On the Preciseness of Subtyping in Session Types

This is the author's manuscript

Original Citation:
((Intervento presentato al convegno PPDP'14 tenutosi a Canterbury nel 2014.

Availability:
This version is available http://hdl.handle.net/2318/149876 since

Publisher:
ACM Press

Terms of use:
Open Access
Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)
This is an author version of the contribution published on:

T. Chen, M. Dezani, N. Yoshida
On the Preciseness of Subtyping in Session Types
Editor: ACM Press
2014

in

PPDP'14
135 - 146
PPDP'14
Canterbury
2014

The definitive version is available at:
On the Preciseness of Subtyping in Session Types

Tzu-Chun Chen  
Università di Torino  
chen@di.unito.it

Mariangiola Dezani-Ciancaglini  
Università di Torino  
dezani@di.unito.it

Nobuko Yoshida  
Imperial College London  
yoshida@doc.ic.ac.uk

Abstract
Subtyping in concurrency has been extensively studied since early 1990s as one of the most interesting issues in type theory. The correctness of subtyping relations has been usually provided as the soundness for type safety. The converse direction, the completeness, has been largely ignored in spite of its usefulness to define the greatest subtyping relation ensuring type safety. This paper formalises preciseness (i.e. both soundness and completeness) of subtyping for mobile processes and studies it for the synchronous and the asynchronous session calculi. We first prove that the well-known session subtyping, the branching-selection subtyping, is sound and complete for the synchronous calculus. Next we show that in the asynchronous calculus, this subtyping is incomplete for type-safety: that is, there exist session types \( T \) and \( S \) such that \( T \) can safely be considered as a subtype of \( S \), but \( T \not\leq S \) is not derivable by the subtyping. We then propose an asynchronous subtyping system which is sound and complete for the asynchronous calculus. The method gives a general guidance to design rigorous channel-based subtypings respecting desired safety properties.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

Keywords Session types, \( \pi \)-calculus, Subtyping, Completeness and Soundness.

1. Introduction
Subtyping in concurrency Since Milner first introduced the idea of assigning types to channels in the \( \pi \)-calculus [23], the subtypings which define an ordering over usages of channels have been recognised as one of the most useful concepts in the studies of the \( \pi \)-calculus.

The earliest work is a simple subtyping between input and output capabilities (called IO-subtyping) [33], which has been extended and implemented in different areas of concurrency (e.g. [17, 34]) and has been continuously studied as one of the core subjects in concurrency (e.g. [19]). Later, a generic type system with subtyping is introduced in [24], where the subtyping plays a fundamental rôle to generate a variety of interesting type systems as its instances.

More recently, another subtyping based on session types [12] has been applied to many aspects of software design and implementations such as web services, programming languages and distributed computing [4, 8, 14, 22, 33, 37]. The standpoint of session types is that communication-centred applications exhibit a highly structured sequence of interactions involving, for example, sequencing, branching, selection and recursion, and such a series of interactions can be abstracted as a session type through a simple syntax. The session subtyping specified along session structures is then used for validating a large set of programs, giving flexibility to programmers.

As an example of session subtyping [4, 8], consider a simple protocol between a Buyer and a Seller from Buyer’s viewpoint: Buyer sends a book’s title (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, he then sends his address (a string) and Seller sends back the delivery date (a date); otherwise he quits the conversation. This can be described by the session type:

\[
!\langle \text{string}\rangle.\langle \text{int}\rangle.\langle \text{ok}\langle \text{string}\rangle\rangle.\langle \text{date}\rangle.\text{end} \oplus \langle \text{quit}\rangle.\text{end} \tag{1}
\]

The prefix \( !\langle \text{string}\rangle \) denotes an output of a value of type string, whereas \( \langle \text{int}\rangle \) denotes an input of a value of type int. Instead \( \text{ok} \) and \( \text{quit} \) are labels distinguishing different branches. The operator \( \oplus \) is an internal choice, meaning the process may choose to either send the label \( \text{ok} \) with a string and receive a date, or send the label quit. The type end represents the termination of the session. From Seller’s viewpoint the same session is described by the dual type

\[
?\langle \text{string}\rangle.\langle \text{int}\rangle.\langle \text{!ok}\langle \text{string}\rangle\rangle.\langle \text{date}\rangle.\text{end} \oplus \langle \text{quit}\rangle.\text{end} \tag{2}
\]

in which \( \& \) means that the process offers two behaviours, one where it receives \( \text{ok} \) with a string and sends a date, and one where it receives \( \text{quit} \).

As \( \text{nat} \leq \text{real} \) in the standard subtyping, a type representing a more defined behaviour is smaller. A selection subtype is a type which selects among fewer options (as outputs). The following is an example of a subtype of (1):

\[
!\langle \text{string}\rangle.\langle \text{int}\rangle.\langle \text{ok}\langle \text{string}\rangle\rangle.\langle \text{date}\rangle.\text{end} \tag{3}
\]

Conversely, a branching subtype is a type which offers more options (as inputs). The following is an example of a subtype of (2):

\[
?\langle \text{string}\rangle.\langle \text{date}\rangle.\text{end} \oplus \langle \text{quit}\rangle.\text{end} \oplus \langle \text{later}\rangle.\text{end} \tag{4}
\]

Intuitively, a type \( T \) is a subtype of a type \( S \) if \( T \) is ready to receive no fewer labels than \( S \), and \( T \) potentially sends no more labels than \( S \) (in other words, \( T \) represents a more permissive behaviour than \( S \) [4, 8]. If we run two processes typed by (3) and (4), they are type safe, i.e. there is no mismatch of labels or types during communication. Hence the subtyping is sound with respect to the type safety. An important question, however, is still remaining: is this subtyping complete? I.e. is this session subtyping the greatest relation which does not violate type safety? The proof...
of soundness is usually immediate as a corollary of the subject reduction theorem. But how can we state and prove completeness?

**Preciseness** This paper first gives a general formulation of soundness and completeness for type safety in concurrency, which does not depend on specific subtypings. We say that a subtyping relation is *precise* if it is both sound and complete. The preciseness in this paper is a simple operational property that specifies a relationship between static and dynamic semantics. To formally define preciseness, we assume a multi-step reduction between processes $P \rightarrow_\ast P'$ (where $P'$ is possibly the error process) as well as typing judgements of the form $P : \{a : T\}$, assuming that the process $P$ has a single free channel $a$ whose type is $T$. We also use reduction contexts $C$ in the standard way. The judgement $C[a : T] : \emptyset \iff X : T \vdash C[X(a)] : \emptyset$, where $X$ is a process variable which does not appear in $C$, see §2.

Our preciseness definition is an adaptation of the preciseness definition for the call-by-value $\lambda$-calculus with sums and product types given in §3.

**Definition 1.1** (Preciseness). A subtyping $\leq$ is *precise* when, for all session types $T$ and $S$:

$$T \leq S \iff \left( \begin{array}{l}
\text{there do not exist } C \text{ and } P \text{ such that:} \\
(C[a : T] : \emptyset \iff X : T \vdash C[X(a)] : \emptyset) \\
\text{and } C[P] \rightarrow_\ast \text{error}
\end{array} \right)$$

When the only-if direction ($\Rightarrow$) of this formula holds, we say that the subtyping is *sound*; when the if direction ($\Leftarrow$) holds, we say that the subtyping is *complete*.

The property of preciseness tells that an arbitrary context which is safe when filled with any process using a channel following $S$, is also safe when filled with a process using the same channel following $T$, if and only if $T \leq S$. Note that the ($\Rightarrow$) direction is useful to check that the subtyping preserves the desired safety property, while the ($\Leftarrow$) direction is essential to assure that the subtyping does cover all processes which satisfy the desired safety property.

Here we are interested in *syntactically* defined subtyping and an *operational* notion of preciseness. Our approach is opposed to semantic subtyping [11], which is given denotationally: in addition, the calculus of [11] has a type case constructor from which completeness follows for free. See §7 for a detailed discussion.

**Preciseness and impreciseness for the $\pi$-calculus** Then $\Omega$-subtyping [13] is not precise. This is because no error can be detected when a read only channel is used to write, or vice versa, by a context without type annotations. For a similar reason, the branching and selection subtyping [8] is also imprecise for the $\pi$-calculus. The branching and selection subtyping is instead precise for the $\pi$-calculus with only linear channels [24], whose expressivity is limited.

In [23] only necessary conditions are stated for subtyping, the aim being that of having the maximum generality. The subtyping relations in the instances of the generic type system depend on the properties (arity-mismatch check, race detection, static garbage-channel collection, deadlock detection) one wants to guarantee.

These results led us to consider preciseness for two representative session calculi, the synchronous [21–23] and the asynchronous [27–29] session calculi.

**Two preciseness results** Session types have sufficiently rich structure to assure completeness, hence if $T \not\leq S$, then $T$ and $S$ can be distinguished by suitable contexts and processes.

The first result of this paper is preciseness of the branching-selection subtyping (dubbed also synchronous subtyping) described above for the synchronous session calculus. Our motivation to study the first result is to gently introduce a proof method for preciseness and justify the correctness of the synchronous subtyping, which is widely used in session-based calculi, programming languages and implementations [4][8][21][22][25].

The case of the asynchronous session calculus is more challenging. The original session typed calculi are based on synchronous communication primitives, assumed to be compiled into asynchronous interactions using queues. Later researchers found that, assuming ordered asynchronous communications for binary interactions, one could directly express asynchronous non-blocking interactions. One can then assure not only the original synchronous safety, but also the asynchronous safety, i.e. deadlock-freedom (every input process will always receive a message) and orphan message-freedom (every message in a queue will always be received by an input process).

Our first observation is that the branching-selection subtyping is not large enough for the asynchronous calculus, i.e. there exist session types $T$ and $S$ such that $T$ can safely be considered as a subtype of $S$, but $T \not\leq S$ is not derivable by the subtyping. The reason is natural: in the presence of queues, the processes typed by the following two non-dual types can run in parallel without compromising type safety:

$$T_a = !\langle\text{int},!\langle\text{char},?\langle\text{string},?\langle\text{nat}\rangle\rangle\rangle\rangle\text{.end}$$
$$T_b = !\langle\text{string},!\langle\text{nat},?\langle\text{int}\rangle\rangle\rangle\text{.end}$$

since the process typed by $T_a$ can put two messages typed by $\text{int}$ and $\text{char}$ in one queue and the process typed by $T_b$ can put two messages typed by $\text{string}$ and $\text{nat}$ in another queue, and they can receive the two messages from each queue, without getting stuck.

The asynchronous subtyping of [27–29] permutes the order of messages, for example:

$$T_c \leq ?\langle\text{string},?\langle\text{nat}\rangle\rangle\langle!\langle\text{int},!\langle\text{char}\rangle\rangle\rangle\text{.end}$$

so that the process typed by $T_c$ can have a type which is dual of $T_b$ by the subsumption rule. This asynchronous permutation is often used as a means of messaging optimisation, e.g., as “message overlapping” in the parallel programming community [30][51]. Our result demonstrates the preciseness of this subtyping, which was introduced for practical motivations.

We have found that the subtyping of [27] is unsound if we require the absence of orphan messages. If we allow orphan messages and we only have deadlock errors, then the subtyping of [27] is sound but not complete. All this is discussed in §7.

The subtypings of [28][29], whose targets are the higher-order $\pi$-calculus and the multiparty session types, respectively, are sound for deadlock and orphan message errors. Hence we simplify the subtypings of [28][29] and we adapt them to the binary session $\pi$-calculus.

**Contributions and outline** As far as we are aware, this is the first time that completeness of subtypings, which is solely based on (un-)typed operational semantics, is formalised and proved in the context of mobile processes. We also demonstrate its applicability to two session type disciplines, the synchronous and the asynchronous one. The most technical challenge is the proof of completeness for the asynchronous subtyping, which requires some ingenuity in the definition of its negation relation. Key in the proofs is the construction of processes which characterise types. These processes allow us to show also the denotational preciseness of both synchronous and asynchronous subtypings.

§2 defines the synchronous session calculus and its typing system, and proves soundness of the branching-selection subtyping. §3 proposes a general scheme for showing completeness and proves completeness of subtyping for the synchronous session calculus. §4 defines the asynchronous session calculus and introduces a new asynchronous subtyping relation which is shown to be sound.
§5 proves completeness of subtyping for the asynchronous calculus. The last completeness proof is non-trivial since the permutations introduced by the asynchronous subtyping rules make session types unstructured. The proof of operational preciseness uses denotational preciseness of both synchronous and asynchronous subtypings, as shown in §4. Related work and conclusion are the contents of §7 and §8, respectively.

2. Synchronous Session Calculus

This section starts by introducing syntax and semantics of a simplification of the most widely studied synchronous session calculus [21]. Since our main focus is on theorems about session types, we eliminate the session initiations (shared channels) and assume that the obtained calculus is similar to that presented in [36].

The extension of preciseness to the full calculus with expressions and shared channels is mechanical, see §8.

2.1 Syntax

A session is a series of interactions between two parties, possibly with branching and recursion, and serves as a unit of abstraction for describing communication protocols. The syntax is given in Table 1. We use the following base sets: variables, ranged over by \( x, y, z, \ldots \); names (often called channels), ranged over by \( a, b, \ldots \); identifiers (names and variables), ranged over by \( u, u', \ldots \); labels, ranged over by \( I, I', \ldots \); process variables, ranged over by \( X, Y, \ldots \); and processes, ranged over by \( P, P', \ldots \).

Session communications are performed between an output process \( u!l(x).P \) and an input process \( \sum_{i \in I} u!l_i(x).P_i \) (the \( l_i \) are pairwise distinct), where the former sends a channel choosing one of the branches offered by the latter. In \( \sum_{i \in I} u!l_i(x).P_i \) the identifier \( u \) is the subject of input and output, respectively. The choice \( P \oplus Q \) internally chooses either \( P \) or \( Q \). In many session calculi \([4, 21, 29]\) the conditional plays the role of the choice. The process \( \text{def } D \text{ in } P \) is a recursive agent and \( X(\bar{a}) \) is a recursive variable. The process \( (vab)P \) is a restriction which binds two channels, \( a \) and \( b \) in \( P \), making them co-channels, i.e., allowing them to communicate (see rule \( \text{[R-COM-SYNC]} \)) in Table 2. This double-restriction is commonly used in the recent literature of session types, e.g., \([12, 35]\). We often omit \( \emptyset \) from the tail of processes.

The bindings for variables are in inputs and declarations; those for channels are in restrictions; and those for process variables are in declarations. The derived notions of bound and free identifiers, alpha equivalence, and substitution are standard.

By \( \text{fpv}(P) / \text{fnv}(P) \) we denote the set of free process variables/free names in \( P \). By \( \text{sn}(P) \) we denote the set of free process variables/free names in \( P \). Defining by \( \text{sn}(u!l(x).P) = \text{fn}(u) \cup \text{sn}(P) \) and \( \text{sn}(\sum_{i \in I} u!l_i(x).P_i) = \text{fn}(u) \cup \bigcup_{i \in I} \text{sn}(P_i) \) as expected in the other cases.

\[
\begin{align*}
P &::= 0 \quad \text{ Process } \\
&| X(\bar{a}) \quad \text{(variable) } \quad D &::= \text{ Declaration } \\
&| \sum_{i \in I} u!l_i(x).P_i \quad \text{(input) } \quad X(\bar{x}) = P \\
&| u!l(x).P \quad \text{(output) } \quad \text{Identifiers } \\
&| P \oplus Q \quad \text{(choice) } \quad u \quad \text{a (name) } \\
&| \text{def } D \text{ in } P \quad \text{parallel) } \quad (x \text{ (variable) } \\
&| (vab)P \quad \text{(definition) } \quad \text{error } \quad \text{(error) } \\
&| \end{align*}
\]

Table 1. Syntax of synchronous processes.

\[
\begin{align*}
(\text{R.COM-SYNC}) & \quad k \in I \\
&\quad \frac{(\text{vab})(a!l(c).P) \sum_{i \in I} b!l_i(x_i).Q_i}{(\text{vab})(P\{Q_k \{c/x_k\}\})} \\
(\text{R-DEF}) & \quad \text{def } X(\bar{x}) = P \text{ in } (X(\bar{a})\mid Q) \rightarrow \text{def } X(\bar{x}) = P \text{ in } (P[\bar{a}/\bar{x}]\mid Q) \\
(\text{R-CONTEXT}) & \quad \frac{P \rightarrow P'}{C[P] \rightarrow C[P']} \\
&\quad \frac{P \equiv P'}{P \rightarrow Q' \quad Q' \equiv Q} \\
\end{align*}
\]

Table 2. Reduction of synchronous processes.

\[
\begin{align*}
(\text{S-PAR 1}) & \quad 0 \equiv P \equiv P \mid 0 \equiv P \quad (P \mid Q) \equiv (P \mid Q) \mid R \\
(\text{S-PAR 2}) & \quad P \mid Q \equiv Q \mid P \equiv P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
(\text{S-PAR 3}) & \quad P \oplus Q \equiv Q \oplus P \equiv (P \oplus Q) \oplus R \equiv Q \oplus (P \oplus R) \\
(\text{S-CH 1}) & \quad a,b \not\in \text{fn}(Q) \\
(\text{S-CH 2}) & \quad (\text{a,b}) \cap \{c,d\} = \emptyset \\
(\text{S-CHOICE}) & \quad \frac{(vab)P \equiv (vab)(P\{Q\})}{(vab)(vcd)P \equiv (vcd)(vab)P} \\
(\text{S-STRUCT}) & \quad \text{def } D \text{ in } 0 \equiv 0 \\
(\text{S-DEF 2}) & \quad \text{def } D \text{ in } (vab)(P) \equiv (vab)(\text{def } D \text{ in } P) \\
(\text{S-DEF 3}) & \quad \text{dpv}(D \cap \text{fpv}(Q) = \emptyset) \\
(\text{S-DEF 4}) & \quad \text{dpv}(D \cap \text{dpv}(D') = \emptyset) \\
\end{align*}
\]

Table 3. Structural congruence for synchronous processes.

\[
\begin{align*}
\text{Table 2:} & \quad \text{Reduction of synchronous processes.} \\
\text{Table 3:} & \quad \text{Structural congruence for synchronous processes.} \\
\text{2.2 Operational semantics} & \quad \text{Table 2 gives the reduction relation between the synchronous processes which do not contain free channel variables. It uses the following reduction context:} \\
&\quad \text{C ::= [ ] } \quad \text{C} \mid P \mid (vab)C \mid \text{def } D \text{ in } C \\
&\quad \text{and the structural rules of Table 3. By dpv}(P) \text{ we denote the set of free process variables introduced in declarations, whose definition by induction on processes has only one interesting case: } \\
&\quad \text{dpv}(X(\bar{x}) = P) = \{X\} \cup \text{dpv}(P). \\
&\quad \text{In Table 2 [R-COM-SYNC] is the main communication rule between input and output at two co-channels } a \text{ and } b, \text{ where the label } l_k \text{ is selected and channel } c \text{ is instantiated into the } k \text{-th input branch. Other rules are standard.} \\
&\quad \text{We also define error reduction (Table 2), which is crucial for stating the preciseness theorem. We do not consider errors due to non linear use of channels, since they cannot arise reducing processes typed with an unsound subtyping. Rule [ERR-MISM-SYNC] is a mismatch between the output and input labels. Rule [ERR-NEW-SYNC] represents an error situation where one of two co-channels (b) is missing. Rule [ERR-OUT-OUT-SYNC] gives an error when two co-channels are both subjects of outputs, destroying the duality of sessions. Similarly rule [ERR-IN-INSYNC] gives an error when two}
\end{align*}
\]
co-channels are both subjects of inputs. We denote by \( \rightarrow_{s} \) the reduction relation for the synchronous processes, generated by the rules in Tables 2 and 3 and by \( \rightarrow_{s} \) the reflexive and transitive closure of \( \rightarrow_{s} \).

### 2.3 Typing synchronous processes

The syntax of synchronous session types, ranged over by \( T \) and \( S \), is:

\[
T, S ::= \&_{i \in I} \eta(S_i) T_i | \bigoplus_{i \in I} \eta(S_i) T_i | T[\mu.T] | \text{end}
\]

The branching type \( \&_{i \in I} \eta(S_i) T_i \) describes a channel willing to branch on an incoming label \( l_i \), receive a channel of type \( S_i \), and then continue its interaction as prescribed by \( T_i \). The selection type \( \bigoplus_{i \in I} \eta(S_i) T_i \) is its dual: it describes a channel willing to send a label \( l_i \) with a channel of type \( S_i \), and then continue its interaction as prescribed by \( T_i \). In branching and in selection types:

- the labels are pairwise distinct
- the types of the exchanged channels are closed.

We omit \& and \( \bigoplus \) and \( \otimes \) and labels when there is only one branch.

We use \( t \) to range over type variables. The type \( \mu.T \) is a recursive type. We take an equi-recursive view of types, not distinguishing between a type \( \mu.T \) and its unfolding \( T[\mu.T/t] \). We assume that the recursive types are guarded, i.e. \( \mu.T \) is not a type. The type \( \text{end} \) represents the termination of a session and it is often omitted. In the examples we use infix notation for \& and \( \bigoplus \) and \& and ground types (int, bool, ...) for messages. The extension to ground types is easy, as discussed in §4.

As usual session duality [31] plays an important rôle for session types. The function \( T \), defined below, yields the dual of the session type \( T \).

\[
& \quad \&_{i \in I} \eta(S_i) T_i \rightarrow \bigoplus_{i \in I} \eta(S_i) T_i \rightarrow \bigoplus_{i \in I} \eta(S_i) T_i \rightarrow \&_{i \in I} \eta(S_i) T_i \\
\text{end} \rightarrow \text{end}
\]

We write \( T_1 \triangleright T_2 \) if \( T_2 = T_1 \).

Table 4 defines the subtyping. Note that the double line in rules indicates that the rules should be interpreted coinductively [32] Chapter 21, §2.1). We follow the ordering of the branching-

### 5. Subtyping rules for synchronous types.

co-channels are both subjects of inputs. We denote by \( \rightarrow_{s} \) the reduction relation for the synchronous processes, generated by the rules in Tables 2 and 3 and by \( \rightarrow_{s} \) the reflexive and transitive closure of \( \rightarrow_{s} \).

### 6. Typing rules for synchronous processes.

Rule \([\text{sub-bra}]\) states that the branching which offers fewer branches is a subtype of the one with more branches; and rule \([\text{sub-sel}]\) is its dual (see the explanations in §4). We write \( T \leq_{s} S \) if \( T \leq S \) is derived by the rules in Table 5. Reflexivity of \( \leq_{s} \) is immediate and transitivity of \( \leq_{s} \) can be shown in the standard way.

The typing judgements for synchronous processes take the following form: \( \Gamma \vdash_{s} P : \Delta \), where \( \Gamma \) is the shared environment which associates process variables to sequences of session types and \( \Delta \) is the session environment which associates identifiers to session types. They are defined by:

\[
\Gamma ::= \emptyset | \Gamma, X : \langle T \rangle \quad \Delta ::= \emptyset | \Delta, u : T
\]

We write \( \Delta_1 \triangleleft \Delta_2 \) for \( \Delta_1 \cup \Delta_2 \) when \( \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset \). We say that \( \Delta \) is end-only if \( u : T \notin \Delta \) implies \( T \not\in \text{end} \).

We define a pre-order between the session environments which reflects subtyping. More precisely, \( \Delta_1 \leq_{s} \Delta_2 \) if:

\[
\begin{align*}
& u \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \implies \Delta_1(u) \leq_{s} \Delta_2(u) \\
& u \notin \text{dom}(\Delta_1) \implies \Delta_1(u) = \emptyset \\
& u \notin \text{dom}(\Delta_2) \implies \Delta_2(u) = \emptyset
\end{align*}
\]

We write \( \Delta_1 \triangleleft \Delta_2 \) if \( \Delta_1 \leq_{s} \Delta_2 \) and \( \Delta_2 \leq_{s} \Delta_1 \). It is easy to verify that \( \Delta \) is end-only iff \( \Delta \triangleleft \emptyset \).

Table 6 gives the typing rules. They are standard in session calculi, see e.g. [12]. Rule \([\text{t-idle}]\) is the introduction rule for the nil process. To type an input process, rule \([\text{t-input}]\) requires the type \( S_j \) of variable \( x_j \) and the type \( T_i \) of channel \( u \) for the continuation \( P_i \). In the resulting session environment, the type \( u \) has the branching type in which \( u \) receives \( S_j \) and then continues with \( T_i \) for each label \( l_i \). The rule for typing output processes is similar and simpler. In rule \([\text{t-par}]\), the session environment of \( P_1 \) or \( P_2 \) is the disjoint union of the environments \( \Delta_1 \) and \( \Delta_2 \) for the two processes, reflecting the linear nature of channels. Contrarily, in rule \([\text{t-choice}]\), the two processes share the same session environment, since at most one of them will be executed. Rule \([\text{t-new-sync}]\) is a standard rule for name binding, where we ensure the co-channels have dual types. Rules \([\text{t-var}]\) and \([\text{t-def}]\) deal with process calls and definitions, requiring the channel parameters have the types which are assumed in the shared environment. Rule \([\text{t-var}]\) gives these types to the arguments of the process variable. In \([\text{t-def}]\), the parameters of the process associated with the process variable must be typed with
these types. The assumption on parameter types is also to type the body of the definition. We write $\Gamma \vdash_s P \triangleright \Delta$ if $P$ is typed using the rules in Table 6.

2.4 Soundness of synchronous subtyping

Our type system enjoys the standard property of subject reduction. Notice that session environments are unchanged since only bound channels can be reduced.

Theorem 2.1 (Subject reduction for synchronous processes). If $\Gamma \vdash_s P \triangleright \Delta$ and $P \rightarrow s P'$, then $\Gamma \vdash_s P' \triangleright \Delta$.

From subject reduction we can easily derive that well-typed processes cannot produce error.

Corollary 2.2. If $\Gamma \vdash_s P \triangleright \Delta$, then $P \not\rightarrow^* s$ error.

The proof of soundness theorem follows easily.

Theorem 2.3. The synchronous subtyping relation $\leq_s$ is sound for the synchronous calculus.

3. Completeness for Synchronous Subtyping

This section proves the first main result, completeness of synchronous subtyping, which together with soundness shows the preciseness theorem. We shall take the following three steps.

- **Step 1** For each type $T$ and identifier $u$, we define a characteristic process $P(u, T)$ typed by $u : T$, which offers the series of interactions described by $T$ on $u$.

- **Step 2** We characterise the negation of the subtyping relation by inductive rules (notation $\not\triangleright$).

- **Step 3** We prove that if $T \not\triangleright S$, then the parallel composition of $P(u, T)$ and $P\langle b, S \rangle$ in the scope of $\triangleright$ reduces to error. Hence choosing $P = P(u, T)$ and $C = \langle \triangleright \rangle \in \{P(b, S)\}$ in the definition of preciseness (Definition 1.1), we achieve completeness.

The same three steps will be used for the completeness proof in the asynchronous case.

**Characteristics of synchronous processes** The characteristics of synchronous processes are defined following the structure of types.

**Definition 3.1** (Characteristics of synchronous processes). The characteristic process offering communication $T$ on identifier $u$ for the synchronous calculus, denoted by $P(u, T)$, is defined by:

$$
\begin{align*}
P(u, T) &\equiv \begin{cases} 
\sum_{i \in J} P\langle u, i, S_i, T_i \rangle & \text{if } T = \& \sum_{i \in J} S_i, T_i \\
\bigoplus_{i \in J} P\langle u, i, S_i, T_i \rangle & \text{if } T = \bigoplus_{i \in J} S_i, T_i \\
X_u(x) = P\langle x, S \rangle \quad \text{if } T = \mu x.S & \text{if } T = \mu x.S \\
0 & \text{if } T = \text{end}
\end{cases}
\end{align*}
$$

A branching type is mapped to the inputs $P\langle u, I, S_i, T_i \rangle \quad (i \in I)$, which uses the input channel $x$ in $P\langle x, S \rangle$. A selection type is mapped to the choice between the outputs $P\langle u, I, S_i, T_i \rangle \quad (i \in I)$, where the fresh channel $x$ carried by $u$ will be received by the dual input, which will interact with the process $P\langle b, S \rangle$. A recursive type is mapped in a definition associated to the characteristic process of the type body. The process body of this definition is just a call to the process variable associated to the recursion type variable.

Type $\text{end}$ is mapped to 0.

<table>
<thead>
<tr>
<th>$\text{N-END R}$</th>
<th>$\text{N-END L}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \neq \text{end}$</td>
<td>$T \neq \text{end}$</td>
</tr>
<tr>
<td>end $\not\rightarrow^*$</td>
<td>$T \not\rightarrow^*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{N-BRASEL}$</th>
<th>$\text{N-SELBRA-SYNC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists j \in L S_j \neq T_j$ &amp; $\forall j \in L S_j = T_j$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{N-EXCH BRA}$</th>
<th>$\text{N-EXCH SEL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists i \in L S_i \neq T_i$ &amp; $\forall i \in L S_i = T_i$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{N-CONT BRA}$</th>
<th>$\text{N-CONT SEL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists i \in L S_i \neq T_i$ &amp; $\forall i \in L S_i = T_i$</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Negation of synchronous subtyping rules.

For example if $T = \nu x \langle \text{end} \rangle \oplus \nu x \langle \text{end} \rangle$, then $P\langle x, T \rangle = P\langle x, \nu x \langle \text{end} \rangle \oplus \nu x \langle \text{end} \rangle \rangle$.

**Rules for negation of synchronous subtyping** Table 7 defines the rules which characterise when a type is not subtype of another type. Rules [N-END R] and [N-END L] say that $\text{end}$ cannot be a super or subtype of a type different from $\text{end}$. Rule [N-BRASEL] says that a branching type cannot be subtype of a selection type. Rule [N-SELBRA-SYNC] is its dual. Rules [N-EXCH BRA] and [N-EXCH SEL] represent the cases that the labels do not conform the subtyping rules. Rules [N-CONT BRA] and [N-CONT SEL] represent the cases that carried types do not match the subtyping rules. Lastly rules [N-CASE BRA] and [N-CASE SEL] represent the cases that continuations do not match the subtyping rules. Notice that the rules [N-CASE BRA] are the negations of rule [SUB-BRA], and the rules [N-CASE SEL] are the negations of rule [SUB-SEL], where $* \in \{\text{LABEL, EXCHANGE}\}$. We write $T \not\triangleright S$ if $T \not\triangleright S$ is generated by the rules in Table 7.

It is easy to verify by induction on types that $\not\triangleright$ is the negation of the synchronous subtyping.

**Proposition 3.3** $S \leq_s T$ is not derivable if and only if $S \not\triangleright_s T$ is derivable.

The main theorem for synchronous subtyping can now be stated. As we described in **Step 3**, by Proposition 3.3 we only have to prove that if $T \not\triangleright_s S$, then

$$
\langle \triangleright \rangle \langle P\langle a, T \rangle \mid P\langle b, S \rangle \rangle \rightarrow_s \text{error}
$$

where $P\langle a, T \rangle$, $P\langle b, S \rangle$ are characteristic synchronous processes. The proof is by induction on $T$ and $S$ and by cases on the definition of $\not\triangleright_s$. 
4. Asynchronous Session Calculus

“Asynchrony” in communication means that messages are non-blocking but their order is preserved. We use FIFO queues at both co-channels to model actions performed from the underlying network such as TCP. This double-queue formulation is the most common for the asynchronous sessions and follows the recent formalism in [13][27][28].

4.1 Syntax and operational semantics

Table 3 shows the asynchronous session calculus obtained by extending the synchronous calculus of Table 1 with queues. A queue $ab \triangleright h$ is used by channel $a$ to enqueue messages in $h$ and by channel $b$ to dequeue messages from $h$. We extend the definition of the set of free names in queues by tending the synchronous calculus of Table 1 with queues.

We use the structural congruence defined by adding the rules in Table 8 to the rules of Table 3. Rule $[\text{S-NULL}]$ represents garbage collection of empty queues.

We use Table 9 to the rules of Table 3. Rule $[\text{SYNCH-ASYNC}]$ enqueues messages and rule $[\text{RECEIVE-ASYNC}]$ dequeues messages. We write $P \rightarrow_a Q$ if $P$ is derived by the rules of Table 9 but rule $[\text{COM-SYN-COM}]$ and by the rules of Table 10.

4.2 Errors in asynchronous processes

Unlike the synchronous case, defining the error reductions for asynchronous communication is not trivial essentially for the presence of queues. We need to identify the following classical error situations (the terminology is from [10]):

1. **deadlocks**: there are inputs waiting to dequeue messages from queues which will be forever empty.

2. **orphan message errors**: there are messages in queues which will be never received by corresponding inputs, i.e. orphan messages will remain forever in queues.

Both errors are important, since we want to ensure every input can receive a message and every message in a queue can be read. These errors correspond to the following processes:

1. **deadlocks**: both bounded channels are waiting for inputs and their queues are both empty or one channel is waiting for an input with an empty queue and the corresponding channel only occurs as name of queues. In the first case the process has the shape (we can omit the binding $(vab)$ since both channels $a, b$ appear in processes and the queues prescribe them to communicate)

$$\sum_{i \in I} a^\triangleright l_i(x_i).P_i \ | \ \sum_{j \in J} b^\triangleright l_j(x_j).Q_j \ | \ ba \triangleright \emptyset \ | \ ab \triangleright \emptyset$$

$$P ::= \quad \text{Process} \quad h ::= \quad \text{Queue}
\begin{array}{ll}
\vdots & \emptyset \quad (\text{empty}) \\
| \quad \text{from Table 11} & \quad l(a) \quad (\text{message}) \\
| \quad ab \triangleright h \quad (\text{queue}) & \quad h \cdot h \quad (\text{composition})
\end{array}$$

Table 8. Syntax of asynchronous processes.

and in the second case the process has the shape

$$(vab)(\sum_{i \in I} a^\triangleright l_i(x_i).P_i | ba \triangleright \emptyset | ab \triangleright h),$$

where $b$ does not occur free in $P_i$ for all $i \in I$.

(2) **orphan message errors**: a queue is not empty, but the corresponding channel will never appear as subject of an input. I.e. the process has the shape $(vab)(P | ba \triangleright h)$ where $h$ is not empty, but $P$ will neither reduce to a parallel process containing an input on channel $a$ nor pass the channel $a$ to an outer process.

To define statically the second error situation, we need to compute an over approximation (denoted by $\Phi(P)$) of the set of names which might appear as subjects of inputs by reducing a process $P$. Notice that to consider the set of names which occur in a process is too large, since for example a recursive process which always sends messages will always contain both the names of the subjects and of the objects of the outputs, but it will never read a message on its queue. The definition of $\Phi$ requires some care, as shown for example by the process:

$$P = b^\triangleright l_0(x) . c^\triangleright l_1(y) . l_0(a) . db \triangleright \emptyset$$

(4)

Notice that $P$ does not contain inputs with subject $a$, but

$$P \rightarrow_a b^\triangleright l_0(x) . c^\triangleright l_1(y) . l_0(a) . db \triangleright \emptyset$$

and this last process has an input with subject $a$. Hence to define $\Phi(P)$, we need to take names carried by outputs as well as names occurring in messages inside queues. For the choice, only one of the two processes could have an input on channel $a$, so the error could arise only if and when the other process is selected. Another delicate case in the definition of $\Phi(P)$ comes from recursive definitions, as exemplified in Example 4.2. For simplicity we consider process variables with only one parameter. This is enough since we look for errors generated by characteristic asynchronous processes (see Definition 5.1) and they satisfy this restriction. We use two auxiliary mappings. The mapping $\delta$ from processes and sets of declarations to sets of identifiers and the mapping $\gamma$ from queues to sets of channels.

**Definition 4.1.** (Mapping $\Phi$) The mapping $\gamma$ is defined by induction on queues:

$$\gamma(\emptyset) = \emptyset \quad \gamma(l(a)) = \{a\} \quad \gamma(h_1 \cdot h_2) = \gamma(h_1) \cup \gamma(h_2)$$

The mapping $\delta$ is defined by induction on processes in Table 11. The mapping $\Phi$ is defined by $\Phi(P) = \delta(P, \emptyset)$.

Table 9. Structural congruence for asynchronous processes.

$$\begin{array}{cccc}
| & [\text{S-NULL}] & [\text{S-QUEUE-EQUIV}] & [\text{S-QUEUE}] \\
\hline
\text{SEND-ASYNC} & (vab)(ab \triangleright \emptyset | ba \triangleright \emptyset) \equiv 0 & h \equiv h' & \frac{ab \triangleright h \equiv ab \triangleright h'}{ab \triangleright h \equiv h'} \\
\text{RECEIVE-ASYNC} & \frac{h \cdot Q \equiv h \cdot \emptyset}{} & \frac{h \cdot \emptyset \equiv h}{} & \frac{h_1 \cdot (h_2 \cdot h_3) \equiv (h_1 \cdot h_2) \cdot h_3}{}
\end{array}$$

Table 10. Reduction of asynchronous processes.
Then we obtain:

\[
\begin{align*}
\delta(0, D) &= \emptyset \\
\delta(X(u), D) &= \{ u \} \\
\delta(\Sigma_{i \in I} u ! l_i(x_i), P \cdot D) &= \{ u \} \cup \bigcup_{i \in I} (\delta(P_i, D) \setminus \{ x_i \}) \\
\delta(\mu u ! x.P, D) &= \emptyset \\
\delta(P_1 \parallel P_2, D) &= \emptyset \\
\delta(\text{def } D \text{ in } P, D) &= \emptyset \\
\delta((v a \odot P), D) &= \emptyset \\
\delta(\text{error}, D) &= \gamma(h) \\
\delta(ab \bullet h, D) &= \gamma(h)
\end{align*}
\]

Table 11. The mapping \( \delta \).

For the process \( P \) of [4], we obtain \( \varphi(P) = \{a, b\} \).

**Example 4.2.** Assume

\[
P = \text{def } X(x) = a ! l(x) \text{ in } X(c) \mid b ! l(y), y ! l(z) \mid ab \bullet \varnothing \mid ba \bullet \varnothing
\]

Then we obtain:

\[
\begin{align*}
\varphi(P) &= \delta(P, \emptyset) \\
&= \delta(X(c)) = \delta(b ! l(y)) = \delta(y ! l(z)) = \delta(ab \bullet \varnothing) = \delta(ba \bullet \varnothing) = \emptyset \\
\varphi(P) &= \delta(X(c)) \cup \delta(b ! l(y)) \cup \delta(y ! l(z)) \cup \delta(ab \bullet \varnothing) \cup \delta(ba \bullet \varnothing) \\
&= (\{(c\langle c/x \rangle \cup \{b\} \cup (\delta(y ! l(z), D) \setminus \{y\}) \\
&= (\{(c\langle c/x \rangle \cup \{b\} = \{c, b\})
\end{align*}
\]

where \( D = X(x) = a ! l(x) \). The evaluation of \( \varphi(P) \) tells that \( c \) can become the subject of an input. We illustrate this fact also by the following reduction:

\[
\begin{align*}
\text{def } X(x) = a ! l(x) \text{ in } X(c) \mid b ! l(y), y ! l(z) \mid ab \bullet \varnothing \mid ba \bullet \varnothing \\
\rightarrow_2 \text{def } X(x) = a ! l(x) \text{ in } a ! l(c) \mid b ! l(y), y ! l(z) \mid ab \bullet \varnothing \mid ba \bullet \varnothing \\
\rightarrow_2 \text{def } X(x) = a ! l(x) \text{ in } b ! l(y), y ! l(z) \mid ab \bullet l(c) \mid ba \bullet \varnothing \\
\rightarrow_2 \text{def } X(x) = a ! l(x) \text{ in } c ! l(z) \mid ab \bullet \varnothing \mid ba \bullet \varnothing
\end{align*}
\]

The error reduction rules for asynchronous processes are the rules of Table 12 plus rule [ERR-CONTEXT] of Table 4. Rule [ERR-MISM-ASYNC] deals with a label mismatch between a message on the top of the queue and an input. Rule [ERR-IN-ASYNC] gives an error when two processes with bounded channels are in deadlock waiting to read from empty queues. Rule [ERR-ASYNC] deals with the case of one process waiting to read from an empty queue which will never contain a message, since there are no occurrences of the unique channel that can enqueue messages. Rule [ERR-ORPH-MESS-ASYNC] corresponds to the orphan message error. In this rule the condition \( fpv(P) = \emptyset \) assures that we consider all needed declarations in computing \( \varphi(P) \). We write \( P \rightarrow \text{error} \) if \( P \rightarrow \text{error} \) can be derived using the rules of Table 12 plus rule [ERR-CONTEXT] of Table 4. The notation \( \rightarrow_2 \) is used with the expected meaning.

Our definition of error reductions captures all deadlocks and orphan message errors that can be generated by reducing parallel compositions of two characteristic asynchronous processes with the two required queues. This is a consequence of preciseness of subtyping (see Theorem 4.12 and the proof of Theorem 5.8) a process \( P(a, T) \mid P(b, S) \mid ab \bullet \varnothing \mid ba \bullet \varnothing \) is typable (in the sense defined in § 4.4) if and only if it does not reduce to error. The function \( \varphi \) is then enough for the definition of error. Obviously arbitrary processes can be stuck without reducing to error. Typical cases are processes in which companion processes or queues are missing, or sessions are interleaved. A simple example is the characteristic process \( P(a, b \text{end}) = a ! l(x) \), which is deadlocked. Note that the type system of § 4.4 is an adaptation of those in [28, 29] to our calculus, and it does not aim to avoid deadlock or orphan message errors.

### 4.3 Asynchronous subtyping

The asynchronous subtyping is not only essential for the completeness result, but it is also important in practice. As observed in [38], implementing this subtyping is a key tool for maximising message-overlapping in the high-performance computing environments. To explain the usefulness of the asynchronous subtyping, consider:

\[
P_1 = a ! l(5), a ! l(1), Q_1 \quad P_2 = b ! l(Large \text{datum}), b ! l(y_2), Q_2
\]

First \( P_2 \) sends a large datum on channel \( b \); then after receiving it, \( P_1 \) sends 5 to \( P_2 \). We note that \( P_1 \)'s output is blocked until this large datum is received. Since the value replacing \( y_1 \) does not influence the subsequent output at \( a \), process \( P_1 \) can be optimised by sending the small datum "5" first, so that once the large datum is put in the queue at \( ba \), process \( P_2 \) can immediately receive the small datum. Thus a better version of \( P_1 \) is \( P_1' \) defined by:

\[
P_1' = a ! l(5), a ! l(1), Q_1
\]

Asynchronous subtyping specifies safe permutations of actions, which we can refine a local protocol to maximise asynchrony without violating session safety.

To define asynchronous subtyping the notion of asynchronous context of types introduced in [27] is handy. An asynchronous context is a sequence of branches containing holes that we index in order to distinguish them.

**Definition 4.3** (Asynchronous context).

\[
\mathcal{A} := [\ ] \mid [a] \mid [a] \in \mathcal{N} \mathcal{A}
\]

We write \( \mathcal{A} \in \mathcal{N} \mathcal{A} \) to denote a context with holes indexed by elements of \( N \) and \( \mathcal{A} \in \mathcal{T} \mathcal{A} \in \mathcal{N} \mathcal{A} \) to denote the same context when the hole \( [a] \) has been filled with \( T \).

**Example 4.4.** Let \( N = \{1, 2\} \) and

\[
T_1 = ! m(S_m) \cdot T_m \parallel ! p(S_p) \cdot T_p \quad T_2 = ! m(S_m') \cdot T_m' \parallel ! p(S_p') \cdot T_p' \parallel ! q(S_q) \cdot T_q
\]

Assume \( \mathcal{A} = ?r(S_r), [1] \parallel [\ ] \parallel [2] \parallel [\ ] \parallel [\ ] \); then

\[
[\mathcal{A}]^2 = [\mathcal{A}]^2 = ?r(S_r) ! m(S_m) \cdot T_m \parallel ! p(S_p) \cdot T_p \parallel ! q(S_q) \cdot T_q
\]

We write \& \in T if \( T = \otimes_{i \in I} ! l_i(S_i), T_i \) or \( T = \bigcup_{i \in I} ! l_i(S_i), T_i \) and \& \in T for all \( i \in I \) (i.e., all selections in \( T \) contain some branching
Figure 1. An application of rule [SUB-PERM-ASYNC], where \( T_m = \tau(S_r).T_r \& ?s(S_u).T_u \) and \( T_p = \tau(S_r).T_r' \& ?s(S_u).T_u' \) and we assume \( S_r' \leq S_r \).

\[
\begin{align*}
S_m' \leq S_m & \quad S_n' \leq S_n \quad S_p' \leq S_p \quad S_p' \leq S_p \quad T_m \leq ?r(S_r).T_r \& ?s(S_u).T_u \quad T_p \leq ?r(S_r).T_r' \& ?s(S_u).T_u' \\
\langle \text{!m}(S_m), T_m \rangle \oplus \langle \text{!p}(S_p), T_p \rangle \leq \langle \text{!r}(S_r), \text{!m}(S_m), T_m \oplus \text{!p}(S_p), T_p \rangle \oplus \langle \text{!q}(S_q), T_q \rangle \rangle & \quad \langle \text{?m}(S_m), T_m \oplus \text{!p}(S_p), T_p \rangle \rangle
\end{align*}
\]

Example 4.5. (Asynchronous subtyping)

1. We show \( T_1 \leq S_1 \), where \( T_1 = \mu t.\ !l(T') \cdot ?l'(S') \cdot t \) and \( S_1 = \mu t.\ !l(T') \cdot ?l'(S') \cdot t \).

2. We show \( T_2 \leq S_2 \), where \( T_2 = \mu l(T') \cdot ?l(S') \cdot t \) and \( T_2 \leq S_2 \).

3. Choosing \( \alpha' \) as in Example 4.4 gives an application of rule [SUB-PERM-ASYNC]. The rightmost premises are

\[
\begin{align*}
T_m \leq \alpha'[T_m]' & \quad T_m \leq \alpha'[T_p]' \\
T_p \leq \alpha'[T_p]' & \quad T_p \leq \alpha'[T_p]'
\end{align*}
\]

which hold by rule [SUB-BRA]. The left-hand-side of the conclusion is a selection between the outputs \( !m(T_m) \) and \( !p(S_p) \).

Theorem 4.6. The relation \( \leq_a \) is transitive.

4.4 Typing asynchronous processes

Since processes now include queues, we need queue types defined by:

\[
\tau ::= \varepsilon \mid \iota(S) \mid \tau \cdot \tau
\]

where we assume associativity of \( \cdot \) and \( \tau \cdot \varepsilon = \varepsilon \cdot \tau = \tau \).

4.5 Soundness of asynchronous subtyping

Reduction of session environments is standard in session calculi to take into account how communications modify the types of free channels and queues. In the synchronous case only restricted channels can exchange messages. We could reduce only restricted channels also in the asynchronous case, but this would make heavier the reduction rules. Table 14 defines the reduction rules for asynchronous processes and queues.
between session environments. Rule [TR-IN] simply corresponds to the deque of a message. Rule [TR-OUT] takes into account the asynchrony of subtyping: we need to dig a selection under a context since a typable process might contain a selection followed by several branches. The following example illustrates rule [TR-OUT].

**Example 4.7.** Let \( T_1, T_2 \) be defined as in Example 4.4 and assume that there is \( S \) such that \( S_m, S_m' \subseteq S \) and that there is no \( S' \) such that \( S_P, S_m' \subseteq S' \). By rule [TR-OUT], only branch \( m \) can be triggered to output, since branch \( q \) is only at hole 2, and the above assumption forbids to choose branch \( p \).

\[
a : a' [T_1] [T_2]_l a b : \tau \Rightarrow a : r(S_P), T_m \& ?(S_l). T_m' a b : \tau \cdot m(S_i).
\]

In order to get subject reduction we cannot start from an arbitrary session environment. For example the process

\[
\alpha !(x). \alpha !'(x + 1) | ba \triangleright l(true)
\]

can be typed with the session environment

\[
\{a : ?(int), !'(int). end, ba : l(boolean)\}
\]

but it reduces to \( \alpha !'(x + 1) \) which cannot be typed. As in 4.21 we restrict to balanced session environments according to the following definition.

**Definition 4.8 (Balanced session environments).** A session environment \( \Delta \) is balanced if:

1. \( a : T, ba : \tau \Rightarrow \) imply that \( T - \tau \) is defined; and
2. \( a : T, ba : \tau, b : T', ab : \tau' \Rightarrow \) imply that \( T' - \tau' \).

Notice that \( ?(int), !'(int). end \) is undefined. It is easy to verify that reduction preserves balancing of session environments.

**Lemma 4.9.** If \( \Delta \Rightarrow \Delta' \) and \( \Delta \) is balanced, then \( \Delta' \) is balanced.

We can now state subject reduction:

**Theorem 4.10 (Subject reduction for asynchronous processes).** If \( \Gamma \vdash P : \Delta \) where \( \Delta \) is balanced and \( P \to^*_\Delta Q \), then there is \( \Delta' \) such that \( \Delta \Rightarrow \Delta' \) and \( \Gamma \vdash^*_Q \). Also the assurance that well-typed processes cannot go wrong requires balanced session environments.

**Corollary 4.11.** If \( \Gamma \vdash^*_P : \Delta \) and \( \Delta \) is balanced, then \( P \to^*_\Delta \) is error.

Lastly we get:

**Theorem 4.12.** The asynchronous subtyping relation \( \leq_a \) is sound for the asynchronous calculus.

### 5. Completeness for Asynchronous Subtyping

We start this section by remarking that the synchronous subtyping is incomplete for the synchronous calculus. In fact the synchronous subtyping is properly included in the asynchrony subtyping. For example \( T \not\leq_a S \) by rule [N-SELBRA-SYN] but \( T \leq_S S \), where \( T = !l(T'), !l(S') \) and \( S = !l(S'), !l(T') \). Then soundness of the asynchrony subtyping implies incompleteness of the synchronous subtyping.

We show completeness for asynchrony subtyping following the three steps described in 5.3. In the third step we need to add two queues for exchanging messages. For the proof of Theorem 5.4 the proofs are more delicate than in the synchronous case since the asynchrony subtyping makes the shapes of types less structured. The first difficulty is to define the negation \( S \leq_a \) inductively and prove that it implies the non-derivability of \( \leq_a \). The second difficulty is to catch the error states arising after an unbounded number of message enqueues, since rule [SUB-TERM] can exchange a selection with an unbounded number of branchings.

**Characteristic asynchronous processes**

The definition of characteristic processes for the asynchronous case differs from that of the synchronous one only for outputs, since the creation of a new pair of restricted channels requires also the creation of the corresponding queues.

**Definition 5.1 (Characteristic asynchronous processes).** The characteristic process offering communication \( T \) on identifier \( u \) for the asynchronous calculus, denoted by \( P(u, T) \), is defined as in Definition 3.2 but for the case of \( P(u, l, S, T) \) which is now:

\[
P(u, l, S, T) \triangleq (\nu b. \alpha l(b). P(u, T) \cdot \{P(b), \nu l\} | ba \triangleright \emptyset | ab \triangleright \emptyset)
\]

For example if \( T = !l_1(\text{end}) \cdot !l_2(\text{end}). \text{end} \), then

\[
P(a, T) = P(a, l_1(\text{end}), \text{end}) \cdot P(a, l_2, l_3(\text{end}), \text{end}, \text{end})
\]

**Rules for negation of the asynchronous subtyping**

To define the negation of asynchronous subtyping we need the projection of a type context on a hole with a given index. This projection chooses...
at each branching the right input to reach the hole, if any. More formally:

\[

t^*|_n = \begin{cases} 
\{t\}^n & \text{if } n = n', \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

\[
(\&(i \in \text{II}(S_i), \alpha(n))|_n = \begin{cases} 
\text{undefined} & \text{if } |n| \text{ occurs in } \alpha(n), \\
\{t\}^n & \text{otherwise.}
\end{cases}
\]

For example, if \( \text{II}(S_i) = \emptyset \) and \( |n| = 1 \), then \( \text{II}(S_i)|_n = \emptyset \).

The negation rules of asynchronous subtyping are the rules of Table 7 excluding rule \([\text{-n-label-sync}]\) plus the rules of Table 5.

Rule \([\text{n-label-sync}]\) deals with the case that the selection cannot find a matching label inside the \(n_0\)-th hole. Rule \([\text{n-exch-sync}]\) considers a mismatch between carried types inside the \(n_0\)-th hole. Rule \([\text{n-cont-sync}]\) considers a mismatch between continuation types, inside the \(n_0\)-th hole, but preceded by the inputs needed to reach that hole. These three rules correspond to the rules \([\text{n-label sel}], \text[n-exch sel] \) and \([\text{n-cont sel}]\) of Table 7, respectively, when the context \( \alpha' \) is just one hole. Rule \([\text{n-label-sync}]\) assures that a type without branchings cannot be a subtype of a branching type. We write \( T \not\models S \) if \( T \not\models S \) is generated by the rules in Table 7 and Table 5 excluding rule \([\text{n-label-sync}]\).

For example, by rule \([\text{r-cont-sync}]\),

\[
!m(S_m) \vdash (\exists \text{rule}(S_i). T_i & \& \text{r}(S_i). \text{end}) \not\models \text{r}(S_i) !m(S_m) ; T_i & \& \text{r}(S_i) . !p(S_p) . T_p
\]

since \( \text{r}(S_i) . T_i & \& \text{r}(S_i) . \text{end} \not\models \text{r}(S_i) . T_i & \& \text{r}(S_i) . !p(S_p) . T_p \).

We can prove that \( \not\models S \) is the negation of the asynchronous subtyping, by observing that the premises of rule \([\text{sub-perm-sync}]\) correspond to the rules of Table 5.

**Lemma 5.3.** \( S \models T \) is not derivable if and only if \( S \not\models T \) is derivable.

Completeness can now be shown:

**Theorem 5.4 (Completeness for asynchronous subtyping).** The asynchronous subtyping relation \( \models \) is complete for the asynchronous calculus.

**Proof.** We prove that \( T \not\models S \) implies

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

where \( P(a,T) \), \( P(b,\overline{S}) \) are characteristic asynchronous processes. The proof is by induction on \( T \) and \( S \) and by cases on the definition of \( \not\models S \). We only consider some interesting cases.

1. For rule \([\text{n-label-sync}]\) we get

\[
T = \bigoplus_{i \in \text{II}(T')} T_i, \quad S = \alpha \bigoplus_{j \in \text{II}(S')} \bigcup_{n \in \text{N}} S_j^n
\]

and there are \( i_0 \in I, n_0 \in \text{N} \) such that for all \( j \in \text{II}(T) \) we get \( T_j^n \neq i_0 \). We show by induction on \( \alpha \) that \( T \not\models S \) implies

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

for an arbitrary queue \( h \).

(a) If \( \alpha \) is just one hole, then \( S = \bigoplus_{i \in \text{II}(T')} \bigcup_{n \in \text{N}} S_j^n \). We get

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

\[
\rightarrow 2 (vab)(P(a,\lambda_i, T_i, \lambda_j, T_j) | \bigcup_{j \in \text{II}(T')} \bigcup_{n \in \text{N}} S_j^n) \vdash \text{error}
\]

(b) If \( \& \in \alpha \), let \( \alpha = \&_{k \in \text{K}} \bigcup_{n \in \text{N}_k} S_k^n \), where \( \bigcup_{k \in \text{K}} \text{N}_k = \text{N} \). Then

\[
S = \bigoplus_{k \in \text{K}} \bigcup_{n \in \text{N}_k} S_k^n \quad \text{and by cases on the definition of subtyping}
\]

\[
\text{error}
\]

We get

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

\[
\rightarrow 2 (vab)(P(a,T) | P(b,\overline{S})) \vdash \text{error}
\]

2. For rule \([\text{n-exch-sync}]\) we get \( \& \not\models T \) and \( S = \bigoplus_{i \in \text{II}(T')} T_i \).

Then \( S = \bigoplus_{i \in \text{II}(T')} T_i \).

We get

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

by rule \([\text{err-mism-sync}]\).

(b) If \& \in \alpha \), let \( \alpha = \&_{k \in \text{K}} \bigcup_{n \in \text{N}_k} S_k^n \), where \( \bigcup_{k \in \text{K}} \text{N}_k = \text{N} \). Then

\[
S = \bigoplus_{k \in \text{K}} \bigcup_{n \in \text{N}_k} S_k^n \quad \text{and by cases on the definition of subtyping}
\]

\[
\text{error}
\]

We get

\[
(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

\[
\rightarrow 2 (vab)(P(a,T) | P(b,\overline{S})) \vdash \text{error}
\]

Then by \([\text{err-context}]\),

\[
C[(vab)(P(a,T) | P(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

6. **Denotational Preciseness**

In \( \lambda \)-calculus types are usually interpreted as subsets of the domains of \( \lambda \)-models \([2,13]\). Denotational preciseness of subtyping is then:

\[
T \subseteq S \text{ if and only if } \llbracket T \rrbracket \subseteq \llbracket S \rrbracket
\]

using \( \llbracket \cdot \rrbracket \) to denote type interpretation.

In the present context let us interpret a session type \( T \) as the set of processes with only one free channel typed by \( T \), i.e.

\[
\llbracket T \rrbracket = \{ P | \vdash_T P \Rightarrow a : T \}
\]

where \( * \in \{s,a\} \). We can then show that both the synchronous and the asynchronous subtypings are denotationally precise. Rule \([\text{t-sub}]\) gives the denotational soundness. Denotational completeness follows from the following key property of characteristic processes:

\[
\vdash_T P_s(a,T) \Rightarrow a : S \text{ if and only if } \llbracket a : S \rrbracket \subseteq \llbracket a : S \rrbracket
\]

where \( P_s(a,T) \) is the synchronous characteristic process if \( * = s \) and the asynchronous characteristic process if \( * = a \). If we could derive \( \vdash_T P_s(a,T) \Rightarrow a : S \) with \( T \not\models S \), then the processes

\[
(vab)(P_s(a,T) | P_s(b,\overline{S})) \quad \text{and}
\]

\[
(vab)(P_s(a,T) | P_s(b,\overline{S})) \vdash sa \Rightarrow \vdash sa \text{ error}
\]

could be typed. Theorems \([3,5]\) and \([4,5]\) show that these processes reduce to error, and this contradicts soundness of the type systems. We get the desired property, which implies denotational completeness, since if \( T \not\models S \), then \( P_s(a,T) \in \llbracket T \rrbracket \), but \( P_s(a,T) \not\in \llbracket S \rrbracket \).
Theorem 6.1 (Denotational preciseness). The synchronous and the asynchronous subtyping relations are denotationally precise for the synchronous calculus and the asynchronous calculus, respectively.

7. Related Work

Preciseness To the best of our knowledge operational preciseness was first defined in [3] for a call-by-value λ-calculus with recursive functions, pairs and sums. In that paper the authors show that the iso-recursive subtyping induced by the Amber rules [5] is incomplete. They propose a new iso-recursive subtyping which they prove to be precise.

Operational and denotational preciseness are shown in [9] for the concurrent λ-calculus with intersection and union types introduced in [10]. In that paper divergence plays the role of reduction to error.

Preciseness in concurrency is more useful and challenging than in the functional setting, since there are many interesting choices for the syntax, semantics, type errors of the calculi and for the typing systems. A similar situation appears in the study of bisimulations where many labelled transition relations can be defined. It is now common that researchers justify the correctness of labelled transition systems by proving that the bisimulation coincides with the contextual congruence [20][26]. Our claim is that preciseness should become a sanity check for subtypings.

Choices of subtypings The first branching-selection subtyping for the session types was proposed in [12] and used for example in [6][7][31][36]. That subtyping is the opposite of the current synchronous subtyping, since branch is covariant and selection is contravariant in the set of labels. We can establish the preciseness of the subtyping in [12] for the synchronous calculus if we reverse the ordering both in the preciseness definition and in the extension of subtyping to session environments. We have chosen the subtyping used in [4][8][28][29] since we claim it fits better with the preciseness definition. In fact the approach of [12] corresponds to safe substitutability of channels, while the approach of [29] corresponds to safe substitutability of processes. The subtyping of [4][8][28][29] differs from ours since the types of exchanged values are invariant.

Other completeness results Subtyping of recursive types requires algorithms for checking subtype relations, as discussed in [52] Chapter 21. These algorithms need to be proved sound and complete with respect to the definition of the corresponding subtyping, as done for example in [7][12][33]. Algorithms for checking the synchronous and asynchronous subtypings of the present paper can be easily designed.

Several works on subtyping formulate the errors using typed reductions or type environments (e.g. [17][33]), and they prove soundness with respect to the typed reductions and their erasure theorems. In contrast with these approaches, our error definitions in Tables 4 and 12 do not rely on any type-case construct or explicit type information, but are defined syntactically over untyped terms. Note that once the calculus is annotated by type information or equipped with type case, completeness becomes trivial, since any two processes of incomparable types can be operationally distinguished.

Semantic subtyping In the semantic subtyping approach each type is interpreted as the set of values having that type and subtyping is subset inclusion between type interpretations [11]. This gives a precise subtyping as soon as the calculus allows to distinguish operationally values of different types. Semantic subtyping has been studied in [4] for a π-calculus with a pattern input and in [7] for a session calculus with internal and external choices and typed input. Types are built using a rich set of type constructors including union, intersection and negation: they extend IO-types in [6] and session types in [7]. Semantic subtyping is precise for the calculi of [6][7][11], thanks to the type case constructor in [11], and to the blocking of inputs for values of “wrong” types in [6][11].

Subtyping of Mostrous Note that our subtyping relation differs from that defined in [27] only for the premises & ∈ α and & ∈ T, in rule [SUB-PERM-ASYNC]. As a consequence T is a subtype of S when T = µ.t.(T.t) and S = t.µ.(T.t′), where (T.t′) (see [27] p. 116)). This subtyping is not sound in our system: intuitively T accumulates infinite orphan messages in a queue, while S ensures that the messages are eventually received. The subtyping relation in [27] unexpectedly allows an unsound process (typed by T) to act as if it were a sound process (typed by S). Let

\[ Q = \text{def} Y(x) = b!l(x).b?l(y).Y(x) \text{ in } (\text{vec}d')(Y(c)). \]

Then we can derive C[a : S] ▷ δ. Let

\[ P = \text{def} Z(z) = a!l(z).Z(z) \text{ in } (\text{vdd})(Z(d')). \]

Then P ▷ [a : T]. We get

\[ C[P] →_S (\text{vad})(\text{vec}d')(P) \mid Q \mid ab ▷ \emptyset \mid b?l(c) \rightarrow_\text{error} \]

by rule [ERR-ORPH-MESS-ASYNC], since a & φ(P) & Q & ab ▷ \emptyset and \fpw(P) & Q & ab ▷ \emptyset = \emptyset.

The subtyping of [27] is sound for the session calculus defined there, which does not consider orphan messages as errors. However, the subtyping of [27] is not complete, an example being µ.λ.(T.t) + µ.λ.(S.t). There is no context C which is safe for all processes with one channel typed by µ.λ.(S.t) and no process P with one channel typed by µ.λ.(T.t) such that C[P] deadlocks.

8. Conclusion

This paper gives, as far as we know, the first formulation and proof techniques for the preciseness of subtyping in mobile processes. We consider the synchronous and asynchronous session calculi to investigate the preciseness of the existing subtypings. While the well-known branching-selection subtyping [4][8][12] is precise for the synchronous calculus, the subtyping in [27] turns out to be not sound for the asynchronous calculi. We propose a simplification of previous asynchronous subtypings [28][29] and prove its preciseness. As a matter of fact only soundness is a consequence of subject reduction, while completeness can fail also when subject reduction holds.

Our calculus lacks the session initialisation as well as the communication of expressions which are present in the original calculus [21]. These extensions are straightforward and we can obtain the same preciseness results. First, for the extension to session initialisation, we just need to add a negation of the subtyping relation over the shared channel type ⟨T,T⟩ in [21], and define its corresponding session initialisation process as a characteristic process for a shared channel type. The definitions of deadlock/error processes remain the same. Second, for the extension to expressions (e.g. succ(n)) and to ground types (e.g. nat), we require contravariance of input and covariance of output for ground types and we add constructors distinguishing values of different ground types (e.g. succ can be applied to a value of type nat but not to a value of type bool). We then use these constructors in building characteristic processes following [3].

The formulation of preciseness along with the proof methods and techniques could be useful to examine other subtypings and calculi. Our future work includes the applications to higher-order processes [27][28], polymorphic types [15], fair subtypings [31] and contract subtyping [11]. We plan to use the characteristic processes in typecheckers for session types. More precisely the error messages can show processes of given types when type checking
fails. One interesting problem is to find the necessary and sufficient conditions to obtain completeness of the generic subtyping\(^\ddagger\). Such a characterisation would give preciseness for the many type systems which are instances of\(^\ddagger\).

The preciseness result for the synchronous calculus in §2 and §3 shows a rigorousness of the branching-selection subtyping, which is implemented (as a default) in most of session-based programming languages and tools\(^\ddagger\)[1][4][8][22][35] for enlarging typability. For the asynchronous calculus, preciseness is more debatable since it depends on the choice of type safety properties, see §4 and §5. But in this case preciseness plays a more important role since a programmer can adjust a subtyping relation to loosen or tighten typings with respect to the type safety properties which she wishes to guarantee. Once preciseness has been proved, she can be sure that her safety specifications and the subtyping have an exact match with respect to both static and dynamic semantics.

Acknowledgments

This work has been partially sponsored by EPSRC EP/K034413/1 and EP/K011715/1, ICT COST Action IC1201 BETTY, MIUR PRIN Project CINA Prot. 2010LHT4K4 and Torino University/Compagnia San Paolo Project SALT. We thank Kohei Honda and Luca Padovani for their suggestions and discussions. We also gratefully thank the anonymous referees for their accurate and enlightening remarks, that strongly improved both the presentation and the technical development.

References


