computations, on *what* the result of the computation should be. During the '90s, new notions of typing allowed the description of properties associated with the behaviour of programs through the type systems, focusing on *how* the computation proceeded. The latter are usually referred to as behavioural types [11].

Session types fall within the more general concept of behavioural types, that allow to represent the evolution of the computation directly within the type system. A successful typechecking ensures the correct interaction between the components of the system.

The first proposals of session types date back to the 1990s by Takeuchi, Honda, and Kubo [16] and Honda, Vasconcelos, and Kubo [10], who introduced session types as a formalized solution in π-calculus to communication problems at compile time, without having to face them at execution time.

A session represents a logic unit of information exchange between several parts [7] specifying messages' sequence and direction.

From the point of view of each part of the communication, a session type can be seen as a protocol in its own perspective.

The basic constructs are the message exchange operations: !`bool` denotes the send, the output of a boolean value, instead ?`bool` denotes its receive, the input operation. Sequencing is represented by . and the termination of the protocol, after which no further interaction is possible, is denoted by **end**.

An example of session type can be

!`bool`.?`nat`.**end**

where the message exchange starts with the output of a boolean value, followed by the input of a natural number, followed by the completion of the protocol.

Intuitively the evolution of the communication can be graphically represented through UML sequence diagrams, as in Figure 1.1, representing the communication between a client and a server. In this example, the server computes the multiplication or the division between two given numbers, depending on the required operation.

Arrows represent the direction of each message and they can be solid lines, if they represent a possible option, like multiplication, division, quotient and error, or dashed lines if they identify related data.

From the UML sequence diagram, it is possible to retrieve a global description of the system, as follows

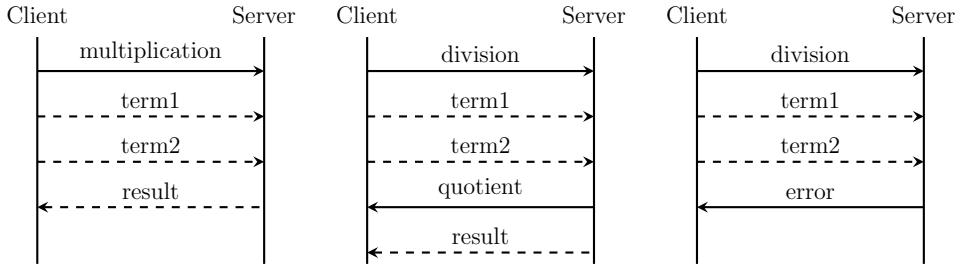


Figure 1.1: UML sequence diagram of possible interactions between a client and a server

```
Client → Server:
     { multiplication: Client → Server: term1.
                       Client → Server: term2.
                       Server → Client: result.
                       end.
      [] division: Client → Server: term1.
                   Client → Server: term2.
                   Server → Client:
                         { quotient: Server → Client: result.
                                       end
                         [] error: end
                         }
      }
```

Options that can be chosen at a given instant are surrounded by curly brackets and are internally separated by []. For each label, messages and their direction are specified.

Replacing data by their actual types (term1 by Integer, term2 by Integer, and so on) gives the global type.

A local view of each part of the communication is obtained by replacing arrows with send and receive operation, represented by ! ($\oplus$ in case of a multiple choice) and ? (& in case of a multiple choice) respectively.

The local description from the Client's point of view is as follows:

```
⊕ {  multiplication: !term1.!term2.?result.end
   [] division: !term1.!term2.& { quotient:?result.end
                                [] error: end
                                }
   }
```

instead the one from the Server's point is:

```
& { multiplication: ?term1.?term2.!result.end
   [] division: ?term1.?term2.⊕ { quotient:!result.end
                                  [] error: end
                                  }
   }
```

that is the dual form of the Client's one.

In this example, the duality of session types ensures the correctness (deadlock absence, no orphan messages, etc.) of the communication. Because of the strictness of duality prerequisite, the idea of subtyping for session types came up. If a type T' is subtype of a type T, written $T' \leq T$, it can safely replace T preserving the correctness of the communication.

1.2 Syntax and Semantics

Before reasoning on subtyping and refinement, it is necessary to recall the syntax of binary [1] session types [2].

Definition 1.2.1 (Session type syntax)**.** Given a set of labels L, ranged over by I, the syntax of two-party session types is given by the following grammar:

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

$\oplus\{l_i : T_i\}_{i \in I}$ represents an internal choice, an output selection. One of the labels l_i is selected and sent over the channel and then the continuation T_i is executed. $\&\{l_i : T_i\}_{i \in I}$ is the corresponding semantic for the input branching, representing an external choice, so, one of the labels l_i is received over the channel and then the continuation T_i is executed. In both internal and external choice, labels are assumed to be pairwise distinct.

[1] Binary or two-party sessions are a particular case of multiparty session types in which the number of participants is not fixed to two. In the following, only binary session types are taken into account.

[2] The notation used further omits the simplified constructors for sending an output $!l$ and for receiving an input $?l$ for sake of simplicity. They can be represented by the multiple internal (respectively external) constructor, where the size of the set of labels is fixed to one.

The type constructors $\mu\mathbf{t}.T$ and $\mathbf{t}$ are used to express recursion. Recursive variables are bound by the $\mu\mathbf{t}$ preceding the type T, within the recursive variable $\mathbf{t}$ occurs. **end** corresponds to the termination of the execution.

In order to unfold the recursive definition in a session type T, it is necessary to recall the unfold function below.

Definition 1.2.2 (Unfold)**.** Given a session type T

$$\mathsf{unfold}(T) = \begin{cases} \mathsf{unfold}(T'\{T/t\}) & \text{if } T = \mu\mathbf{t}.T', \\ T & \text{otherwise} \end{cases}$$

where $T'\{T/t\}$ represents the replacement in T' of every free occurrence of $\mathbf{t}$ by T.

Definition 1.2.3 (Dual of session type)**.** The dual of session type T, written $\overline{T}$, is defined as follows:

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

These definitions will be exploited in the following chapter, in the presentation of the subtyping idea.

Chapter 2

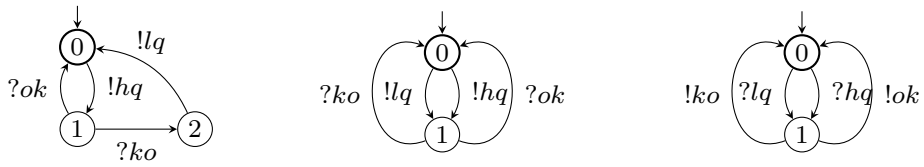
On Previous Definitions of Subtyping

In some cases it can be useful to replace a session type with another one for efficiency and performance improvements. This replacement operation can be done if the new type is a subtype of the previous one. The aim of subtyping is replacing a session type with another one, preserving the correctness of the system.

Before discussing the new proposal of subtyping, the state of the art will be discussed in this chapter, highlighting the differences between the synchronous case and the asynchronous one, which is the more interesting form for its closeness to the actual implementation of concurrent and distributed systems. In this chapter the main focus will be on the definition of fair asynchronous subtyping [3] introduced by Bravetti, Lange, and Zavattaro, which is the starting point for the new definition proposed in the following chapter.

2.1 Session Subtyping

As discussed in the previous chapter, the protocol defined by a session type and its dual is correct but in practice this constraint is too strict. The idea of session subtyping tries to solve this problem.



T_R $T_C = \overline{T_S}$ T_S

$$
\begin{aligned}
T_R &= \mu \mathbf{t}. \oplus \{hq : \&\{ok : \mathbf{t}, ko : \oplus\{lq : \mathbf{t}\}\}\} \\
T_C &= \mu \mathbf{t}. \oplus \{lq : \&\{ok : \mathbf{t}, ko : \mathbf{t}\}, hq : \&\{ok : \mathbf{t}, ko : \mathbf{t}\}\} \\
T_S &= \mu \mathbf{t}.\&\{lq : \oplus\{ok : \mathbf{t}, ko : \mathbf{t}\}, hq : \oplus\{ok : \mathbf{t}, ko : \mathbf{t}\}\}
\end{aligned}
$$

Figure 2.1: Video streaming protocol. T_R is the refined session type of the client T_C, and T_S is the partner, the session type of the server.

In the case of synchronous session subtyping, the subtype can perform fewer internal choices (sends) and more external choices (receives) than its supertype, as shown by the rules below.

$$\frac{}{\mathbf{end} \leq \mathbf{end}}$$

$$\frac{\forall i \in I : T_i \leq T_i'}{\&_{i \in I \cup J}\{l_i : T_i\} \leq \&_{i \in I}\{l_i : T_i'\}}$$

$$\frac{\forall i \in I : T_i \leq T_i'}{\oplus_{i \in I}\{l_i : T_i\} \leq \oplus_{i \in I \cup J}\{l_i : T_i'\}}$$

The asynchronous case is more interesting, because of the non-blocking send actions and the possibility of anticipating send actions in the subtype if they don't affect the partner.

To present the asynchronous session subtyping, it is useful to discuss an example, like the one of a video streaming service [5], shown in Figure 2.1.

T_S represents a server, that can receive high($?hq$) or low($?lq$) quality requests, and replies with $!ok$ if the request can be fulfilled, with $!ko$ otherwise and then it returns to the initial state.

T_C represents a client, that is the dual of the server T_S, as expected by binary session types without subtyping.

A possible improvement of the client protocol can be represented by T_R, that requires the high quality streaming first ($!hq$) and, only if the request cannot be fulfilled ($?ko$), it requires the low quality version ($!lq$).

T_R is an asynchronous subtype of T_C, because the subtype is able to receive the same messages of T_C and messages sent by T_R can also be sent by T_C, so the parallel composition of T_S and T_R, written as $T_S \mid T_R$, is correct.

2.2 Asynchronous Session Typing

Asynchronous session calculus can be considered as an extension of the synchronous one with FIFO queues [6]. A queue is used to enqueue received messages and to dequeue messages that must be read.

Henceforth, a sequence of incoming messages is represented by a queue ω, i.e. an unbounded buffer that ranges over words in $\mathcal{L}^*$. ϵ stands for the empty word. The word $\omega_1 \cdot \omega_2$ represents the concatenation of words ω_1 and ω_2. In the asynchronous case, configurations of the states in the transition systems have to provide the sequence of incoming messages, ω_i, along with the session types, written $[T_1, \omega_1]||[T_2, \omega_2]$.

Definition 2.2.1 (Transition Relation)**.** The transition relation $\rightarrow$ over configurations is the minimal relation satisfying the rules below (plus symmetric ones):

1. if $j \in I$ then $[\oplus\{l_i : T_i\}_{i \in I}, \omega_1]||[T_2, \omega_2] \rightarrow [T_j, \omega_1]||[T_2, \omega_2 \cdot l_j]$
2. if $j \in I$ then $[\&\{l_i : T_i\}_{i \in I}, l_j \cdot \omega_1]||[T_2, \omega_2] \rightarrow [T_j, \omega_1]||[T_2, \omega_2]$
3. if $[\mathsf{unfold}(T_1), \omega_1]||[T_2, \omega_2] \rightarrow s$ then $[T_1, \omega_1]||[T_2, \omega_2] \rightarrow s$

The transition relation $\rightarrow^*$ is the reflexive and transitive closure of the $\rightarrow$ relation.

A configuration s reduces to s' when (1) one type sends a message to the other, adding it to its queue; (2) one type consumes a message from the head of its queue; (3) the unfolding of a type can perform one of the transitions above. Successful configurations are the ones where both types reached an **end** and both queues are empty.

Definition 2.2.2 (Successful configuration)**.** A successful configuration, written $s\surd$, is defined as follows:

$$[T, \omega_T]|[S, \omega_S]\surd \text{ iff } \mathsf{unfold}(T) = \mathsf{unfold}(S) = \mathbf{end} \text{ and } \omega_T = \omega_S = \epsilon$$

Definition 2.2.3 (Correct composition)**.** Given a configuration s, it is a correct composition if, whenever $s \rightarrow^* s'$, there exists a configuration s'' such that $s' \rightarrow^* s''$ and $s''\surd$.

Definition 2.2.4 (Compliance)**.** Two session types are compliant if $[T, \epsilon]|[S, \epsilon]$ is a correct composition.

Definition 2.2.4 is a strong definition of compliance [3], because all the sent messages are received and both the types reach the termination, i.e. an **end** state. According to this definition, compliance does not hold for all the pairs type T and dual one $\overline{T}$.

For example, let $T = \oplus\{a : \mathbf{end}, b : \mu\mathbf{t}.\&\{c : \mathbf{t}\}\}$ and its dual $\overline{T} = \&\{a : \mathbf{end}, b : \mu\mathbf{t}. \oplus \{c : \mathbf{t}\}\}$. T and $\overline{T}$ are not compliant because, when T sends b, the configuration $[\mathbf{end}, \epsilon]|[\mathbf{end}, \epsilon]$ is not reachable anymore.

Definition 2.2.5 (Refinement)**.** A session type T refines S, $T \sqsubseteq S$, if for every S' s.t. S and S' are compliant then T and S' are also compliant.

Differently from traditional subtyping relation, this refinement notion is not covariant [3].

If $T = \mu\mathbf{t}. \oplus \{a : \mathbf{t}\}$ and $S = \oplus\{a : \mathbf{t}, b : \mathbf{end}\}$, T is a subtype of S, due to output covariance, but it is not a refinement, because it exists a type $\overline{S} = \mu\mathbf{t}.\&\{a : \mathbf{t}, b : \mathbf{end}\}$, that is compliant with S but not with T, since T cannot reach an **end**.

2.3 CFSM Representation

Before discussing the notion of fair asynchronous refinement, it is necessary to recall the correspondence between session types and communicating finite-state machines (CFSMs) [12], that are fundamental in the explanation of the algorithm [3] for verifying the subtyping relation by Bravetti, Lange, and Zavattaro. Thanks to this characterisation of session types as CFSMs, it is possible to exploit directly CFSMs in order to solve the subtyping verification problem.

Let $\mathbb{A}$ be a finite alphabet, let words be in $\mathbb{A}^*$ and let $\cdot$ be the concatenation operator. Let the set of actions be $Act = \{!, ?\} \times \mathbb{A}$, in order to express send and receive operations, respectively. The direction of an operation, $dir(\ell)$, is defined as $dir(!a) \stackrel{\text{def}}{=} !$ and $dir(?b) \stackrel{\text{def}}{=} ?$.

Definition 2.3.1 (Communication machine)**.** A communicating machine M is a tuple (Q, q_0, δ) where Q is the (finite) set of states, $q_0 \in Q$ is the initial state, and $\delta \in Q \times Act \times Q$ is the transition relation such that $\forall q, q', q'' \in Q, \forall \ell, \ell' \in Act$:

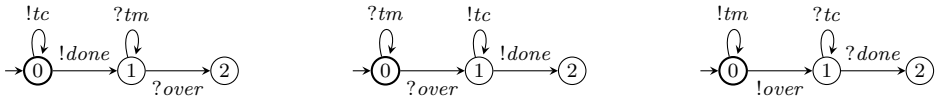
1. $(q, \ell, q'), (q, \ell', q'') \in \delta \implies dir(\ell) = dir(\ell')$

2. $(q, \ell, q'), (q, \ell, q'') \in \delta \implies q' = q''$

The relation $\rightarrow^*$ represents the reflexive transitive closure of $\rightarrow$.

A state $q \in Q$ is final, written $q \not\rightarrow$, iff $\forall q' \in Q, \forall \ell \in Act, (q, \ell, q') \notin \delta$. A state $q \in Q$ is *sending* (respectively *receiving*) iff q is not final and $\forall q' \in Q, \forall \ell \in Act, (q, \ell, q') \in \delta, dir(\ell) = !$ (respectively dir$(\ell) = ?$). The dual of a communicating machine M, written $\overline{M}$, is like M, with the difference that each sending transition, $(q, !a, q') \in \delta$, is replaced by the corresponding receiving one, $(q, ?a, q')$, and vice-versa for receive transitions.

To transform a session type in automaton, it is sufficient to take its labelled transition system according to an operational semantics, that can be



T'_G $T_G = \overline{T_S}$ T_S

$$
\begin{aligned}
T'_G &= \mu \mathbf{t}.\, \oplus \{tc : \mathbf{t}, done : \mu \mathbf{t}'.\, \&\{tm : \mathbf{t}', over : \mathbf{end}\}\} \\
T_G &= \mu \mathbf{t}.\, \&\{tm : \mathbf{t}, over : \mu \mathbf{t}'.\, \oplus \{tc : \mathbf{t}', done : \mathbf{end}\}\} \\
T_S &= \mu \mathbf{t}.\, \oplus \{tm : \mathbf{t}, over : \mu \mathbf{t}'.\, \&\{tc : \mathbf{t}', done : \mathbf{end}\}\}
\end{aligned}
$$

Figure 2.2: Satellite protocols. T'_G is the refined session type of the ground station, T_G is the session type of ground station, and T_S is the session type of the spacecraft.

defined essentially by two rules:

$$
\begin{aligned}
&\text{if } \mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i \in I} \text{ then } T \xrightarrow{!l_i} T_i && \forall i \in I \\
&\text{if } \mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I} \text{ then } T \xrightarrow{?l_i} T_i && \forall i \in I
\end{aligned}
$$

This conversion allows to reason on session types through their equivalent CFSM representation [12].

2.4 On Fair Asynchronous Subtyping

Taking into account the syntax of types, session subtyping aims to characterise the refinement property, that is defined on the basis of their operational semantics.

The main problem of the definitions antecedent to the one by Bravetti, Lange, and Zavattaro is the inefficiency of the protocols, in which no more than one party does a send action at any time. In this case, the communication is defined as half-duplex. The example provided by Bravetti, Lange, and Zavattaro is about a satellite protocol (see Figure 2.2).

T_S represents a spacecraft that sends some telemetries (tm), and then an *over* message and, after that, it receives some telecommands (tc) until a message *done* is received.

Consider as partner its dual, $\overline{T_S} = T_G$, that receives some telemetries until an over message is received and, after that, sends some telecommands, followed by a *done* message.

Allowing send actions by more than one party, i.e. allowing a full-duplex communication, is the key for systems like the one in the example, where there is an intermittent communication, e.g. the two parts are not always visible.

The idea of Bravetti, Lange, and Zavattaro was to introduce a new definition that formally guarantees that T'_G is a safe replacement for T_G, in which there is an output anticipation, even if the outputs were preceeded by an unbounded number of input loops.

2.4.1 Controllability

To introduce the notion of fair asynchronous subtyping [3], Bravetti, Lange, and Zavattaro introduced an algorithmic definition of controllability in an asynchronous context to check the existence of a session type that is compliant with the given one, according to Definition 2.2.4.

Definition 2.4.1 (Characterisation of controllability, $T\,\mathsf{ctrl}$)**.** Given a session type T, the judgement $T\,\mathsf{ok}$ is defined inductively as follows:

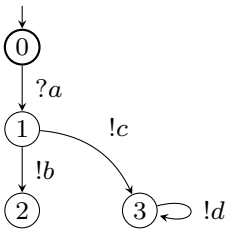
$$\frac{}{\mathbf{end}\,\mathsf{ok}} \qquad \frac{\mathbf{end} \in T \qquad T\{\mathbf{end}/\mathbf{t}\}\,\mathsf{ok}}{\mu\mathbf{t}.T\,\mathsf{ok}} \qquad \frac{T\,\mathsf{ok}}{\&\{l : T\}\,\mathsf{ok}} \qquad \frac{\forall i \in I.\ T_i\,\mathsf{ok}}{\oplus\{l_i : T_i\}_{i\in I}\,\mathsf{ok}}$$

where $\mathbf{end} \in T$ holds if **end** occurs in T.

It is possible to write $T\,\mathsf{ctrl}$ if there exists T' such that (i) T' is obtained from T by syntactically replacing every input prefix $\&\{l_i : T_i\}_{i\in I}$ occurring in T with a term $\&\{l_j : T_j\}$ (with $j \in I$) and (ii) $T'\,\mathsf{ok}$ holds.

Theorem 2.4.1. $T\,\mathsf{ctrl}$ holds if and only if there exists a session type S such that T and S are compliant.

If a session type is not controllable, there exists no session type with which it is compliant.



$$T = \&\{a : \oplus\{b : end, c : \mu\mathbf{t}.\oplus\{d : \mathbf{t}\}\}\}$$

Figure 2.3: Example of uncontrollable type.

An example of uncontrollable type is the one in Figure 2.3. Thanks to the controllability algorithm it can be derived that type T of the example is uncontrollable, so there is no session type S that is compliant with T.

2.4.2 Fair Asynchronous Subtyping

The main reference for this master project is the notion of fair asynchronous subtyping [3] by Bravetti, Lange, and Zavattaro. To introduce this definition they had to define a new notion of unfolding [1].

Definition 2.4.2 (Selective Unfolding)**.** Given a term T, $\mathsf{selUnfold}(T) =$

$$\begin{cases} \oplus\{l_i : T_i\}_{i\in I} & \text{if } T = \oplus\{l_i : T_i\}_{i\in I} \\ \&\{l_i : \mathsf{selUnfold}(T_i)\}_{i\in I} & \text{if } T = \&\{l_i : T_i\}_{i\in I} \\ T'\{^{\mu\mathbf{t}.T'}/_{\mathbf{t}}\} & \text{if } T = \mu\mathbf{t}.T',\ \oplus g(\mathbf{t}, T') \\ \mu\mathbf{t}.\mathsf{selUnfold}(\mathsf{selRepl}(\mathbf{t}, \hat{\mathbf{t}}, T')\{^{\mu\mathbf{t}.T'}/_{\hat{\mathbf{t}}}\}) \textit{ with } \hat{\mathbf{t}} \textit{ fresh} & \text{if } T = \mu\mathbf{t}.T',\ \neg \oplus g(\mathbf{t}, T') \\ \mathbf{t} & \text{if } T = \mathbf{t} \\ \mathbf{end} & \text{if } T = \mathbf{end} \end{cases}$$

where, $\mathsf{selRepl}(\mathbf{t}, \hat{\mathbf{t}}, T')$ is obtained from T' by replacing the free occurrences of $\mathbf{t}$ that are inside a subterm $\oplus\{l_i : S_i\}_{i\in I}$ of T' by $\hat{\mathbf{t}}$.

[1] The predicate $\oplus g(\mathbf{t}, T)$ holds if all instances of variable $\mathbf{t}$ are output selection guarded.

The asynchronous case allows the possibility of output anticipations. The inputs that can be delayed in the candidate supertype are usually referred to as asynchronous context or input context. For fair asynchronous refinement, because of the reasons explained through the example in Figure 2.2, the definition of input context includes recursive constructs, in contrast to previous results in this field.

Definition 2.4.3 (Input Context)**.** An input context $\mathcal{A}$ is a session type with holes defined by the syntax:

$$\mathcal{A} \ ::= \quad []^k \quad | \quad \&\{l_i : \mathcal{A}_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.\mathcal{A} \quad | \quad \mathbf{t}$$

where the holes $[]^k$, with $k \in K$, of an input context $\mathcal{A}$ are assumed to be pairwise distinct. Recursion is assumed to be guarded, i.e., in an input context $\mu\mathbf{t}.\mathcal{A}$, the recursion variable $\mathbf{t}$ must occur within a subterm $\&\{l_i : \mathcal{A}_i\}_{i\in I}$.

The set of hole indices in $\mathcal{A}$ is denoted by $holes(\mathcal{A})$. Given a type T_k for each $k \in K$, $\mathcal{A}[T_k]^{k\in K}$ is the type obtained by filling each hole k in $\mathcal{A}$ with the corresponding T_k.

At this point it is possible to introduce the definition of fair asynchronous subtyping, that corresponds to playing a simulation game between a candidate subtype T and its candidate supertype S.

Definition 2.4.4 (Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu\mathbf{t}.T'$ then $(T'\{^T/_{\mathbf{t}}\}, S) \in \mathcal{R}$;
3. if $T = \&\{l_i : T_i\}_{i\in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j\in J}$, $I \supseteq K$, and $\forall k \in K.\, (T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;
4. if $T = \oplus\{l_i : T_i\}_{i\in I}$ then $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_i : S_{ki}\}_{i\in I}]^{k\in K}$ and $\forall i \in I.\, (T_i, \mathcal{A}[S_{ki}]^{k\in K}) \in \mathcal{R}$.

T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.
T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.

The idea behind Definition 2.4.4 is to play a so-called subtyping simulation game, in order to check if T is a valid replacement for S, as follows.

Case (1) says that if T is the **end** type, S must be **end** too.

Case (2) says that if T is a recursive definition, T performs an unfolding and S does not need to reply, so the game proceeds.

Case (3) says that if T is an input branching, the controllable sub-terms in S can reply by inputting *some* of the labels l_i in the branching, in accordance with the contravariance of inputs, so the game proceeds.

Case (4) says that if T is an output selection, S can reply by outputting *all* the labels l_i in the selection, so the game proceeds.

As it is possible to notice from the requirements of case (4), covariance of outputs is not allowed, because the set of labels of the candidate subtype and the candidate supertype must be the same (see Chapter 3 for the attempts of covariance introduction).

The fair asynchronous subtyping is sound but not complete with respect to fair refinement. For example, let $T = \oplus\{a : \&\{c : \mathbf{end}\}\}$ and $S = \&\{c : \oplus\{a : \mathbf{end}, b : \mathbf{end}\}\}$. T is a refinement but not a fair asynchrnous subtype of S, since $\{a\} \neq \{a, b\}$, i.e. output covariance is not allowed.

Because of the undecidability of the problem, the search for algorithms that were at least sound but could give an *unknown* result, began (see Chapter 4 for the algorithm proposed by Bravetti, Lange, and Zavattaro).

Chapter 3

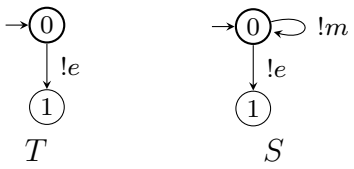
Covariance Introduction

The main goal of this thesis is, starting from Definition 2.4.4, to define a new variant of fair asynchronous subtyping that admits some kind of covariance.

In this chapter, all the attempts of modifications of Definition 2.4.4, that were done during the development of this thesis, will be shown. After discussing the failed attempts, we arrive to the last one, which we demonstrated to be successful. Changes to the original version of fair asynchronous subtyping by Bravetti, Lange, and Zavattaro are hightlighted to clearly show the adjustments that each attempt would bring. A counter-example is shown for each failed attempt, instead, the proof of its correctness is provided for the successful one.

3.1 Attempt 1

The attempt on which we worked most of the time is enclosed in Definition 3.1.2, that allows a reduction of the set of output labels through the possible removal of output self loops in the subtype. For this attempt, we implemented the correspondent solution and we discovered only in the end that it was unsound. If this definition was sound, types like the ones in Figure 3.1 would be in subtyping relation. Since the controllability check was deeply used



$$T = \oplus\{e : \mathbf{end}\}$$
$$S = \mu\mathbf{t}.\oplus\{m : \mathbf{t}, e : \mathbf{end}\}$$

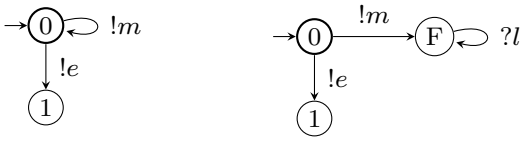
Figure 3.1: Example of subtyping ($T \leq S$) that would be allowed by the first attempt.

by this attempt, the adaptation of the tool to Definition 3.1.2 allowed us to discover a bug in the implementation, that will be discussed further in Subsection 4.2.2. In order to present this subtyping, it was necessary to introduce a new definition of a selective unfolding, slightly different from Definition 2.4.2 in case of internal choice.

Definition 3.1.1 (μ-Selective Unfolding)**.** We define a variant of selective unfolding, denoted with $\mathsf{selUnfold}'(T)$, which is defined inductively as $\mathsf{selUnfold}(T)$ with the difference that in case $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i\in I}$, it always returns T, also in case T starts with recursive definitions.

Definition 3.1.2 (Attempt 1 - Variant of Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu\mathbf{t}.T'$ then $(T'\{^{T}/_{\mathbf{t}}\}, S) \in \mathcal{R}$;
3. if $T = \&\{l_i : T_i\}_{i\in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j\in J}$, $I \supseteq K$, and $\forall k \in K.\,(T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;
4. if $T = \oplus\{l_i : T_i\}_{i\in I}$ then
   - $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j\in J_k}]^{k\in K}$,
   - $\mathsf{selUnfold}'(S) = \mathcal{A}[\mu\tilde{\mathbf{t}}_{\mathbf{k}}.\oplus\{l_j : S'_{kj}\}_{j\in J_k}]^{k\in K}$,



T T'

$$T = \mu \mathbf{t}. \oplus \{m : \mathbf{t}, e : \mathbf{end}\}$$
$$T' = \oplus \{m : \mathit{fail}, e : \mathbf{end}\}$$

Figure 3.2: Example of introduction of a failure state in place of a self loop output branch, given by $!m$.

- $\forall k \in K.\ I \subseteq J_k$ and $\forall j \in J_k \setminus I.\ S'_{kj}\{^{\mathit{fail}}/_{\tilde{\mathbf{t}}_\mathbf{k}}\}$ is uncontrollable,
- $\forall i \in I.\ (T_i, \mathcal{A}[S_{ki}]^{k \in K}) \in \mathcal{R}$.

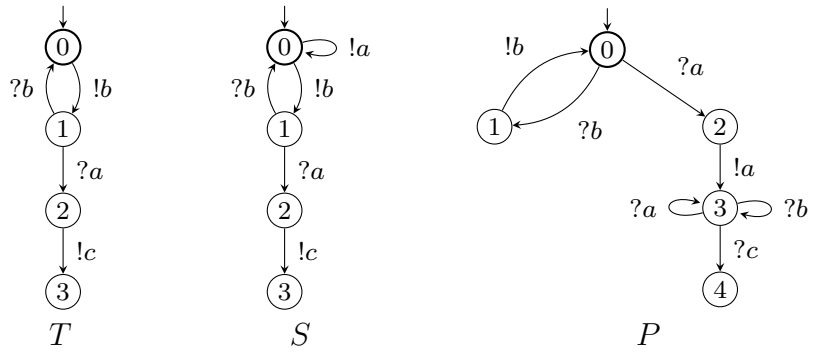
where given the sequence of variables $\tilde{\mathbf{t}} = \mathbf{t}_1 \ldots \mathbf{t}_n$, we use $\mu\tilde{\mathbf{t}}.T$ to denote $\mu\mathbf{t_1}. \ldots \mu\mathbf{t_n}.T$, *fail* is any uncontrollable session type (e.g. $\mu\mathbf{t}.\&\{l : \mathbf{t}\}$), and $\{^{\mathit{fail}}/_{\tilde{\mathbf{t}}}\}$ is the substitution of all free occurrences of variables in $\tilde{\mathbf{t}}$ with *fail*.

T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.

T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.

The implementation of this subtyping definition was possible thanks to the *fail* notion graphically explained in Figure 3.2. In an actual implementation, the addition of a failure concept is realised by adding another state, that can perform only receive loops, and by redirecting the excluded edges to this state.

Unfortunately, Definition 3.1.2 turned out to be unsound, as shown by the counter-example to its soundess in Figure 3.3.



$$
\begin{aligned}
T &= \mu\mathbf{t}.\oplus\{b : \&\{a : \oplus\{c : \mathbf{end}\}, b : \mathbf{t}\}\} \\
S &= \mu\mathbf{t}.\oplus\{a : \mathbf{t}, b : \&\{a : \oplus\{c : \mathbf{end}\}, b : \mathbf{t}\}\} \\
P &= \mu\mathbf{t}.\&\{b : \oplus\{b : \mathbf{t}\}, a : \oplus\{a : \mu\mathbf{t}'.\&\{a : \mathbf{t}', b : \mathbf{t}', c : \mathbf{end}\}\}\}
\end{aligned}
$$

Figure 3.3: Counter-example to soundness of the first attempt of covariance introduction. S is compliant with P, T should be a subtype of S according to 3.1.2 but T is not compliant with P.

3.2 Attempt 2

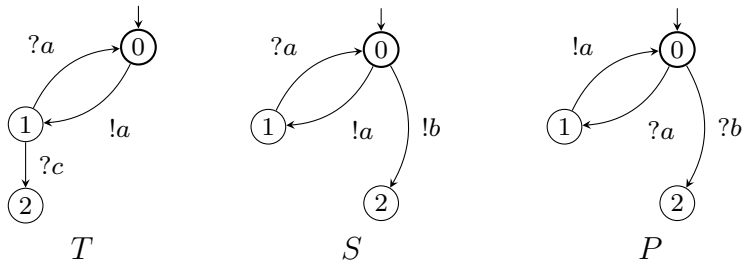
Another unsafe option, on which we briefly reasoned about while working on the first attempt, is the following one, that is shown only for sake of completeness.

The idea was to take into consideration the subtype and to check

- the respect of output covariance between the subtype and the supertype,
- the controllability of the subtype.

Definition 3.2.1 (Attempt 2 - Variant of Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu\mathbf{t}.T'$ then $(T'\{^{T}/_{\mathbf{t}}\}, S) \in \mathcal{R}$;



$$
\begin{aligned}
T &= \mu \mathbf{t}. \oplus \{a : \&\{a : \mathbf{t}, c : \mathbf{end}\}\} \\
S &= \mu \mathbf{t}. \oplus \{a : \&\{a : \mathbf{t}\}, b : \mathbf{end}\} \\
P &= \mu \mathbf{t}.\&\{a : \oplus\{a : \mathbf{t}\}, b : \mathbf{end}\}
\end{aligned}
$$

Figure 3.4: Counter-example to soundness of the second attempt of covariance introduction. S is compliant with P, T should be a subtype of S according to 3.2.1 but T is not compliant with P.

3. if $T = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j \in J}$, $I \supseteq K$, and $\forall k \in K. (T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;

4. if $T = \oplus\{l_i : T_i\}_{i \in I}$ then

   - $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in K}$,
   - T is controllable,
   - $\forall k \in K.\ I \subseteq J_k$,
   - $\forall i \in I. (T_i, \mathcal{A}[S_{ki}]^{k \in K}) \in \mathcal{R}$.

T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.
T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.

In this case it was easier to find a counter-example, that is shown in Figure 3.4. Here the problem came from combining the removal of output-labels with the addition of input ones. The addition of an input label, from which an **end** is reachable, ensures the respect of the controllability constraint of

the subtype, but this branch will never be executed, so an **end** is actually unreachable.

3.3 Attempt 3

While reasoning on the first attempt, a sort of union of the first and the second attempt led to the third one, that turned out to be unsafe too. The problem of the second attempt was checking the controllability only on the subtype. To solve this problem, the idea was to move the focus from the subtype to a revised version of the supertype in which branches corresponding to the output labels, that are absent in the subtype, are excluded.

The controllability of the supertype without these edges would have determined whether the subtyping relation applied or not.

Definition 3.3.1 (Attempt 3 - Variant of Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu \mathbf{t}.T'$ then $(T'\{^{T}/_{\mathbf{t}}\}, S) \in \mathcal{R}$;
3. if $T = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j \in J}$, $I \supseteq K$, and $\forall k \in K. (T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;
4. if $T = \oplus\{l_i : T_i\}_{i \in I}$ then
   - $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in K}$,
   - $\forall k \in K.\ I \subseteq J_k$ and $\mathcal{A}[\oplus\{l_i : S_{ki}\}_{i \in I}]^{k \in K}$ is controllable,
   - $\forall i \in I. (T_i, \mathcal{A}[S_{ki}]^{k \in K}) \in \mathcal{R}$.

T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.
T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.

Unfortunately, also this case turned out to be unsafe, as it is possible to observe by considering the same counter-example to the first attempt, shown in Figure 3.3.

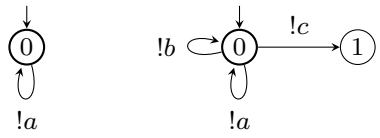
3.4 Attempt 4

After finding out that even the third attempt was unsafe, we briefly considered another definition that seemed to be equivalent to the first one (Definition 3.1.2) and that we discarded quite immediately.

Definition 3.4.1 (Attempt 4 - Variant of Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu \mathbf{t}.T'$ then $(T'\{^{T}/_{\mathbf{t}}\}, S) \in \mathcal{R}$;
3. if $T = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j \in J}$, $I \supseteq K$, and $\forall k \in K.\, (T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;
4. if $T = \oplus\{l_i : T_i\}_{i \in I}$ then
   - $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in K}$,
   - $\mathsf{selUnfold}'(S) = \mathcal{A}[\mu \tilde{\mathbf{t}}_{\mathbf{k}}.\oplus\{l_j : S'_{kj}\}_{j \in J_k}]^{k \in K}$,
   - $\forall k \in K.\, I \subseteq J_k$ and $\mathcal{A}[\oplus\{l_j : S'_{kj}\{^{fail}/_{\tilde{\mathbf{t}}_{\mathbf{k}}}\}_j\}_{j \in J_k \setminus I}]^{k \in K}$ is uncontrollable,
   - $\forall i \in I.\, (T_i, \mathcal{A}[S_{ki}]^{k \in K}) \in \mathcal{R}$.

   where given the sequence of variables $\tilde{\mathbf{t}} = \mathbf{t}_1 \ldots \mathbf{t}_n$, we use $\mu \tilde{\mathbf{t}}.T$ to denote $\mu \mathbf{t_1}. \ldots \mu \mathbf{t_n}.T$, *fail* is any uncontrollable session type (e.g. $\mu \mathbf{t}.\&\{l : \mathbf{t}\}$), and $\{^{fail}/_{\tilde{\mathbf{t}}}\}$ is the substitution of all free occurrences of variables in $\tilde{\mathbf{t}}$ with *fail*.



T S

$$T = \mu \mathbf{t}. \oplus \{a : \mathbf{t}\}$$
$$S = \mu \mathbf{t}. \oplus \{a : \mathbf{t}, b : \mathbf{t}, c : \mathbf{end}\}$$

Figure 3.5: Counter-example to soundness of the fourth attempt of covariance introduction. We would have $T \leq S$, because $(b : \mathit{fail}, c : \mathbf{end})$ is uncontrollable, but T doesn't admit any partner and S is controllable.

T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.
T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.

This definition turned out soon to be unsafe and not equivalent to Definition 3.1.2, because the controllability (Definition 2.4.1) for the internal choice requires all the branches to be controllable. Thus, if one of these branches is uncontrollable, the whole type is considered uncontrollable, allowing therefore the removal of controllable branches. Also in this case, taking into account the previous observation, we have found a counter-example, shown in Figure 3.5. Hence, we continued working on the first attempt until we found out a counter-example to its soundness too.

3.5 Final Attempt

In order to include some form of covariance, taking inspiration from the results in the synchronous case by Padovani [15], the last and more recent attempt requires the finiteness of at least one between the candidate subtype and the candidate supertype, if some of the labels of the supertype are excluded in the subtype. In this way, it is possible to consider in subtyping

relation also cases like those in Figure 3.6, that were excluded by the original proposal of Definition 2.4.4, and are now allowed thanks to the finiteness of the candidate subtype or supertype.

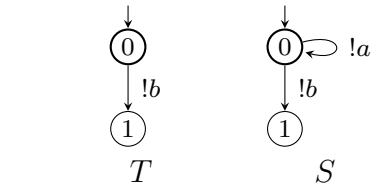
Before displaying the last definition of a variant of fair asynchronous subtyping, we formally define the finiteness of a type, as follows.

Definition 3.5.1 (Finiteness of a type)**.** A type T is finite if no recursion variable $\mathbf{t}$ occurrs in T.

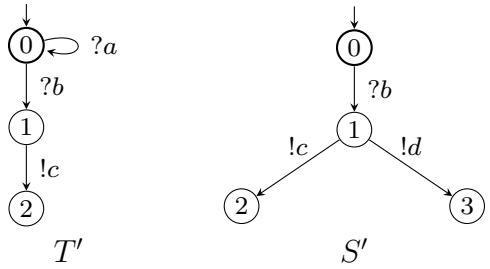
Definition 3.5.2 (Variant of Fair Asynchronous Subtyping, $\leq$)**.** A relation $\mathcal{R}$ on session types is a controllable subtyping relation whenever $(T, S) \in \mathcal{R}$ implies:

1. if $T = \mathbf{end}$ then $\mathsf{unfold}(S) = \mathbf{end}$;
2. if $T = \mu\mathbf{t}.T'$ then $(T'\{^{T}/_{\mathbf{t}}\}, S) \in \mathcal{R}$;
3. if $T = \&\{l_i : T_i\}_{i\in I}$ then $\mathsf{unfold}(S) = \&\{l_j : S_j\}_{j\in J}$, $I \supseteq K$, and $\forall k \in K.\, (T_k, S_k) \in \mathcal{R}$, where $K = \{k \in J \mid S_k \text{ is controllable}\}$;
4. if $T = \oplus\{l_i : T_i\}_{i\in I}$ then
   - $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j\in J_k}]^{k\in K}$,
   - $\forall k \in K.\, (I = J_k)$ or $(I \subset J_k$ and (T is finite or S is finite))
   - $\forall i \in I.\, (T_i, \mathcal{A}[S_{ki}]^{k\in K}) \in \mathcal{R}$.

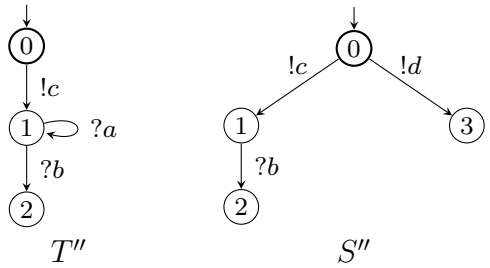
T is a controllable subtype of S if there is a controllable subtyping relation $\mathcal{R}$ s.t. $(T, S) \in \mathcal{R}$.
T is a *fair asynchronous subtype* of S, written $T \leq S$, whenever: S controllable implies that T is a controllable subtype of S.



$$T = \oplus\{b : \mathbf{end}\}$$
$$S = \mu\mathbf{t}.\oplus\{a : \mathbf{t}, b : \mathbf{end}\}$$



$$T' = \mu\mathbf{t}.\&\{a : \mathbf{t}, b : \oplus\{c : \mathbf{end}\}\}$$
$$S' = \mu\mathbf{t}.\&\{b : \oplus\{c : \mathbf{end}, d : \mathbf{end}\}\}$$



$$T'' = \oplus\{c : \mu\mathbf{t}.\&\{a : \mathbf{t}, b : \mathbf{end}\}\}$$
$$S'' = \oplus\{c : \&\{b : \mathbf{end}\}, d : \mathbf{end}\}$$

Figure 3.6: Examples accepted by the new definition of subtyping ($T \leq S$, $T' \leq S'$, $T'' \leq S''$)

3.5.1 The Soundness of the New Definition

Before explaining how the new definition has been introduced in the tool, it is necessary to discuss about its soundness. To prove the soundness of Definition 3.5.2, we tried to follow the proofs that can be found in the Appendix of the original paper [3].

In order to discuss the main lemma for our definition, Lemma 3.5.4, that is strictly connected to Proposition 3.5.5, we have to reformulate some of the previous results[1] as follows.

Lemma 3.5.1. Consider the session type $T = \mathcal{A}[\oplus\{l_j : T_{kj}\}_{j\in J_k}]^{k\in K}$. Let $P_2 = [T, \omega_T]||[S, \omega_S]$ and $P_1^i = [\mathcal{A}[T_{ki}]^{k\in K}, \omega_T]||[S, \omega_S \cdot l_i]$, for every $i \in \bigcap_{k\in K} J_k$. If P_2 is a correct composition then one of the following holds:

- $\mathcal{A}$ does not contain any input branching and $P_2 \to P_1^i$, for every $i \in \bigcap_{k\in K} J_k$;
- $\mathcal{A}$ contains an input branching and P_1^i (for every $i \in \bigcap_{k\in K} J_k$) and P_2 have at least one outgoing transition.
  For every possible transition $P_1^i \to P_1'$ we have that one of the following holds:

  1. P_1^i does not consume the label l_i and there exist $\mathcal{A}'$, $W \subseteq K$, T'_{wj} (for every $w \in W$, $j \in J_w$), S', ω'_T and ω'_S s.t. $P_1' = [\mathcal{A}'[T'_{wi}]^{w\in W}, \omega'_T]||[S', \omega'_S \cdot l_i]$ and
     $P_2 \to [\mathcal{A}'[\oplus\{l_j : T'_{wj}\}_{j\in J_w}]^{w\in W}, \omega'_T]||[S', \omega'_S]$;
  2. P_1^i consumes the label l_i, hence $P_1' = [\mathcal{A}[T_{ki}]^{k\in K}, \omega_T]||[S', \omega_S]$, and $\exists j \in \{1, \ldots, m\}$ s.t. $P_2 \to^* [T_{ji}, \omega'_T]||[S', \omega_S]$ and $\omega_T = a_1 \cdot \ldots \cdot a_w \cdot \omega'_T$, where $a_1, \ldots, a_w$ are the labels in one of the paths to $[]^j$ in $\mathcal{A}$.

  For every possible transition $P_2 \to P_2'$ we have that there exist $\mathcal{A}'$, $W \subseteq K$, T'_{wj} (for every $w \in W$, $j \in J_w$), S', ω'_T and ω'_S s.t.

[1] The proofs of the preliminary results are omitted, because they are analogous to the ones of the original paper [3].

$P_2' = [\mathcal{A}'[\oplus\{l_j : T'_{wj}\}_{j\in J_w}]^{w\in W}, \omega'_T]||[S', \omega'_S]$ and
$P_1^i \to [\mathcal{A}'[T'_{wi}]^{w\in W}, \omega'_T]||[S', \omega'_S \cdot l_i]$.

Lemma 3.5.2. Consider the session type $T = \mathcal{A}[\oplus\{l_j : T_{kj}\}_{j\in J_k}]^{k\in K}$. Let $P_2 = [T, \omega_T]||[S, \omega_S]$ and $P_1^i = [\mathcal{A}[T_{ki}]^{k\in K}, \omega_T]||[S, \omega_S \cdot l_i]$, for every $i \in \bigcap_{k\in K} J_k$. If P_2 is a correct composition then, for every $i \in \bigcap_{k\in K} J_k$, there exists $[T', \omega'_T]||[S', \omega'_S]$ such that $P_1^i \to^* [T', \omega'_T]||[S', \omega'_S]$ and $[T', \omega'_T]||[S', \omega'_S]\surd$.

Proposition 3.5.3. Consider the session type $T = \mathcal{A}[\oplus\{l_j : T_{kj}\}_{j\in J_k}]^{k\in K}$. We have that if $[T, \omega_T]||[S, \omega_S]$ is a correct composition then, for every $i \in \bigcap_{k\in K} J_k$, we have that also $[\mathcal{A}[T_{ki}]^{k\in K}, \omega_T]||[S, \omega_S \cdot l_i]$ is a correct composition.

The demonstration of the soundness of the new attempt, requires to reason about the finiteness of a type, so the concept of depth of a type has been introduced.

Definition 3.5.3 (Depth of a type)**.** Given a type T

$$\mathsf{depth}(T) = \begin{cases} 1 & \text{if } T = \mathbf{t} \text{ or } T = \mathbf{end} \\ 1 + \mathsf{depth}(T) & \text{if } T = \mu\mathbf{t}.\mathbf{t} \\ 1 + \max_{i\in I}\{\mathsf{depth}(T_i)\} & \text{if } T = \oplus\{l_i : T_i\}_{i\in I} \text{ or } T = \&\{l_i : T_i\}_{i\in I} \end{cases}$$

At this point, in order to demonstrate Proposition 3.5.5, we introduced Lemma 3.5.4 to deal with the soundness if one of the considered types is finite.

Lemma 3.5.4. Given two session types T and S, if $T \leq S$ and one between T and S is finite then, for every ω, R, and ω_R such that $[S, \omega]||[R, \omega_R]$ is a correct composition, there exist T', ω', R', and ω'_R such that $[T, \omega]||[R, \omega_R] \to^* [T', \omega']||[R', \omega'_R]$ and $[T', \omega']||[R', \omega'_R]\surd$.

Proof. We proceed by induction on the depth of the finite type.

Base case. If one of the two types is finite, it has depth 1, then it is **end**. Given that $T \leq S$, also the other type is **end** (possibly by applying unfold).

$[S, \omega]||[R, \omega_R]$ correct and $\mathsf{unfold}(S) = \mathbf{end}$ imply that ω is empty and $[S, \omega]||[R, \omega_R] \rightarrow^* [S, \omega]||[R', \omega'_R]\surd$. Hence, also $[T, \omega]||[R, \omega_R] \rightarrow^* [T, \omega]||[R', \omega'_R]\surd$, because also $\mathsf{unfold}(T) = \mathbf{end}$.

Inductive case. We consider three cases:

1. $[S, \omega]||[R, \omega_R]\surd$. We proceed as in the above case.

2. $\mathsf{unfold}(S) = \oplus\{l_i : S_i\}_{i \in I}$. As $T \leq S$, we have $\mathsf{unfold}(T) = \oplus\{l_j : S_j\}_{j \in J}$ and $J \subseteq I$.

   We take $i \in J$. $[S_i, \omega]||[R, \omega_R \cdot l_i]$ is correct. We consider $[T, \omega]||[R, \omega_R] \rightarrow [T_i, \omega]||[R, \omega_R \cdot l_i]$. We can apply the inductive hypothesis because $T_i \leq S_i$ and the depth of finite type strictly decreases, i.e. if T is finite, then $\mathsf{depth}(T_i) < \mathsf{depth}(T)$, if S is finite, then $\mathsf{depth}(S_i) < \mathsf{depth}(S)$.

3. $\mathsf{unfold}(S) = \&\{l_i : S_i\}_{i \in I}$. We have two sub-cases:

   - $\mathsf{unfold}(T) = \&\{l_j : T_j\}_{j \in J}$. We consider $[S, l_i \cdot \omega]||[R, \omega_R] \rightarrow [S_i, \omega]||[R, \omega_R]$. Also $[T, l_i \cdot \omega]||[R, \omega_R] \rightarrow [T_i, \omega]||[R, \omega_R]$.

     We can apply the inductive hypothesis, because $[S_i, \omega]||[R, \omega_R]$ is correct, $T_i \leq S_i$ and the depth of the finite type strictly decreases.

   - $\mathsf{unfold}(T) = \oplus\{l_j : T_j\}_{j \in J}$. Given that $T \leq S$, $\mathsf{selUnfold}(S) = \mathcal{A}[\oplus\{l_i : S_{ik}\}_{i \in I_k}]^{k \in K}$ with $J \subseteq I_k$ for every $k \in K$. Consider now $i \in J$. We have that $i \in \bigcap_{k \in K} I_k$.

     By Proposition 3.5.3, we have that $[\mathcal{A}[S_{ik}]^{k \in K}, \omega]||[R, \omega_R \cdot l_i]$ is correct.

     Consider now $[T, \omega]||[R, \omega_R] \rightarrow [T_i, \omega]||[R, \omega_R \cdot l_i]$. Given that $T \leq S$, we also have $T_i \leq \mathcal{A}[S_{ik}]^{k \in K}$. We can apply the inductive hypothesis because the depth of the finite type strictly decreases.

□

Proposition 3.5.5. Given two session types T and S, if $T \leq S$ then, for every ω, R, and ω_R such that $[S, \omega]||[R, \omega_R]$ is a correct composition, there

exist T', ω', R', and ω'_R such that $[T, \omega]|[R, \omega_R] \rightarrow^* [T', \omega']|[R', \omega'_R]$ and $[T', \omega']|[R', \omega'_R]\surd$.

Proof. We have to consider whether one between S and T is finite.

- If one between S and T is finite, the thesis follows from Lemma 3.5.4.
- If both of them are not finite, Definition 3.5.2 does not allow any sort of covariance, so the proof proceeds like in the original paper [3].

□

Note that the new variant of fair asynchronous subtyping is sound with respect to fair refinement but it is not complete, as it is shown by the example below.

Let $T = \mu\mathbf{t}.\oplus\{l_1 : \mathbf{end}, l_2 : \mathbf{t}\}$ and $S = \mu\mathbf{t}.\oplus\{l_1 : \mathbf{end}, l_2 : \mathbf{t}, l_3 : \mathbf{t}\}$. T is a refinement of S but T is not a subtype of S since neither of them is finite.

Chapter 4

Implementation

To get a checker of the subtyping relation on the basis of Definition 3.5.2, the original implementation by Bravetti, Lange, and Zavattaro has been modified in order to allow some covariance. Since fair subtyping is undecidable, it is impossible to have a sound and complete algorithm, therefore a sound algorithm, that can return also *unknown* as result, has been realised.

In this chapter the main features of the original algorithm and its implementation are shown along with the changes in the code which allow the introduction of a slight form of covariance.

4.1 On the Subtyping Algorithm

The algorithm is based on the construction of a simulation tree according to the simulation game corresponding to the subtyping Definition 2.4.4, then adapted to Definition 3.5.2.

The simulation tree is the labelled tree representing the simulation game, represented by $simtree(T, S)$, i.e. a tuple $(N, n_0, \twoheadrightarrow, \lambda)$, where N corresponds to the set of nodes, $n_0 \in N$ is the root, $\twoheadrightarrow$ is the transition function, corresponding to the definition of fair asynchronous subtyping, and λ is the labelling function. The label of the root is $\lambda(n_0) = (S, T)$.

If S is not controllable, there is no need to run the simulation game since

the subtyping relation holds, otherwise, the definition of the simulation tree is required to check whether $T \leq S$.

If a branch of the simulation tree is infinite or it finishes in an $(\mathbf{end}, \mathbf{end})$ leaf, it is successful, otherwise it is recognized as unsuccessful.

Assuming that S is controllable, $T \leq S$ iff all branches of $simtree(T, S)$ are successful. The problem of checking the success of all the branches is usually undecidable, because of the possible generation of infinitely many pairs.

To solve the problem in presence of unbounded accumulation, Bravetti, Lange, and Zavattaro introduced the notion of witness subtrees, which are finite subtrees of a simulation tree that are successful, because they satisfy a certain accumulation pattern.

They are based on the idea of ancestor of a node n, that is a node n' such that $n \neq n'$ and n' is on the path from n_0 to n.

The input contexts tracked down by witness trees are the ones with

1. growing holes leading to an infinite growth, or
2. constant holes stable during the simulation game.

An input context is defined *extended* when it contains holes with the same index.

To have an idea about extended input contexts with the same index and their reductions, consider the following example. Let

$$\begin{aligned} \mathcal{A}_1 &= \mu\mathbf{t}.\&\{a : []^1, b : \&\{c : \mathbf{t}\}\} \\ \mathsf{unfold}(\mathcal{A}_1) &= \&\{a : []^1, b : \&\{c : \mu\mathbf{t}.\&\{a : []^1, b : \&\{c : \mathbf{t}\}\}\}\} \\ \mathcal{A}_2 &= \&\{c : \mu\mathbf{t}.\&\{a : []^1, b : \&\{c : \mathbf{t}\}\}\} \end{aligned}$$

Both $\mathsf{unfold}(\mathcal{A}_1)$ and $\mathcal{A}_2$ are reductions of $\mathcal{A}$. $\mathsf{unfold}(\mathcal{A}_1)$ falls within the previous mentioned cases in which two distinct holes have the same index (1 in this example). $\mathcal{A}_2$ is reachable from the unfolding of $\mathcal{A}_1$ by inputting b.

The set of *reduction* $\mathcal{S}$ of an input context $\mathcal{A}$ is the minimal set $\mathcal{S}$ such that:

1. $\mathcal{A} \in \mathcal{S}$
2. if $\&\{l_i : \mathcal{A}_i\}_{i \in I} \in \mathcal{S}$ then $\forall i \in I . \mathcal{A}_i \in \mathcal{S}$
3. if $\mu \mathbf{t} . \mathcal{A}' \in \mathcal{S}$ then $\mathcal{A}'\{^{\mu \mathbf{t} . \mathcal{A}'}/_{\mathbf{t}}\} \in \mathcal{S}$

In aftermath of the unfolding, reductions of an input context may contain extended input contexts. If $\mathcal{A}'$ is a reduction of $\mathcal{A}$, $holes(\mathcal{A}') \subseteq holes(\mathcal{A})$.

Let $\mathcal{A}$ be an extended context and $K \subseteq holes(\mathcal{A})$ a set of hole indices. In the formal definition of witness tree provided by Bravetti, Lange, and Zavattaro, the following abbreviations are used:

- $\mathcal{A}\lfloor T_k \rfloor^{k \in K}$ corresponds to the extended context obtained by replacing each hole $k \in K$ in $\mathcal{A}$ by the type T_k for each $k \in K$,
- $\mathcal{A}\langle \mathcal{A}' \rangle^K$ corresponds to the extended context obtained by replacing each hole $k \in K$ in $\mathcal{A}$ by the extended context $\mathcal{A}'$.

If $K = \{k\}$, the notation will be $\mathcal{A}\lfloor T_k \rfloor^k$ and $\mathcal{A}\langle \mathcal{A}' \rangle^k$ respectively.

Definition 4.1.1 (Witness Tree)**.** A tree $(N, n_0, \twoheadrightarrow, \lambda)$ is a *witness tree* for $\mathcal{A}$, such that $holes(\mathcal{A}) = I$, with $\emptyset \subseteq K \subset I$ and $J = I \setminus K$, if all the following conditions are satisfied:

1. for all $n \in N$ either $\lambda(n) = (T, \mathcal{A}'\langle \mathcal{A}\lfloor S_j \rfloor^{j \in J} \rangle^J \lfloor S_k \rfloor^{k \in K})$ or $\lambda(n) = (T, \mathcal{A}'\langle \mathcal{A}\langle \mathcal{A}\lfloor S_j \rfloor^{j \in J} \rangle^J \rangle^J \lfloor S_k \rfloor^{k \in K})$, where $\mathcal{A}'$ is a reduction of $\mathcal{A}$, and it holds that
   - $holes(\mathcal{A}') \subseteq K$ implies that n is a leaf and
   - if $\lambda(n) = (T, \mathcal{A}[S_i]^{i \in I})$ and n is not a leaf then $\mathsf{unfold}(T)$ starts with an output selection;
2. each leaf n of the tree satisfies one of the following conditions:
   (a) $\lambda(n) = (T, S)$ and n has an ancestor n' s.t. $\lambda(n') = (T, S)$

(b) $\lambda(n) = (T, \mathcal{A}\langle\mathcal{A}\lfloor S_j\rfloor^{j\in J}\rangle^J\lfloor S_k\rfloor^{k\in K})$ and n has an ancestor n' s.t. $\lambda(n') = (T, \mathcal{A}[S_i]^{i\in I})$

(c) $\lambda(n) = (T, \mathcal{A}[S_i]^{i\in I})$ and
n has an ancestor n' s.t. $\lambda(n') = (T, \mathcal{A}\langle\mathcal{A}\lfloor S_j\rfloor^{j\in J}\rangle^J\lfloor S_k\rfloor^{k\in K})$

(d) $\lambda(n) = (T, \mathcal{A}'[S_k]^{k\in K'})$ where $K' \subseteq K$

and for all leaves (T, S) of type (2c) or (2d) $T \leq S$ holds.

Condition (1) refers to witness subtree nodes, that are labelled by pairs (T, S) where S contains a fixed context $\mathcal{A}$ whose holes are partitioned into growing (J-indexed) and constant holes (K-indexed). When all growing holes are removed by context reduction, the pair is labelling a leaf of the subtree. If the initial input is limited to only one instance of $\mathcal{A}$, T begins with an output choice and this input cannot be consumed in the subtyping simulation game.

Condition (2) refers to constraints that all leaves need to respect in order to ensure the correctness of the branches.

Condition (2a) applies on leaves that have ancestors having the same label, so the success of these branches and the corresponding simulation game is trivially ensured.

Condition (2b) is satisfied by leaves with a regular "increase" of the growing (J-indexed) holes in compliance with the same accumulation pattern from their ancestors.

Condition (2c) is satisfied by leaves with a regular "decrease" of the types in the growing holes in compliance with the same reduction pattern from their ancestors.

Condition (2d) is satisfied by leaves using only constant (K-indexed) holes, because context reduction leads to the removal of growing holes containing the accumulation $\mathcal{A}$.

Algorithm. The first step of the algorithm is the controllability check on S. If S is uncontrollable, it is possible to declare that $T \leq S$, otherwise the following steps need to be performed.

S1 Compute a finite fragment of $simtree(T, S)$ stopping if

- a leaf (successful or not) is encountered,
- a node respecting the Condition about the ancestor (2a, 2b, 2c) of Definition 4.1.1 is encountered,
- the length of the path between the root and the current node is bigger than a bound corresponding to twice the depth of the abstract syntax tree of S.

S2 Remove successful branches with finitely many labels from the tree computed in S1, i.e. the subtrees whose each leaf is successful or has an ancestor in the same subtree with the same label.

S3 Forest of subtrees rooted in the ancestor nodes which do not have ancestors themselves are extracted from the tree computed in **S2** in order to be checked.

S4 Check whether each candidate from **S3** is a witness tree or not.

The result of the algorithm can be

- *False*, if considered session types are not related, i.e. an unsuccessful leaf is found in **S1**,
- *True*, if considered session types are related, i.e. all checks in **S4** succeed,
- *Unknown* in all the other cases in which the algorithm is unable to return an answer, i.e. either when in **S1** the generation of the subtree reached the bound before reaching a successful state (leaf or node with an ancestor) or the candidate subtree in **S4** is not a witness.

Theorem 4.1.1. Let T and S be two session types with $simtree(T, S) = (N, n_0, \twoheadrightarrow, \lambda)$. If $simtree(T, S)$ contains a witness subtree with root n then for every node $n' \in N$ s.t. $n \twoheadrightarrow^* n'$, either n' is a successful leaf, or there exists n'' s.t. $n' \twoheadrightarrow n''$.

Therefore, if the candidate subtrees of $simtree(T, S)$ are also witness trees, it is possible to assert $T \leq S$ [1] [3].

4.2 On the Implementation of the Tool

The implementation of the tool [8] for verifying the new variant of fair subtyping relation of Definition 3.5.2 derives from changes to the original Haskell implementation [4] by Bravetti, Lange, and Zavattaro.

It takes two types T and S as input, and it tries to determine whether $T \leq S$. In addition to candidate subtype and supertype, the user can provide an additional value corresponding to the bound.

The tool works with the automata representation of the types. Each local state in supertype automaton has two counters:

- the c-counter, for the number of occurrences of a state in an input context
- the h-counter, for the number of occurrences of a state within a hole of an input context

Hence, state labels include the original value of the state and both the additional counters, and are used to identify the context $\mathcal{A}$ to use in the check of witness trees.

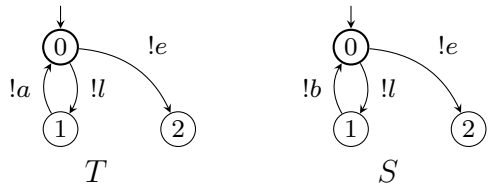
4.2.1 On oneStep Function

The part of the tool that required a special attention, in order to adapt the original version of the tool to the new one supporting the new definition

[1] The proof of this result, that has been completely demonstrated for the original definition of fair asynchronous subtyping (see Definition 2.4.4), can be adapted to the simulation tree given by the new definition (Definition 3.5.2) through trivial changes. These adjustments have to be applied only to the case in which T starts with an output selection of Lemma 7 (see Appendix of the original paper [3]). It can be easily proven that, if there is covariance and T or the r.h.s. are finite, the context with an extra level compared with context $\mathcal{A}$ can have the same simulation step in which covariance is used.

of subtyping, is oneStep function.

```
oneStep :: Bool -> Machine -> Value -> Maybe [(Label, Value)]
oneStep debug m1 v@(p,m)
  | isFinalConf m1 v = (if debug then (trace ("Final: "++(show (p,(
      tinit m)))++"\n"++(printMachine m)) ) else (\x  -> x  )) $
    Just []
    --
  | not $ isControllable m = Just []
    --
  | (isInput m1 p) && (isInput m (tinit m)) && ((inControllableBarb m
    (tinit m)) `isSubsetOf` (inBarb m1 p)) =
    (if debug then (trace ("In: "++(show (p,(tinit m)))) ) else (\x
         -> x  )) $
    let  psmoves = L.map snd $ L.filter (\(x,(y,z)) -> x==p) $
        transitions m1
         qsmoves = L.map snd $ L.filter (\(x,(y,z)) -> x==(tinit m))
              $ transitions m
         next = L.nub
                $ [(a,(x, cleanUp $ updateInit y m)) |
                   (a,x) <- psmoves,
                   (b,y) <- qsmoves,
                   c <- S.toList (inControllableBarb m (tinit m)),
                   b==(Receive, c),
                   a==b]
    in Just next
       --
- | (isOutput m1 p) && (isOutput m (tinit m)) && ((outBarb m1 p) == (
    outBarb m (tinit m))) =
+ | (isOutput m1 p) && (isOutput m (tinit m)) && (outputCovariance m1
    p m) =
      (if debug then (trace ("OutSync: "++(show (p,(tinit m)))++"\n
          "++(printMachine m)) ) else (\x  -> x  )) $
      let psmoves = L.map snd $ L.filter (\(x,(y,z)) -> x==p) $
          transitions m1
          qsmoves = L.map snd $ L.filter (\(x,(y,z)) -> x==(tinit m)
              ) $ transitions m
          next = L.nub $ [(b,(x,cleanUp $ updateInit y m)) | (a,x)
              <- psmoves, (b,y) <- qsmoves, a==b]
      in Just next
  | (isOutput m1 p) && not (isOutput m (tinit m)) =
        (if debug then (trace ("Out: "++(show (p,(tinit m)))++"\n
            "++(printMachine m)) ) else (\x  -> x  )) $
        let psmoves = L.map snd $ L.filter (\(x,(y,z)) -> x==p) $
            transitions m1
            qstates = reachableSendStates (tinit m) m
            newmachines = L.map (\a -> ((Send, a), replaceInMachine
                m (Send, a) qstates)) $ S.toList (outBarb m1 p)
```



$$T = \mu \mathbf{t}. \oplus \{l : \oplus\{a : \mathbf{t}\}, e : \mathbf{end}\}$$
$$S = \mu \mathbf{t}. \oplus \{l : \oplus\{b : \mathbf{t}\}, e : \mathbf{end}\}$$

Figure 4.1: Example that was considered erroneously True by the previous version of the tool.

```
33                next = L.nub $ [(a, (x, updateInit (ssucc (tinit m)) m')
                      )| (a,x) <- psmoves, (b,m') <- newmachines, a==b]
34            in if (not $ L.null qstates)
35                  &&
36
37  -               (L.and $ L.map (\x -> (outBarb m1 p) == (outBarb m x))
        qstates)
38  +               (outputCovariance m1 p m)
39                then Just next
40                else (if debug then (trace ("BadOut: "++(show (p,(tinit m
                      )))++"\n"++(printMachine m)) ) else (\x  -> x  )) $
41                    Nothing
42    | otherwise = (if debug then (trace ("Bad: "++(show (p,(tinit m)))
          ++"\n"++(printMachine m)) ) else (\x  -> x  )) $
43                  Nothing
```

As it is possible to see from the added lines, oneStep has been modified to include the support of covariance, as explained in Subsection 4.2.3.

4.2.2 The Controllability Check

The first check performed by the algorithm is the controllability one. While investigating on the way the tool works, we discovered a slight problem that needed to be fixed, in order to ensure the correctness of the algorithm. The problem was found while reasoning on the example shown in Figure 4.1. The previous version of the checker stated $T \leq S$ erroneously. Following the code execution, we found out that this problem was given by the check

```
6       |isControllable m = Just[]
```

inside the oneStep function. The problem is that the type is considered uncontrollable, so the check in the oneStep function succeeds, but the type is controllable actually. The issue of the previous version was the termination condition of the isControllable function, that was previously included in the isOutput case, as follows.

When starting from state 0, the automata in Figure 4.1 are correctly considered as controllable, but, when the starting point is moved from state 0 to 1, it is considered uncontrollable, because the set of unseen states is empty but state 2 is still reachable so the arrival to a termination state is still possible.

Trying to fix this problem and get closer to the definition of controllability, i.e. Definition 2.4.1, isControllable has been modified. The new version of isControllable creates a list of automata, one for each external choice, and then checks whether T ok, as follows.

```
1  isControllable :: Machine -> Bool
2  isControllable ma = any (\x -> helper x [] (tinit x)) (
       singleExtChoices ma)
3    where helper m seen q
4            | q `L.elem` seen = endReachable m q
5            | isFinal m q = True
6            | isInput m q = case successors m q of
7                              [(l,t)] -> helper m (q:seen) t
8                              ys -> error (show (q, ys))
9            | isOutput m q = all (\x -> helper m (q:seen) (snd x)) (
                 successors m q)
10
11
12 endReachable :: Machine -> State -> Bool
13 endReachable m q = helper [] q
14   where helper seen q
15           | q `L.elem` seen = False
16           | isFinal m q = True
17           | otherwise = any (\x -> helper (q:seen) (snd x)) (
                 successors m q)
18
19 singleExtChoices :: Machine -> [Machine]
20 singleExtChoices m = mlist
21   where mlist = L.map (\x -> Machine { states = states m
22                                      , tinit = tinit m
23                                      , transitions = x++sndtrans
```

```
24                                      , accepts = accepts m
25                                      }
26                      ) combo
27          sndtrans = ftrans Send
28          ftrans dir = L.filter (\(s,((d,l),t)) -> d == dir) (
                transitions m)
29          combo = sequence $ L.groupBy (\x y -> (fst x) == (fst y)) (
                sortBy (comparing fst) (ftrans Receive))
```

Thanks to this fix, cases like the one in Figure 4.1 are now considered correctly controllable and the subtyping relation between T and S does not hold anymore, because the right case of the oneStep function is considered.

4.2.3 Covariance Introduction

To introduce a slight form of covariance, according to Definition 3.5.2, some changes have been done inside oneStep function, in particular in two cases:

- the one where both the candidate subtype and supertype are in an output state,
- the one where the candidate subtype is in an output state and the candidate subtype is in an input state, so there is an output anticipation in the subtype and an input context in the supertype.

As shown in Subsection 4.2.1, in both cases a call to a new function, outputCovariance, is required.

To realize the covariance check, an additional function for the finiteness of the type has been implemented, as follows.

```
1  isFinite :: Machine -> State -> Bool
2  isFinite m t = helper m t []
3    where helper m s seen
4            | L.null (successors m s) = True
5            | not $ L.null $ L.filter (\x -> (snd x) `L.elem` seen) $
                 successors m s = False
6            | otherwise = L.and $ L.map (\x -> helper m x $ seen++(L.map
                 (snd) $ successors m s)) $ L.map(snd) $ successors m s
```

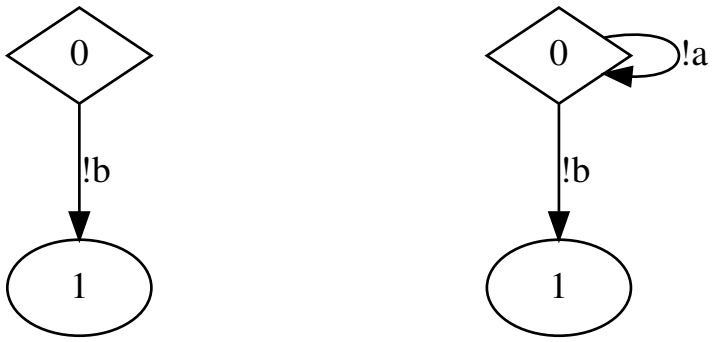
The outputCovariance check is correct when either

- the set of output labels is the same for both the candidate subtype and the candidate supertype, or
- the set of output labels of the candidate subtype is included in the one of the candidate supertype and at least one of the two types is finite.

```
1 outputCovariance :: Machine -> State -> Machine -> Bool
2 outputCovariance m1 p m = (L.and $ L.map (\x -> (outBarb m1 p) == (
      outBarb m x)) sendStates)
3                           ||
4                           ((L.and $ L.map (\x -> (outBarb m1 p) `
                                isSubsetOf` (outBarb m x)) sendStates)
5                             &&
6                             (isFinite m1 p || isFinite m (tinit m)))
7   where sendStates = reachableSendStates (tinit m) m
```

4.2.4 Example of Tool Outputs

In the following, we present some automata produced by the tool in debug mode, by inputting the examples previously presented in Figure 3.6 and accepted by the new version of the tool. After these examples, another one supported by both versions of the tool is shown, to illustrate a case where the simulation tree cannot be completely pruned and there is a witness subtree to check.



(a) Subtype (b) Supertype

Figure 4.2: Input session types as CFSMs

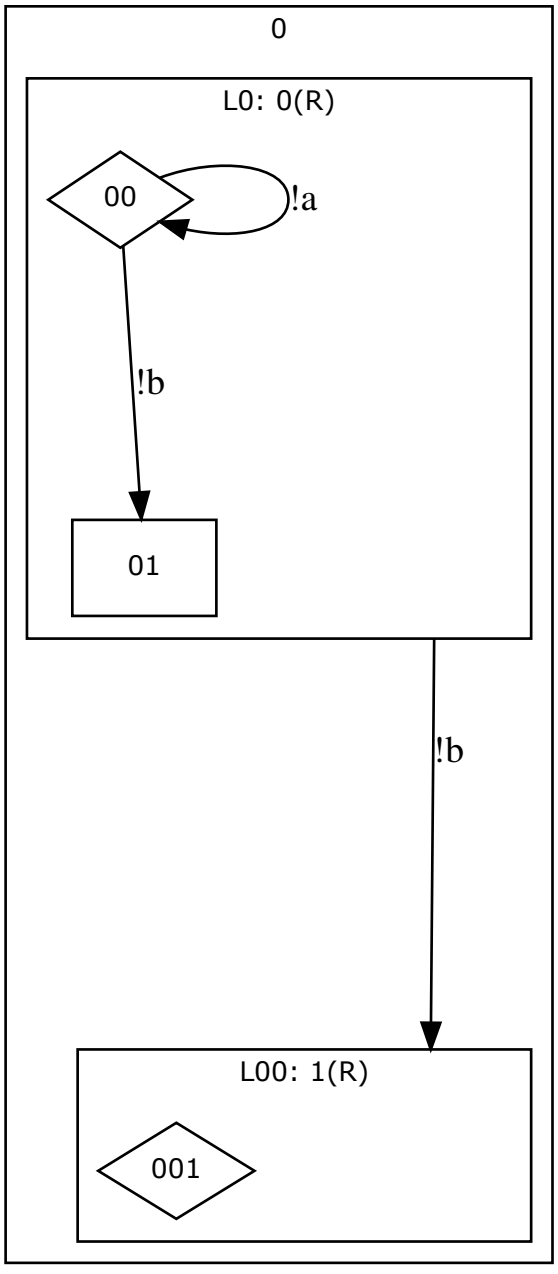
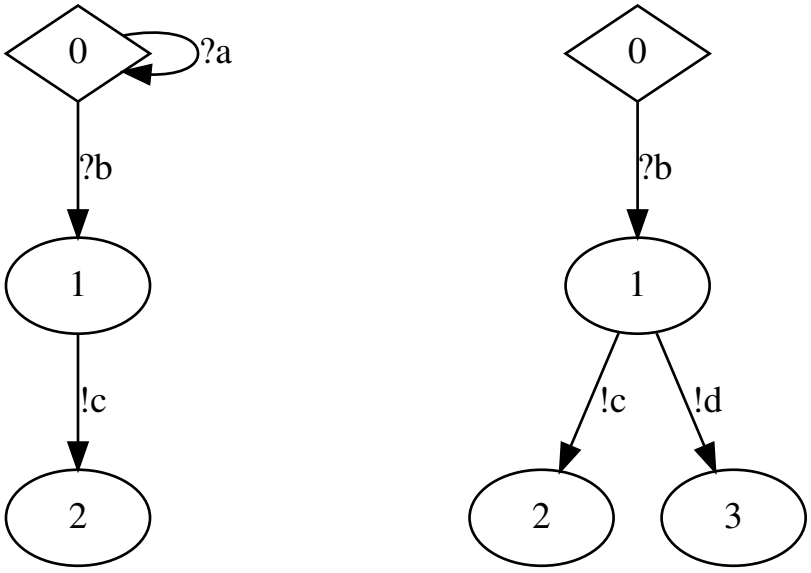


Figure 4.3: Simulation tree for Figure 4.2



(a) Subtype (b) Supertype

Figure 4.4: Input session types as CFSMs

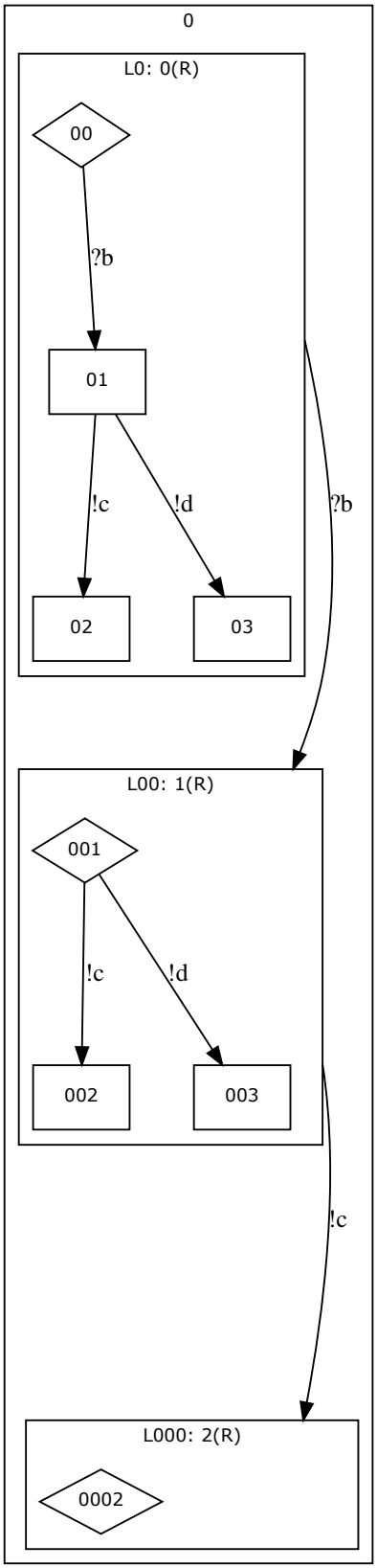
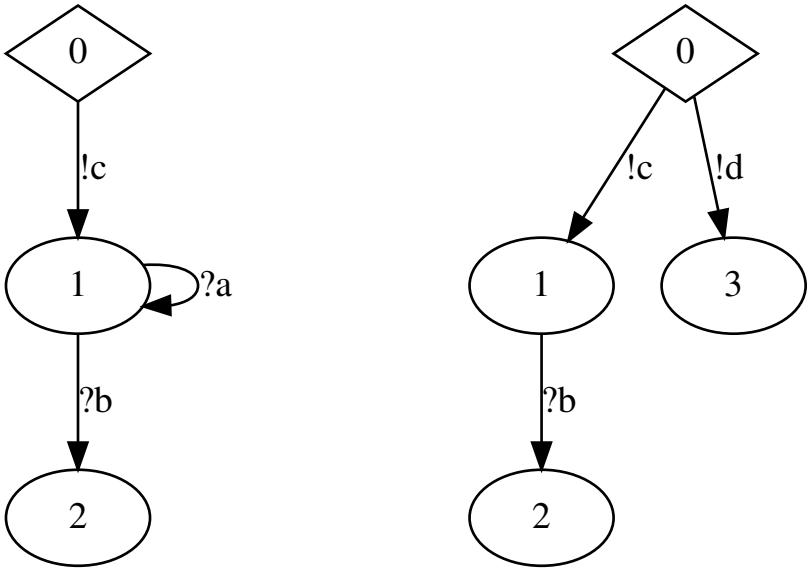


Figure 4.5: Simulation tree for Figure 4.4



(a) Subtype (b) Supertype

Figure 4.6: Input session types as CFSMs

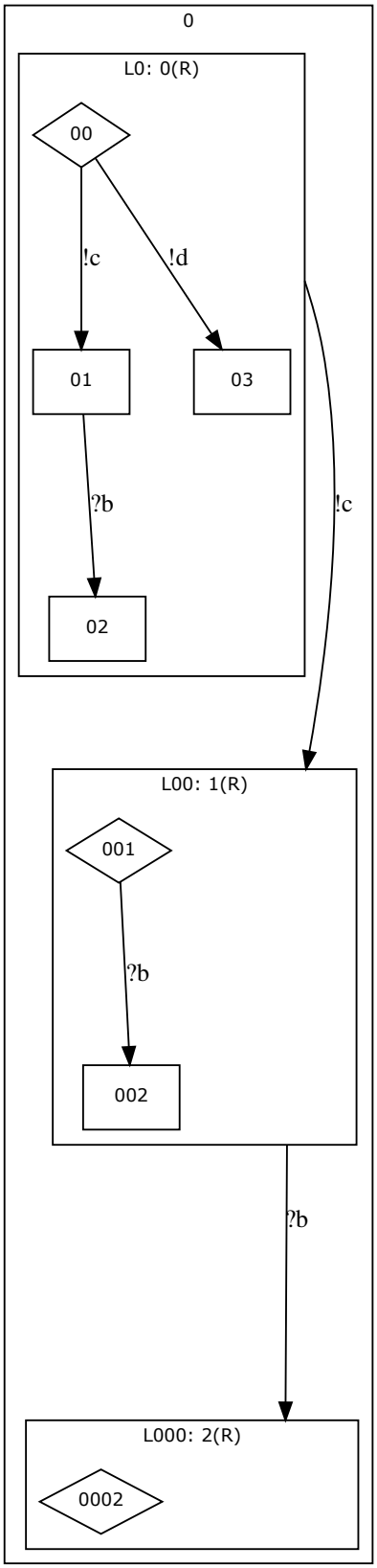
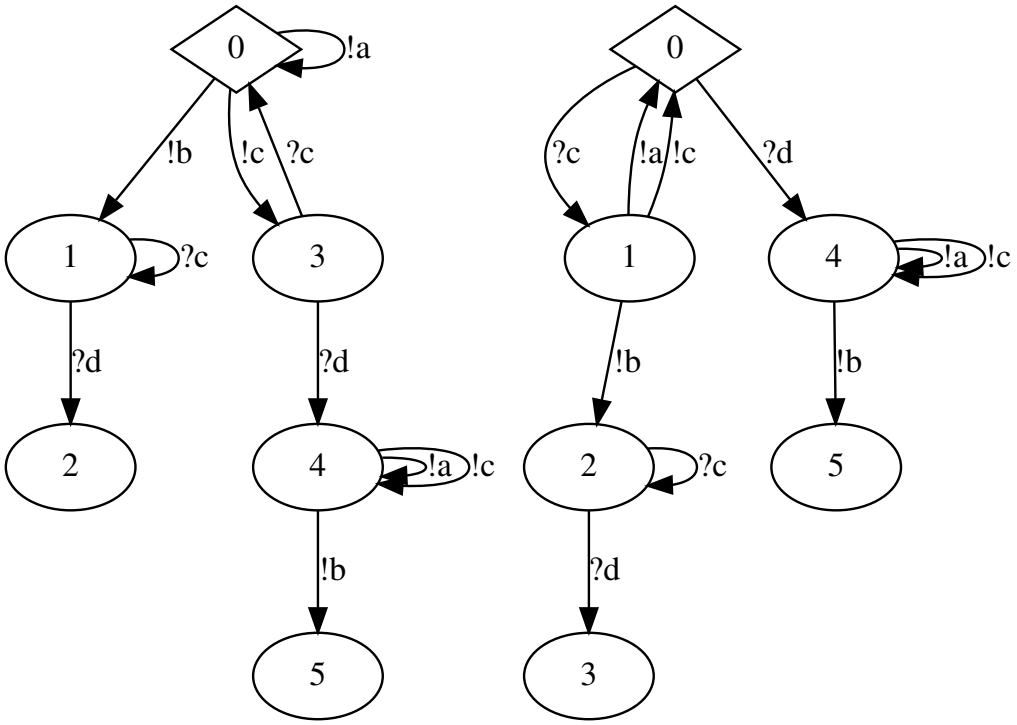


Figure 4.7: Simulation tree for Figure 4.6



(a) Subtype (b) Supertype

Figure 4.8: Input session types as CFSMs

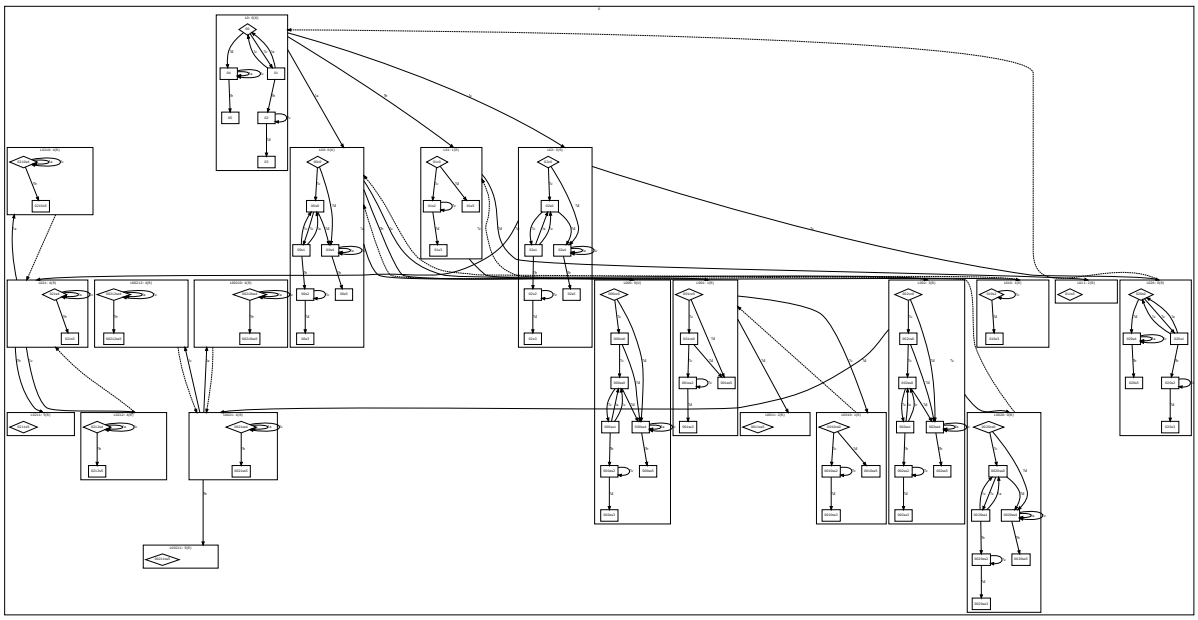


Figure 4.9: Simulation tree for Figure 4.8

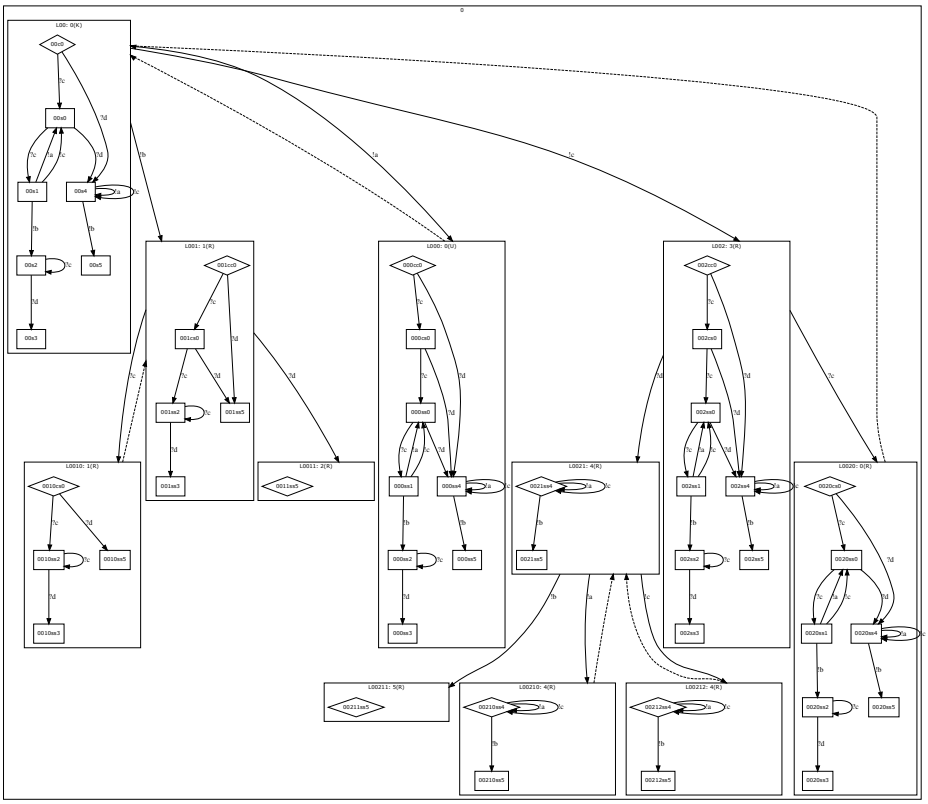


Figure 4.10: Witness tree for Figure 4.8, subtree of Figure 4.9

Conclusions and Future Works

In concurrent and distributed systems, reasoning on ways to prevent communication problems, especially at compile time, is fundamental. Session types are one of the most promising mechanisms to avoid issues like deadlocks and orphan messages.

This master thesis represents an attempt of introduction of covariance in the fair asynchronous refinement by Bravetti, Lange, and Zavattaro [3].

After all the attempts shown in Chapter 3, we have proposed a new definition of subtyping, in Definition 3.5.2, in the awareness that the constraints that we require are quite strict, differently from the ones of the first alternative definition that we considered, Definition 3.1.2, that turned out to be unsound.

We have integrated the new definition in the pre-existing tool [4], that before this thesis did not support any form of output covariance. Keeping the tool updated with the new definitions allows us to have a concrete way to verify the subtyping relation and to discuss about the simulation game on concrete cases, also thanks to its graphical outputs.

The new definition (Definition 3.5.2) requires one between the candidate subtype and the candidate supertype to be finite. This constraint allowed us to introduce covariance and to demonstrate the soundness of the new definition, but it reduced the set of possible cases on which the definition can be applied to. In the future, the aim is to look for new sound definitions, in order to allow a more relaxed form of covariance and to get closer to concrete needs of real systems.

Bibliography

[1] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. *On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping*. 2017. arXiv: 1703.00659 [cs.PL].

[2] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. *Undecidability of Asynchronous Session Subtyping*. 2016. arXiv: 1611.05026 [cs.PL].

[3] Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. *Fair Refinement for Asynchronous Session Types*. 2021. arXiv: 2101.08181 [cs.PL].

[4] Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. *Fair Refinement for Asynchronous Session Types*. https://github.com/julien-lange/fair-asynchronous-subtyping.

[5] Mario Bravetti et al. *A Sound Algorithm for Asynchronous Session Subtyping and its Implementation*. 2019. arXiv: 1907.00421 [cs.PL].

[6] Tzu-Chun Chen et al. "On the Preciseness of Subtyping in Session Types". In: *CoRR* abs/1610.00328 (2016). arXiv: 1610.00328. URL: http://arxiv.org/abs/1610.00328.

[7] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. "Sessions and Session Types: An Overview". In: *Web Services and Formal Methods*. Ed. by Cosimo Laneve and Jianwen Su. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–28. ISBN: 978-3-642-14458-5.

[8] *Fair Refinement for Asynchronous Session Types. New subtyping*. https://github.com/signax/fair-asynchronous-subtyping.

[9] Simon J. Gay and Malcolm Hole. "Subtyping for session types in the pi calculus". In: *Acta Informatica* 42 (2005), pp. 191–225.

[10] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-69722-0.

[11] Hans Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (Apr. 2016). ISSN: 0360-0300. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052.

[12] Julien Lange and Nobuko Yoshida. "On the Undecidability of Asynchronous Session Subtyping". In: *Foundations of Software Science and Computation Structures*. Ed. by Javier Esparza and Andrzej S. Murawski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 441–457. ISBN: 978-3-662-54458-7.

[13] Dimitris Mostrous and Nobuko Yoshida. "Session typing and asynchronous subtyping for the higher-order π-calculus". In: *Information and Computation* 241 (2015), pp. 227–263. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2015.02.002. URL: http://www.sciencedirect.com/science/article/pii/S0890540115000139.

[14] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. "Global Principal Typing in Partially Commutative Asynchronous Sessions". In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 316–332. ISBN: 978-3-642-00590-9.

[15] Luca Padovani. "Fair Subtyping for Multi-party Session Types". In: *Coordination Models and Languages*. Ed. by Wolfgang De Meuter and Gruia-Catalin Roman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 127–141. ISBN: 978-3-642-21464-6.

[16] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. “An Interaction-based Language and its Typing System”. In: *In PARLE’94, volume 817 of LNCS*. Springer-Verlag, 1994, pp. 398–413.