On the Preciseness of Subtyping in Session Types

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 $T \leqslant S$ 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, π-calculus, Subtyping, Completeness and Soundness.

1. Introduction

Subtyping in concurrency Since Milner first introduced the idea of assigning types to channels in the π-calculus [25], 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 π-calculus.

The earliest work is a simple subtyping between input and output capabilities (called IO-subtyping) [33], which has been extended to 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 [23], where the subtyping plays a fundamental rôle to generate a variety of interesting type systems as its instances.

PPDP '14, September 08–10 2014, Canterbury, UK.

ACM 978-1-4503-2947-7/14/09...$15.00.
http://dx.doi.org/10.1145/2643135.2643138

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, 35, 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:

$$!\langle\texttt{string}\rangle.?(\texttt{int}).\{!\mathsf{ok}\langle\texttt{string}\rangle.?(\texttt{date}).\mathsf{end} \oplus !\mathsf{quit}.\mathsf{end}\} \quad (1)$$

The prefix $!\langle\texttt{string}\rangle$ denotes an output of a value of type `string`, whereas $?(\texttt{int})$ denotes an input of a value of type `int`. Instead ok and quit are labels distinguishing different branches.The operator $\oplus$ is an internal choice, meaning the process may choose to either send the label 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

$$?(\texttt{string}).!\langle\texttt{int}\rangle.\{?\mathsf{ok}(\texttt{string}).!\langle\texttt{date}\rangle.\mathsf{end} \,\&\, ?\mathsf{quit}.\mathsf{end}\} \quad (2)$$

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

As $\texttt{nat} \leqslant \texttt{real}$ 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):

$$!\langle\texttt{string}\rangle.?(\texttt{int}).!\mathsf{ok}\langle\texttt{string}\rangle.?(\texttt{date}).\mathsf{end} \quad (3)$$

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

$$?(\texttt{string}).!\langle\texttt{int}\rangle.\{?\mathsf{ok}(\texttt{string}).!\langle\texttt{date}\rangle.\mathsf{end} \,\&\, ?\mathsf{quit}.\mathsf{end} \,\&\, ?\mathsf{later}.\mathsf{end}\}$$

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 $P \to^* P'$ (where P' is possibly the `error` process) as well as typing judgements of the form $P \triangleright \{a : T\}$, assuring 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 $C[a : T] \triangleright \emptyset$ means that filling the hole of C with any process P typed by $a : T$ produces a well-typed closed process (formally $C[a : T] \triangleright \emptyset \iff X : T \vdash C[X\langle a \rangle] \triangleright \emptyset$, 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 λ-calculus with sums and product types given in [3].

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

$$T \leqslant S \iff \begin{pmatrix}\text{there do not exist } C \text{ and } P \text{ such that:} \\ C[a : S] \triangleright \emptyset \text{ and } P \triangleright \{a : T\} \text{ and } C[P] \to^* \texttt{error}\end{pmatrix}$$

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 $T \leqslant S$. 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 π-calculus Then IO-subtyping [33] 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 π-calculus. The branching and selection subtyping is instead precise for the π-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, 36] and the asynchronous [27–29] session calculi.

Two preciseness results Session types have sufficiently rich structure to assure completeness, hence if $T \not\leqslant S$, 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, 22, 35].

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 $T \leqslant S$ 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:

$$\begin{aligned} T_a &= !\langle\texttt{int}\rangle.!\langle\texttt{char}\rangle.?(\texttt{string}).?(\texttt{nat}).\texttt{end} \\ T_b &= !\langle\texttt{string}\rangle.!\langle\texttt{nat}\rangle.?(\texttt{int}).?(\texttt{char}).\texttt{end} \end{aligned}$$

since the process typed by T_a can put two messages typed by `int` and `char` in one queue and the process typed by T_b can put two messages typed by `string` and `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:

$$T_a \leqslant ?(\texttt{string}).?(\texttt{nat}).!\langle\texttt{int}\rangle.!\langle\texttt{char}\rangle.\texttt{end}$$

so that the process typed by T_a 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 "messaging overlapping" in the parallel programming community [30, §6]. 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 π-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 π-calculus.

Contributions and outline As far as we are aware, this is the first time that completeness of subtypings, which is solely based on (untyped) 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 gives us denotational preciseness of both synchronous and asynchronous subtypings, as shown in § 6. 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 subtypings between session channels, we eliminate the session initiations (shared channels) and the expressions. The obtained calculus is similar to that presented in [36]. We then define the typing system and prove soundness of subtyping as defined in Definition 1.1.

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; *identifiers* (names and variables), ranged over by $u,u'\ldots$; *labels*, ranged over by $l,l',\ldots$; *process variables*, ranged over by $X,Y,\ldots$; and *processes*, ranged over by $P,Q\ldots$.

Session communications are performed between an output process $u!l\langle u'\rangle.P$ and an input process $\sum_{i\in I} u?l_i(x_i).P_i$ (the l_i are pairwise distinct), where the former sends a channel choosing one of the branches offered by the latter. In $\sum_{i\in I} u?l_i(x_i).P_i$ and $u!l\langle u'\rangle.P$ 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 rôle of the choice. The process $\texttt{def}\ D\ \texttt{in}\ P$ is a recursive agent and $X\langle\tilde{u}\rangle$ is a recursive variable. The process $(\nu ab)P$ is a restriction which binds two channels, a and b in P, making them *co-channels*, i.e. allowing them to communicate (see rule [R-COM-SYNC] in Table 2). This double-restriction is commonly used in the recent literature of session types, e.g. [12, 36]. We often omit $\mathbf{0}$ 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 $\mathsf{fpv}(P)$/$\mathsf{fn}(P)$ we denote the set of *free process variables*/*free names* in P. By $\mathsf{sn}(P)$ we denote the set of *free subject names* in P, defined by $\mathsf{sn}(u!l\langle u'\rangle.P)=\mathsf{fn}(u)\cup\mathsf{sn}(P)$ and $\mathsf{sn}(\sum_{i\in I}u?l_i(x_i).P_i)=\mathsf{fn}(u)\cup\bigcup_{i\in I}\mathsf{sn}(P_i)$ and as expected in the other cases.

$P ::=$		**Process**
	$\mathbf{0}$	(nil)
$\mid$	$X\langle\tilde{u}\rangle$	(variable)
$\mid$	$\sum_{i\in I}u?l_i(x_i).P_i$	(input)
$\mid$	$u!l\langle u'\rangle.P$	(output)
$\mid$	$P\oplus P$	(choice)
$\mid$	$P\mid P$	(parallel)
$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	(definition)
$\mid$	$(\nu ab)P$	(restriction)
$\mid$	$\texttt{error}$	(error)

$D ::=$	**Declaration**
$X(\tilde{x})=P$	

$u ::=$		**Identifiers**
	a	(name)
$\mid$	x	(variable)

Table 1. Syntax of synchronous processes.

$$\text{[R-COM-SYNC]}\quad \frac{k\in I}{(\nu ab)(a!l_k\langle c\rangle.P\mid\sum_{i\in I}b?l_i(x_i).Q_i)\to(\nu ab)(P\mid Q_k\{c/x_k\})}$$

$$\text{[R-DEF]}\quad \texttt{def}\ X(\tilde{x})=P\ \texttt{in}\ (X\langle\tilde{a}\rangle\mid Q)\to\texttt{def}\ X(\tilde{x})=P\ \texttt{in}\ (P\{\tilde{a}/\tilde{x}\}\mid Q)$$

$$\text{[R-CHOICE]}\quad P\oplus Q\to P \qquad \text{[R-CONTEXT]}\quad \frac{P\to P'}{C[P]\to C[P']} \qquad \text{[R-STRUCT]}\quad \frac{P\equiv P' \quad P'\to Q' \quad Q'\equiv Q}{P\to Q}$$

Table 2. Reduction of synchronous processes.

$$\text{[S-PAR 1]}\quad \mathbf{0}\mid P\equiv P \qquad \text{[S-PAR 2]}\quad P\mid Q\equiv Q\mid P \qquad \text{[S-PAR 3]}\quad P\mid(Q\mid R)\equiv(P\mid Q)\mid R$$

$$\text{[S-CH 1]}\quad P\oplus Q\equiv Q\oplus P \qquad \text{[S-CH 2]}\quad (P\oplus Q)\oplus R\equiv Q\oplus(P\oplus R)$$

$$\text{[S-RES 1]}\quad \frac{a,b\notin\mathsf{fn}(Q)}{(\nu ab)P\mid Q\equiv(\nu ab)(P\mid Q)} \qquad \text{[S-RES 2]}\quad \frac{\{a,b\}\cap\{c,d\}=\emptyset}{(\nu ab)(\nu cd)P\equiv(\nu cd)(\nu ab)P}$$

$$\text{[S-DEF 1]}\quad \texttt{def}\ D\ \texttt{in}\ \mathbf{0}\equiv\mathbf{0} \qquad \text{[S-DEF 2]}\quad \frac{a,b\notin\mathsf{fn}(D)}{\texttt{def}\ D\ \texttt{in}\ (\nu ab)P\equiv(\nu ab)(\texttt{def}\ D\ \texttt{in}\ P)}$$

$$\text{[S-DEF 3]}\quad \frac{\mathsf{dpv}(D)\cap\mathsf{fpv}(Q)=\emptyset}{(\texttt{def}\ D\ \texttt{in}\ P)\mid Q\equiv\texttt{def}\ D\ \texttt{in}\ (P\mid Q)}$$

$$\text{[S-DEF 4]}\quad \frac{\mathsf{dpv}(D)\cap\mathsf{dpv}(D')=\emptyset}{\texttt{def}\ D\ \texttt{in}\ \texttt{def}\ D'\ \texttt{in}\ P\equiv\texttt{def}\ D'\ \texttt{in}\ \texttt{def}\ D\ \texttt{in}\ P}$$

Table 3. Structural congruence for synchronous processes.

2.2 Operational semantics

Table 2 gives the reduction relation between the synchronous processes which do not contain free channel variables. It uses the following reduction context:

$$C ::= [\,]\ \mid\ C\mid P\ \mid\ (\nu ab)C\ \mid\ \texttt{def}\ D\ \texttt{in}\ C$$

and the structural rules of Table 3. By $\mathsf{dpv}(P)$ we denote the set of *process variables introduced in declarations*, whose definition by induction on processes has only one interesting case:

$$\mathsf{dpv}(X(\tilde{x})=P)=\{X\}\cup\mathsf{dpv}(P).$$

In Table 2, [R-COM-SYNC] is the main communication rule between input and output at two co-channels a and b, where the label l_k is selected and channel c is instantiated into the k-th input branch. Other rules are standard.

We also define error reduction (Table 4), 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-IN-SYNC] gives an error when two

$$\text{[ERR-MISM-SYNC]}\ \frac{\forall i \in I : l \neq l_i}{(\nu ab)(a!l\langle c\rangle.P \mid \sum_{i\in I} b?l_i(x_i).Q_i) \to \texttt{error}} \qquad \text{[ERR-NEW-SYNC]}\ \frac{a \in \mathsf{sn}(P) \quad b \notin \mathsf{fn}(P)}{(\nu ab)P \to \texttt{error}}$$

$$\text{[ERR-CONTEXT]}\ C[\texttt{error}] \to \texttt{error} \qquad \text{[ERR-OUT-OUT-SYNC]}\ (\nu ab)(a!l\langle c\rangle.P \mid b!l'\langle c'\rangle.Q) \to \texttt{error}$$

$$\text{[ERR-IN-IN-SYNC]}\ (\nu ab)(\sum_{i\in I} a?l_i(x_i).P_i \mid \sum_{j\in J} b?l'_j(x'_j).Q_j) \to \texttt{error}$$

Table 4. Error reduction for synchronous processes.

$$\text{[SUB-END]}\ \texttt{end} \leqslant \texttt{end} \qquad \text{[SUB-BRA]}\ \frac{\forall i \in I : S_i \leqslant S'_i \quad T_i \leqslant T'_i}{\&_{i\in I\cup J}?l_i(S_i).T_i \leqslant \&_{i\in I}?l_i(S'_i).T'_i}$$

$$\text{[SUB-SEL]}\ \frac{\forall i \in I : S'_i \leqslant S_i \quad T_i \leqslant T'_i}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \leqslant \bigoplus_{i\in I\cup J} !l_i\langle S'_i\rangle.T'_i}$$

Table 5. Subtyping rules for synchronous types.

co-channels are both subjects of inputs. We denote by $\to_{\mathsf{s}}$ the reduction relation for the synchronous processes, generated by the rules in Tables 2 and 4, and by $\to_{\mathsf{s}}^*$ the reflexive and transitive closure of $\to_{\mathsf{s}}$.

2.3 Typing synchronous processes

The syntax of *synchronous session types*, ranged over by T and S, is:

$$T,S \quad ::= \quad \&_{i\in I}?l_i(S_i).T_i \mid \oplus_{i\in I} !l_i\langle S_i\rangle.T_i \mid \mathbf{t} \mid \mu\mathbf{t}.T \mid \texttt{end}$$

The *branching type* $\&_{i\in I}?l_i(S_i).T_i$ 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* $\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i$ 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 $\oplus$ and labels when there is only one branch.
We use $\mathbf{t}$ to range over type variables. The type $\mu\mathbf{t}.T$ is a *recursive type*. We take an equi-recursive view of types, not distinguishing between a type $\mu\mathbf{t}.T$ and its unfolding $T\{\mu\mathbf{t}.T/\mathbf{t}\}$. We assume that the recursive types are guarded, i.e. $\mu\mathbf{t}.\mathbf{t}$ is not a type. The type $\texttt{end}$ represents the termination of a session and it is often omitted. In the examples we use infix notation for & and $\oplus$ and ground types ($\texttt{int}, \texttt{bool}, \ldots$) for messages. The extension to ground types is easy, as discussed in § 8.

As usual *session duality* [21] plays an important rôle for session types. The function $\overline{T}$, defined below, yields the dual of the session type T.

$$\overline{\&_{i\in I}?l_i(S_i).T_i} = \bigoplus_{i\in I} !l_i\langle S_i\rangle.\overline{T_i} \quad \overline{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i} = \&_{i\in I}?l_i(S_i).\overline{T_i}$$
$$\overline{\mathbf{t}} = \mathbf{t} \qquad \overline{\mu\mathbf{t}.T} = \mu\mathbf{t}.\overline{T} \qquad \overline{\texttt{end}} = \texttt{end}$$

We write $T_1 \bowtie T_2$ if $T_2 = \overline{T_1}$.

Table 5 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-selection subtyping from [4, 8, 27–29]. Rule [SUB-BRA] states that the branching which offers fewer branches is a supertype of the one with more branches; and rule [SUB-SEL] is its dual (see the explanations in § 1). We write $T \leqslant_{\mathsf{s}} S$ if $T \leqslant S$ is derived by the rules in Table 5. Reflexivity of $\leqslant_{\mathsf{s}}$ is immediate and transitivity of $\leqslant_{\mathsf{s}}$ can be shown in the standard way.

$$\text{[T-IDLE]}\ \Gamma \vdash \mathbf{0} \triangleright \emptyset \qquad \text{[T-VAR]}\ \Gamma, X : \langle \tilde{T}\rangle \vdash X\langle \tilde{u}\rangle \triangleright \{\tilde{u} : \tilde{T}\}$$

$$\text{[T-INPUT]}\ \frac{\forall i \in I : \Gamma \vdash P_i \triangleright \Delta, u : T_i, x_i : S_i}{\Gamma \vdash \sum_{i\in I} u?l_i(x_i).P_i \triangleright \Delta, u : \&_{i\in I}?l_i(S_i).T_i}$$

$$\text{[T-OUTPUT]}\ \frac{\Gamma \vdash P \triangleright \Delta, u : T}{\Gamma \vdash u!l\langle u'\rangle.P \triangleright \Delta, u : !l\langle S\rangle.T, u' : S}$$

$$\text{[T-PAR]}\ \frac{\Gamma \vdash P_1 \triangleright \Delta_1 \qquad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \qquad \text{[T-CHOICE]}\ \frac{\Gamma \vdash P_1 \triangleright \Delta \qquad \Gamma \vdash P_2 \triangleright \Delta}{\Gamma \vdash P_1 \oplus P_2 \triangleright \Delta}$$

$$\text{[T-NEW-SYNC]}\ \frac{\Gamma \vdash P \triangleright \Delta, a : T_1, b : T_2 \quad T_1 \bowtie T_2}{\Gamma \vdash (\nu ab)P \triangleright \Delta} \qquad \text{[T-SUB]}\ \frac{\Gamma \vdash P \triangleright \Delta \qquad \Delta \leqslant_{\mathsf{s}} \Delta'}{\Gamma \vdash P \triangleright \Delta'}$$

$$\text{[T-DEF]}\ \frac{\Gamma, X : \langle \tilde{T}\rangle \vdash P \triangleright \{\tilde{x} : \tilde{T}\} \qquad \Gamma, X : \langle \tilde{T}\rangle \vdash Q \triangleright \Delta}{\Gamma \vdash \texttt{def}\ X(\tilde{x}) = P\ \texttt{in}\ Q \triangleright \Delta}$$

Table 6. Typing rules for synchronous processes.

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

$$\Gamma ::= \emptyset \mid \Gamma, X : \langle \tilde{T}\rangle \qquad \Delta ::= \emptyset \mid \Delta, u : T$$

We write Δ_1, Δ_2 for $\Delta_1 \cup \Delta_2$ when $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset$. We say that Δ is *end-only* if $u : T \in \Delta$ implies $T = \texttt{end}$.

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

$$\begin{array}{l} u \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) \text{ implies } \Delta_1(u) \leqslant_{\mathsf{s}} \Delta_2(u) \\ u \in \mathsf{dom}(\Delta_1) \text{ and } u \notin \mathsf{dom}(\Delta_2) \text{ imply } \Delta_1(u) = \texttt{end} \\ u \notin \mathsf{dom}(\Delta_1) \text{ and } u \in \mathsf{dom}(\Delta_2) \text{ imply } \Delta_2(u) = \texttt{end} \end{array}$$

We write $\Delta_1 \approx_{\mathsf{s}} \Delta_2$ if $\Delta_1 \leqslant_{\mathsf{s}} \Delta_2$ and $\Delta_2 \leqslant_{\mathsf{s}} \Delta_1$. It is easy to verify that Δ is end-only iff $\Delta \approx_{\mathsf{s}} \emptyset$.

Table 6 gives the typing rules. They are standard in session calculi, see e.g. [12]. Rule [T-IDLE] is the introduction rule for the nil process. To type an input process, rule [T-INPUT] requires the type S_i of variable x_i 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_i and then continues with T_i for each label l_i. The rule for typing output processes is similar and simpler. In rule [T-PAR], the session environment of $P_1 \mid P_2$ is the disjoint union of the environments Δ_1 and Δ_2 for the two processes, reflecting the linear nature of channels. Contrarily, in rule [T-CHOICE], the two processes share the same session environment, since at most one of them will be executed. Rule [T-NEW-SYNC] is a standard rule for name binding, where we ensure the co-channels have dual types. Rules [T-VAR] and [T-DEF] deal with process calls and definitions, requiring the channel parameters have the types which are assumed in the shared environment. Rule [T-VAR] gives these types to the arguments of the process variable. In [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 used to type the body of the definition. We write $\Gamma \vdash_{\mathsf{s}} P \triangleright \Delta$ 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 $\Gamma \vdash_{\mathsf{s}} P \triangleright \Delta$ and $P \rightarrow_{\mathsf{s}}^{*} Q$, then $\Gamma \vdash_{\mathsf{s}} Q \triangleright \Delta$.*

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

Corollary 2.2. *If $\Gamma \vdash_{\mathsf{s}} P \triangleright \Delta$, then $P \not\rightarrow_{\mathsf{s}}^{*}$ `error`.*

The proof of soundness theorem follows easily.

Theorem 2.3. *The synchronous subtyping relation $\leqslant_{\mathsf{s}}$ 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* $\mathbf{P}(u,T)$ 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 $\not\leqslant_{\mathsf{s}}$).
- **[Step 3]** We prove that if $T \not\leqslant_{\mathsf{s}} S$, then the parallel composition of $\mathbf{P}(a,T)$ and $\mathbf{P}(b,\overline{S})$ in the scope of (νab) reduces to error. Hence choosing $P = \mathbf{P}(a,T)$ and $C = (\nu ab)([\,] \mid \mathbf{P}(b,\overline{S}))$ 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.

Characteristic synchronous processes The characteristic synchronous processes are defined following the structure of types.

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

$$\mathbf{P}(u,T) \stackrel{\text{def}}{=} \begin{cases} \sum_{i\in I} \mathbf{P}_{?}(u,l_i,S_i,T_i) & \text{if } T = \&_{i\in I}?l_i(S_i).T_i \\ \bigoplus_{i\in I} \mathbf{P}_{!}(u,l_i,S_i,T_i) & \text{if } T = \bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \\ \mathtt{def}\ X_{\mathbf{t}}(x) = \mathbf{P}(x,S)\ \mathtt{in}\ X_{\mathbf{t}}\langle u\rangle & \text{if } T = \mu\mathbf{t}.S \\ X_{\mathbf{t}}\langle u\rangle & \text{if } T = \mathbf{t} \\ \mathbf{0} & \text{if } T = \mathtt{end} \end{cases}$$

$$\mathbf{P}_{?}(u,l,S,T) \stackrel{\text{def}}{=} u?l(x).(\mathbf{P}(u,T) \mid \mathbf{P}(x,S))$$

$$\mathbf{P}_{!}(u,l,S,T) \stackrel{\text{def}}{=} (\nu ab)(u!l\langle a\rangle.\mathbf{P}(u,T) \mid \mathbf{P}(b,\overline{S}))$$

A branching type is mapped to the inputs $\mathbf{P}_{?}(u,l_i,S_i,T_i)$ $(i \in I)$, which uses the input channel x in $\mathbf{P}(x,S_i)$. A selection type is mapped to the choice between the outputs $\mathbf{P}_{!}(u,l_i,S_i,T_i)$ $(i \in I)$, where the fresh channel a carried by u will be received by the dual input, which will interact with the process $\mathbf{P}(b,\overline{S}_i)$. 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 `end` is mapped to $\mathbf{0}$.

$$\frac{T \neq \mathtt{end}}{\mathtt{end} \not\leqslant T}\ [\text{N-END R}] \qquad \frac{T \neq \mathtt{end}}{T \not\leqslant \mathtt{end}}\ [\text{N-END L}]$$

$$\&_{i\in I}?l_i(S_i).T_i \not\leqslant \bigoplus_{j\in J} !l'_j\langle S'_j\rangle.T'_j\ [\text{N-BRASEL}] \qquad \bigoplus_{j\in J} !l'_j\langle S'_j\rangle.T'_j \not\leqslant \&_{i\in I}?l_i(S_i).T_i\ [\text{N-SELBRA-SYNC}]$$

$$\frac{\exists j \in J\ \forall i \in I : l_i \neq l'_j}{\&_{i\in I}?l_i(S_i).T_i \not\leqslant \&_{j\in J}?l'_j(S'_j).T'_j}\ [\text{N-LABEL BRA}] \qquad \frac{\exists i \in I\ \forall j \in J : l_i \neq l'_j}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \bigoplus_{j\in J} !l'_j\langle S'_j\rangle.T'_j}\ [\text{N-LABEL SEL}]$$

$$\frac{\exists i \in I\ \exists j \in J : l_i = l'_j \quad S_i \not\leqslant S'_j}{\&_{i\in I}?l_i(S_i).T_i \not\leqslant \&_{j\in J}?l'_j(S'_j).T'_j}\ [\text{N-EXCH BRA}] \qquad \frac{\exists i \in I\ \exists j \in J : l_i = l'_j \quad S'_j \not\leqslant S_i}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \bigoplus_{j\in J} !l'_j\langle S'_j\rangle.T'_j}\ [\text{N-EXCH SEL}]$$

$$\frac{\exists i \in I\ \exists j \in J : l_i = l'_j \quad T_i \not\leqslant T'_j}{\&_{i\in I}?l_i(S_i).T_i \not\leqslant \&_{j\in J}?l'_j(S'_j).T'_j}\ [\text{N-CONT BRA}] \qquad \frac{\exists i \in I\ \exists j \in J : l_i = l'_j \quad T_i \not\leqslant T'_j}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \bigoplus_{j\in J} !l'_j\langle S'_j\rangle.T'_j}\ [\text{N-CONT SEL}]$$

Table 7. Negation of synchronous subtyping rules.

For example if $T = !l_1\langle \mathtt{end}\rangle.\mathtt{end} \oplus !l_2\langle !l_3\langle \mathtt{end}\rangle.\mathtt{end}\rangle.\mathtt{end}$, then

$$\begin{array}{rcl} \mathbf{P}(a,T) & = & \mathbf{P}_{!}(a,l_1,\mathtt{end},\mathtt{end}) \oplus \mathbf{P}_{!}(a,l_2,!l_3\langle \mathtt{end}\rangle.\mathtt{end},\mathtt{end}) \\ & = & (\nu bb')(a!l_1\langle b\rangle.\mathbf{P}(a,\mathtt{end}) \mid \mathbf{P}(b',\mathtt{end})) \oplus \\ & & (\nu cc')(a!l_2\langle c\rangle.\mathbf{P}(a,\mathtt{end}) \mid \mathbf{P}(c',?l_3(\mathtt{end}).\mathtt{end})) \\ & = & (\nu bb')(a!l_1\langle b\rangle.\mathbf{0} \mid \mathbf{0}) \oplus \\ & & (\nu cc')(a!l_2\langle c\rangle.\mathbf{0} \mid c'?l_3(x).(\mathbf{P}(c',\mathtt{end}) \mid \mathbf{P}(x,\mathtt{end}))) \\ & = & (\nu bb')(a!l_1\langle b\rangle.\mathbf{0} \mid \mathbf{0}) \oplus \\ & & (\nu cc')(a!l_2\langle c\rangle.\mathbf{0} \mid c'?l_3(x).(\mathbf{0} \mid \mathbf{0})) \\ & \equiv & (\nu bb')a!l_1\langle b\rangle \oplus (\nu cc')(a!l_2\langle c\rangle \mid c'?l_3(x)) \end{array}$$

We can easily check that characteristic processes are well typed as expected.

Lemma 3.2. $\vdash_{\mathsf{s}} \mathbf{P}(u,T) \triangleright \{u : T\}$.

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 `end` cannot be a super or subtype of a type different from `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-LABEL BRA] and [N-LABEL SEL] represent the cases that the labels do not conform the subtyping rules. Rules [N-EXCH BRA] and [N-EXCH SEL] represent the cases that carried types do not match the subtyping rules. Lastly rules [N-CONT BRA] and [N-CONT SEL] represent the cases that continuations do not match the subtyping rules. Notice that the rules [N-$\star$ BRA] are the negations of rule [SUB-BRA], and the rules [N-$\star$ SEL] are the negations of rule [SUB-SEL], where $\star \in \{\text{LABEL, EXCH, SEL}\}$. We write $T \not\leqslant_{\mathsf{s}} S$ if $T \not\leqslant S$ is generated by the rules in Table 7.

It is easy to verify by induction on types that $\not\leqslant_{\mathsf{s}}$ is the negation of the synchronous subtyping.

Proposition 3.3. *$S \leqslant_{\mathsf{s}} T$ is not derivable if and only if $S \not\leqslant_{\mathsf{s}} T$ 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 $T \not\leqslant_{\mathsf{s}} S$, then

$$(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S})) \rightarrow_{\mathsf{s}}^{*} \mathtt{error}$$

where $\mathbf{P}(a,T)$, $\mathbf{P}(b,\overline{S})$ are characteristic synchronous processes. The proof is by induction on T and S and by cases on the definition of $\not\leqslant_{\mathsf{s}}$.

Theorem 3.4 (Completeness for synchronous subtyping)**.** *The synchronous subtyping relation $\leqslant_{\mathsf{s}}$ is complete for the synchronous calculus.*

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 8 shows the asynchronous session calculus obtained by extending the synchronous calculus of Table 1 with queues. A queue $ab \blacktriangleright h$ 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 $\mathsf{fn}(ab \blacktriangleright h) = \{a,b\} \cup \mathsf{fn}(h)$, $\mathsf{fn}(\varnothing) = \emptyset$, $\mathsf{fn}(l\langle a\rangle) = \{a\}$, and $\mathsf{fn}(h_1 \cdot h_1) = \mathsf{fn}(h_1) \cup \mathsf{fn}(h_2)$.

We use the structural congruence defined by adding the rules in Table 9 to the rules of Table 3. Rule [S-NULL] represents garbage collection of empty queues.

The reduction rules for asynchronous processes in Table 10 are obtained from the reduction rules of synchronous processes given in Table 2 by replacing rule [R-COM-SYNC] with rules [R-SEND-ASYNC] and [R-RECEIVE-ASYNC]. Rule [R-SEND-ASYNC] enqueues messages and rule [R-RECEIVE-ASYNC] dequeues messages. We write $P \rightarrow_{\mathsf{a}} Q$ if $P \rightarrow Q$ is derived by the rules of Table 2 but rule [R-COM-SYNC] 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 [16]):

(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 (νab) since both channels a, b appear in processes and the queues prescribe them to communicate)

$$\sum_{i\in I} a?l_i(x_i).P_i \mid \sum_{j\in J} b?l'_j(x'_j).Q_j \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing$$

$P ::=$		**Process**	h	$::=$		**Queue**
	$\vdots$	from Table 1			$\varnothing$	(empty)
				$\mid$	$l\langle a\rangle$	(message)
$\mid$	$ab \blacktriangleright h$	(queue)		$\mid$	$h \cdot h$	(composition)

Table 8. Syntax of asynchronous processes.

[S-NULL]
$(\nu ab)(ab \blacktriangleright \varnothing \mid ba \blacktriangleright \varnothing) \equiv \mathbf{0}$

[S-QUEUE-EQUIV]
$$\frac{h \equiv h'}{ab \blacktriangleright h \equiv ab \blacktriangleright h'}$$

[S-QUEUE 1] $\varnothing \cdot h \equiv h$ [S-QUEUE 2] $h \cdot \varnothing \equiv h$ [S-QUEUE 3] $h_1 \cdot (h_2 \cdot h_3) \equiv (h_1 \cdot h_2) \cdot h_3$

Table 9. Structural congruence for asynchronous processes.

[R-SEND-ASYNC]
$ab \blacktriangleright h \mid a!l\langle c\rangle.P \rightarrow ab \blacktriangleright h \cdot l\langle c\rangle \mid P$

[R-RECEIVE-ASYNC]
$$\frac{k \in I}{ab \blacktriangleright l_k\langle c\rangle \cdot h \mid \sum_{i\in I} b?l_i(x_i).P_i \rightarrow ab \blacktriangleright h \mid P_k\{c/x_k\}}$$

Table 10. Reduction of asynchronous processes.

and in the second case the process has the shape

$$(\nu ab)(\sum_{i\in I} a?l_i(x_i).P_i \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright h),$$

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 $(\nu ab)(P \mid ba \blacktriangleright h)$ 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 $\varphi(P)$) 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 φ requires some care, as shown for example by the process:

$$P = b?l_0(x).x?l_1(y) \mid c!l_0\langle a\rangle \mid cb \blacktriangleright \varnothing \tag{4}$$

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

$$P \rightarrow_{\mathsf{a}} b?l_0(x).x?l_1(y) \mid cb \blacktriangleright l_0\langle a\rangle \rightarrow_{\mathsf{a}} a?l_1(y) \mid cb \blacktriangleright \varnothing$$

and this last process has an input with subject a. Hence to define $\varphi(P)$, 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 $\varphi(P)$ 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 δ from processes and sets of declarations to sets of identifiers and the mapping γ from queues to sets of channels.

Definition 4.1. (Mapping φ) The mapping γ is defined by induction on queues:

$$\gamma(\varnothing) = \emptyset \qquad \gamma(l\langle a\rangle) = \{a\} \qquad \gamma(h_1 \cdot h_2) = \gamma(h_1) \cup \gamma(h_2)$$

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

$$
\begin{array}{rcl}
\delta(\mathbf{0},\tilde{D}) & = & \emptyset \\
\delta(X\langle u\rangle,\tilde{D}) & = & \begin{cases} \delta(P,\tilde{D})\{u/x\} & \text{if } X\langle x\rangle = P \in \tilde{D} \\ \emptyset & \text{otherwise.} \end{cases} \\
\delta(\sum_{i\in I} u?l_i(x_i).P_i,\tilde{D}) & = & \{u\} \cup \bigcup_{i\in I}(\delta(P_i,\tilde{D}) \setminus \{x_i\}) \\
\delta(u!l\langle u'\rangle.P,\tilde{D}) & = & \{u'\} \cup \delta(P,\tilde{D}) \\
\delta(P_1 \oplus P_2,\tilde{D}) & = & \delta(P_1,\tilde{D}) \cup \delta(P_2,\tilde{D}) \\
\delta(P_1 \mid P_2,\tilde{D}) & = & \delta(P_1,\tilde{D}) \cup \delta(P_2,\tilde{D}) \\
\delta(\mathtt{def}\ D\ \mathtt{in}\ P,\tilde{D}) & = & \delta(P,\tilde{D}\cdot D) \\
\delta((\nu ab)P,\tilde{D}) & = & \delta(P,\tilde{D}) \setminus \{a,b\} \\
\delta(\mathtt{error},\tilde{D}) & = & \emptyset \\
\delta(ab\blacktriangleright h,\tilde{D}) & = & \gamma(h)
\end{array}
$$

Table 11. The mapping δ.

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

Example 4.2. Assume

$$P = \mathtt{def}\ X(x) = a!l\langle x\rangle\ \mathtt{in}\ X\langle c\rangle \mid b?l(y).y?l(z) \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing$$

Then we obtain:

$$
\begin{array}{rcl}
\varphi(P) & = & \delta(P,\emptyset) \\
& = & \delta(X\langle c\rangle \mid b?l(y).y?l(z) \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing, D) \\
& = & \delta(X\langle c\rangle, D) \cup \delta(b?l(y).y?l(z), D) \cup \\
& & \delta(ab\blacktriangleright\varnothing, D) \cup \delta(ba\blacktriangleright\varnothing, D) \\
& = & (\delta(a!l\langle x\rangle, D)\{c/x\}) \cup \{b\} \cup (\delta(y?l(z), D) \setminus \{y\}) \\
& = & (\{x\}\{c/x\}) \cup \{b\} = \{c\} \cup \{b\} = \{c,b\}
\end{array}
$$

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

$$
\begin{array}{l}
\mathtt{def}\ X(x) = a!l\langle x\rangle\ \mathtt{in}\ X\langle c\rangle \mid b?l(y).y?l(z) \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing \\
\to_{\mathsf{a}} \mathtt{def}\ X(x) = a!l\langle x\rangle\ \mathtt{in}\ a!l\langle c\rangle \mid b?l(y).y?l(z) \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing \\
\to_{\mathsf{a}} \mathtt{def}\ X(x) = a!l\langle x\rangle\ \mathtt{in}\ b?l(y).y?l(z) \mid ab\blacktriangleright l\langle c\rangle \mid ba\blacktriangleright\varnothing \\
\to_{\mathsf{a}} \mathtt{def}\ X(x) = a!l\langle x\rangle\ \mathtt{in}\ c?l(z) \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing
\end{array}
$$

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-IN-ASYNC] gives an error when two processes with bounded channels are in deadlock waiting to read from empty queues. Rule [ERR-IN-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 $\mathsf{fpv}(P) = \emptyset$ assures that we consider all needed declarations in computing $\varphi(P)$. We write $P \to_{\mathsf{a}} \mathtt{error}$ if $P \to \mathtt{error}$ can be derived using the rules of Table 12 plus rule [ERR-CONTEXT] of Table 4. The notation $\to_{\mathsf{a}}^*$ 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 5.4): a process $\mathbf{P}(a,T) \mid \mathbf{P}(b,S) \mid ba\blacktriangleright\varnothing \mid ab\blacktriangleright\varnothing$ is typable (in the system defined in § 4.4) if and only if it does not reduce to `error`. The function φ 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 $\mathbf{P}(a, ?l(\mathtt{end})) = a?l(x)$, 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.

$$
\text{[ERR-MISM-ASYNC]}\quad \frac{\forall i \in I : l \neq l_i}{ab\blacktriangleright l\langle a\rangle \cdot h \mid \sum_{i\in I} b?l_i(x_i).P_i \to \mathtt{error}}
$$

$$
\text{[ERR-IN-IN-ASYNC]}\quad \sum_{i\in I} a?l_i(x_i).P_i \mid \sum_{j\in J} b?l'_j(x'_j).Q_j \mid ba\blacktriangleright\varnothing \mid ab\blacktriangleright\varnothing \to \mathtt{error}
$$

$$
\text{[ERR-IN-ASYNC]}\quad \frac{\forall i \in I : b \notin \mathsf{fn}(P_i)}{(\nu ab)(\sum_{i\in I} a?l_i(x_i).P_i \mid ba\blacktriangleright\varnothing \mid ab\blacktriangleright h) \to \mathtt{error}}
$$

$$
\text{[ERR-ORPH-MESS-ASYNC]}\quad \frac{a \notin \varphi(P) \quad \mathsf{fpv}(P) = \emptyset \quad h \neq \varnothing}{(\nu ab)(P \mid ba\blacktriangleright h) \to \mathtt{error}}
$$

Table 12. Error reduction for asynchronous processes.

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:

$$P_1 = a?l(y_1).a!l\langle 5\rangle.Q_1 \quad P_2 = b!l\langle Large\_datum\rangle.b?l(y_2).Q_2$$

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:

$$P'_1 = a!l\langle 5\rangle.a?l(y_1).Q_1$$

Asynchronous subtyping specifies safe permutations of actions, by 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 branchings containing holes that we index in order to distinguish them.

Definition 4.3 (Asynchronous context)**.**

$$\mathscr{A} ::= [\,]^n \mid \&_{i\in I} ?l_i(S_i).\mathscr{A}_i$$

We write $\mathscr{A}[\,]^{n\in N}$ to denote a context with holes indexed by elements of N and $\mathscr{A}[T_n]^{n\in N}$ to denote the same context when the hole $[\,]^n$ has been filled with T_n.

Example 4.4. Let $N = \{1,2\}$ and

$$T_1 = !\mathsf{m}\langle S_m\rangle.T_m \oplus !\mathsf{p}\langle S_p\rangle.T_p,\ T_2 = !\mathsf{m}\langle S'_m\rangle.T'_m \oplus !\mathsf{p}\langle S'_p\rangle.T'_p \oplus !\mathsf{q}\langle S_q\rangle.T_q.$$

Assume $\mathscr{A} = ?\mathsf{r}(S_r).[\,]^1 \ \&\ ?\mathsf{s}(S_s).[\,]^2$, then

$$
\begin{array}{rcl}
\mathscr{A}[T_1]^1[T_2]^2 & = & ?\mathsf{r}(S_r).!\mathsf{m}\langle S_m\rangle.T_m \oplus !\mathsf{p}\langle S_p\rangle.T_p\ \& \\
& & ?\mathsf{s}(S_s).!\mathsf{m}\langle S'_m\rangle.T'_m \oplus !\mathsf{p}\langle S'_p\rangle.T'_p \oplus !\mathsf{q}\langle S_q\rangle.T_q.
\end{array}
$$

We write $\& \in T$ if $T = \&_{i\in I} ?l_i(S_i).T_i$, or $T = \bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i$ and $\& \in T_i$ for all $i \in I$ (i.e. all selections in T contain some branching

$$\frac{S_m^r \leqslant S_m \quad S_m^s \leqslant S_m \quad S_p^r \leqslant S_p \quad S_p^s \leqslant S_p \quad T_m \leqslant ?\mathsf{r}(S_r).T_r \,\&\, ?\mathsf{s}(S_s).T_s \quad T_p \leqslant ?\mathsf{r}(S_r).T_r' \,\&\, ?\mathsf{s}(S_s).T_s'}{(!\mathsf{m}\langle S_m\rangle.T_m) \oplus (!\mathsf{p}\langle S_p\rangle.T_p) \leqslant (?\mathsf{r}(S_r).(!\mathsf{m}\langle S_m^r\rangle.T_r \oplus !\mathsf{p}\langle S_p^r\rangle.T_r' \oplus !\mathsf{q}\langle S_q\rangle.T_q)) \,\&\, (?\mathsf{s}(S_s).(!\mathsf{m}\langle S_m^s\rangle.T_s \oplus !\mathsf{p}\langle S_p^s\rangle.T_s'))}$$

Figure 1. An application of rule [SUB-PERM-ASYNC], where $T_m = ?\mathsf{r}(S_r).T_r \,\&\, ?\mathsf{s}(S_s).T_s \,\&\, ?\mathsf{u}(S_u).T_u$ and $T_p = ?\mathsf{r}(S_r').T_r' \,\&\, ?\mathsf{s}(S_s).T_s'$ and we assume $S_r' \leqslant S_r$.

types), or $T = \mu\mathbf{t}.T$ and $\& \in T$. Similarly, we write $\& \in \mathscr{A}$ if $\mathscr{A}$ is a branching, i.e. it is not a single hole.

We define the asynchronous subtyping relation $T \leqslant_{\mathsf{a}} S$ if $T \leqslant S$ is derived by the rule:

[SUB-PERM-ASYNC]
$$\frac{\forall i \in I\ \forall n \in N: \quad S_i^n \leqslant S_i \quad T_i \leqslant \mathscr{A}[T_i^n]^{n\in N} \quad \& \in \mathscr{A} \quad \& \in T_i}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \leqslant \mathscr{A}[\bigoplus_{i\in I\cup J_n} !l_i\langle S_i^n\rangle.T_i^n]^{n\in N}}$$

together with the rules in Table 5.

Rule [SUB-PERM-ASYNC] allows the asynchronous safe permutation explained above. It postpones a selection after an unbounded but finite number of branchings, and the selections inside these branchings can be bigger according to rule [SUB-SEL] of Table 5.

Example 4.5. (Asynchronous subtyping)

1. We show $T_1 \leqslant_{\mathsf{a}} S_1$, where $T_1 = \mu\mathbf{t}.!l\langle T'\rangle.?l'(S').\mathbf{t}$ and $S_1 = \mu\mathbf{t}.?l'(S').!l\langle T'\rangle.\mathbf{t}$. If we assume $T_1 \leqslant S_1$, we obtain
$$!l\langle T'\rangle.?l'(S').T_1 \leqslant ?l'(S').!l\langle T'\rangle.S_1$$
by rule [SUB-PERM-ASYNC], which is $T_1 \leqslant S_1$ by folding.
2. We show $T_2 \leqslant_{\mathsf{a}} S_2$, where $T_2 = !l\langle T'\rangle.T_1$ and $S = ?l'(S').S_1$ and T_1, S_1 are as in previous example. We assume $T_2 \leqslant S_2$. We get
$$\begin{array}{rcl} !l\langle T'\rangle.!l\langle T'\rangle.?l'(S').T_1 & \leqslant & ?l'(S').!l\langle T'\rangle.!l\langle T'\rangle.T_1 \\ & & \text{by rule [SUB-PERM-ASYNC]} \\ & \leqslant & ?l'(S').!l\langle T'\rangle.?l'(S').S_1 \\ & & \text{by the assumption } T_2 \leqslant S_2 \\ & \leqslant & ?l'(S').?l'(S').!l\langle T'\rangle.S_1 \\ & & \text{by rule [SUB-PERM-ASYNC]} \end{array}$$
which is $T_2 \leqslant S_2$ by folding.
3. Choosing $\mathscr{A}$ as in Example 4.4 Figure 1 gives an application of rule [SUB-PERM-ASYNC]. The rightmost premises are
$$T_m \leqslant \mathscr{A}[T_r]^1[T_s]^2 \text{ and } T_p \leqslant \mathscr{A}[T_r']^1[T_s']^2,$$
which hold by rule [SUB-BRA] . The left-hand-side of the conclusion is a selection between the outputs $!\mathsf{m}\langle S_m\rangle.T_m$ and $!\mathsf{p}\langle S_p\rangle.T_p$. The right-hand-side of the conclusion is the type
$$\mathscr{A}[!\mathsf{m}\langle S_m^r\rangle.T_r \oplus !\mathsf{p}\langle S_p^r\rangle.T_r' \oplus !\mathsf{q}\langle S_q\rangle.T_q]^1[!\mathsf{m}\langle S_m^s\rangle.T_s \oplus !\mathsf{p}\langle S_p^s\rangle.T_s']^2.$$
Notice that selections are moved inside branchings (possibly making smaller the types of the sent channels) and extra selections (in this case $!\mathsf{q}\langle S_q\rangle.T_q$) can be added.

Reflexivity of $\leqslant_{\mathsf{a}}$ is immediate, while the proof of transitivity requires some ingenuity.

Theorem 4.6. *The relation $\leqslant_{\mathsf{a}}$ is transitive.*

4.4 Typing asynchronous processes

Since processes now include queues, we need queue types defined by:
$$\tau \quad ::= \quad \varepsilon \mid l\langle S\rangle \mid \tau\cdot\tau$$
where we assume associativity of $\cdot$ and $\tau\cdot\varepsilon = \varepsilon\cdot\tau = \tau$. We also extend session environments as follows:
$$\Delta \quad ::= \quad \ldots \mid \Delta, ab : \tau$$

[T-NEW-ASYNC]
$$\frac{\Gamma \vdash P \triangleright \Delta, a : T_1, b : T_2, ba : \tau_1, ab : \tau_2 \quad T_1 - \tau_1 \bowtie T_2 - \tau_2}{\Gamma \vdash (\nu ab)P \triangleright \Delta}$$

[T-EMPTY-Q]
$$\Gamma \vdash ba \blacktriangleright \varnothing \triangleright \{ba : \varepsilon\}$$

[T-MESSAGE-Q]
$$\frac{\Gamma \vdash ba \blacktriangleright h \triangleright \Delta, ba : \tau}{\Gamma \vdash ba \blacktriangleright h \cdot l\langle c\rangle \triangleright \Delta, c : S, ba : \tau\cdot l\langle S\rangle}$$

Table 13. Typing rules for asynchronous processes and queues.

The added new element $ab : \tau$ is the type of messages in the queue $ab \blacktriangleright h$.

We denote by $\mathsf{dom}_{\mathsf{q}}(\Delta)$ the set of local queues which occur in Δ. Two session environments Δ_1 and Δ_2 *agree* if
$$\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \mathsf{dom}_{\mathsf{q}}(\Delta_1) \cap \mathsf{dom}_{\mathsf{q}}(\Delta_2) = \emptyset.$$
If Δ_1 and Δ_2 agree, their composition Δ_1, Δ_2 is given by $\Delta_1, \Delta_2 = \Delta_1 \cup \Delta_2$ as in the synchronous case. We also define $\Delta_1 \leqslant_{\mathsf{a}} \Delta_2$ by:

$u \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$ implies $\Delta_1(u) \leqslant_{\mathsf{a}} \Delta_2(u)$ and
$u \in \mathsf{dom}(\Delta_1)$ and $u \notin \mathsf{dom}(\Delta_2)$ imply $\Delta_1(u) = \mathtt{end}$ and
$u \notin \mathsf{dom}(\Delta_1)$ and $u \in \mathsf{dom}(\Delta_2)$ imply $\Delta_2(u) = \mathtt{end}$ and
$\mathsf{dom}_{\mathsf{q}}(\Delta_1) = \mathsf{dom}_{\mathsf{q}}(\Delta_2)$ and
$ab \in \mathsf{dom}_{\mathsf{q}}(\Delta_1)$ implies $\Delta_1(ab) = \Delta_2(ab)$

We write $\Delta_1 \approx_{\mathsf{a}} \Delta_2$ if $\Delta_1 \leqslant_{\mathsf{a}} \Delta_2$ and $\Delta_2 \leqslant_{\mathsf{a}} \Delta_1$.

We need to take into account the interplay between the session type of a channel and the queue type of the queue dequeued by this channel. Following [27, 28] we define the *session remainder* of a session type T and a queue type τ as the session type obtained from T by erasing all branchings that have corresponding selections in τ. Clearly the session remainder is defined only if T and τ agree on labels and on types of exchanged channels. More formally we define:

[RM-EMPTY]
$$T - \varepsilon = T$$

[RM-BRA]
$$\frac{T_k - \tau = T' \quad S_k \leqslant_{\mathsf{a}} S \quad k \in I}{\&_{i\in I} ?l_i(S_i).T_i - l_k\langle S\rangle\cdot\tau = T'}$$

[RM-SEL]
$$\frac{\forall i \in I : T_i - \tau = T_i'}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i - \tau = \bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i'}$$

The typing rules for asynchronous processes are the rules of Table 6 by replacing rule [T-NEW-SYNC] with rule [T-NEW-ASYNC] and $\leqslant_{\mathsf{s}}$ with $\leqslant_{\mathsf{a}}$ in rule [T-SUB] and adding the rules for typing the queues. Table 13 gives all the new rules. In rule [T-NEW-ASYNC] we take into account not only the types of the channels, but also those of the queues, and we require duality between their remainders. Rule [T-EMPTY-Q] types the empty queue. Rule [T-MESSAGE-Q] says how the type of a queue changes when a message is added.

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 [4, 21]. 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

[TR-IN]
$$\frac{S_k \leqslant_{\mathsf{a}} S \qquad k \in I}{b : \&_{i\in I} ?l_i(S_i).T_i, ab : l_k\langle S\rangle \cdot \tau \Rightarrow b : T_k, ab : \tau}$$

[TR-RES]
$$\frac{\Delta_1 \Rightarrow \Delta_1'}{\Delta_1, \Delta_2 \Rightarrow \Delta_1', \Delta_2}$$

[TR-OUT]
$$\frac{\forall n \in N\ \exists i_n \in I_n : l^n_{i_n} = l \wedge S^n_{i_n} \leqslant_{\mathsf{a}} S}{a : \mathscr{A}[\bigoplus_{i\in I_n} !l^n_i\langle S^n_i\rangle.T^n_i]^{n\in N}, ab : \tau \Rightarrow a : \mathscr{A}[T^n_{i_n}]^{n\in N}, ab : \tau \cdot l\langle S\rangle}$$

Table 14. Reduction of asynchronous session environments.

between session environments. Rule [TR-IN] simply corresponds to the dequeue of a message. Rule [TR-OUT] takes into account the asynchronous 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 $T_1, T_2, \mathscr{A}$ be defined as in Example 4.4 and assume that there is S such that $S_m, S'_m \leqslant_{\mathsf{a}} S$ and that there is no S' such that $S_p, S'_p \leqslant_{\mathsf{a}} S'$. 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.

$$a : \mathscr{A}[T_1]^1[T_2]^2, ab : \tau \Rightarrow a : ?\mathsf{r}(S_r).T_m \ \&\ ?\mathsf{s}(S_s).T'_m, ab : \tau \cdot \mathsf{m}\langle S\rangle.$$

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

$$a?l(x).a!l'\langle x+1\rangle \mid ba \blacktriangleright l\langle \mathtt{true}\rangle$$

can be typed with the session environment

$$\{a : ?l(\mathtt{int}).!l'\langle \mathtt{int}\rangle.\mathtt{end}, ba : l\langle \mathtt{bool}\rangle\}$$

but it reduces to $a!l'\langle \mathtt{true}+1\rangle$ 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 Δ is *balanced* if:

1. $a : T, ba : \tau \in \Delta$ imply that $T - \tau$ is defined; and
2. $a : T, ba : \tau, b : T', ab : \tau' \in \Delta$ imply that $T - \tau \bowtie T' - \tau'$.

Notice that $?l(\mathtt{int}).!l'\langle \mathtt{int}\rangle.\mathtt{end} - l\langle \mathtt{bool}\rangle$ is undefined. It is easy to verify that reduction preserves balancing of session environments.

Lemma 4.9. *If $\Delta \Rightarrow \Delta'$ and Δ is balanced, then Δ' is balanced.*

We can now state subject reduction:

Theorem 4.10 (Subject reduction for asynchronous processes)**.** *If $\Gamma \vdash_{\mathsf{a}} P \triangleright \Delta$ where Δ is balanced and $P \to^*_{\mathsf{a}} Q$, then there is Δ' such that $\Delta \Rightarrow^* \Delta'$ and $\Gamma \vdash_{\mathsf{a}} Q \triangleright \Delta'$.*

Also the assurance that well-typed processes cannot go wrong requires balanced session environments.

Corollary 4.11. *If $\Gamma \vdash_{\mathsf{a}} P \triangleright \Delta$ and Δ is balanced, then $P \not\to^*_{\mathsf{a}} \mathtt{error}$.*

Lastly we get:

Theorem 4.12. *The asynchronous subtyping relation $\leqslant_{\mathsf{a}}$ 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 asynchronous calculus. In fact the synchronous subtyping is properly included in the asynchronous subtyping. For example $T \not\leqslant_{\mathsf{s}} S$ by rule [N-SELBRA-SYNC] but $T \leqslant_{\mathsf{a}} S$, where $T = !l\langle T'\rangle.?l'(S')$ and $S = ?l'(S').!l\langle T'\rangle$. Then soundness

[N-LABEL-ASYNC]
$$\frac{\exists i_0 \in I\ \exists n_0 \in N\ \forall j \in J_{n_0} : l^{n_0}_j \neq l_{i_0}}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \mathscr{A}[\bigoplus_{j\in J_n} !l^n_j\langle S^n_j\rangle.T^n_j]^{n\in N}}$$

[N-EXCH-ASYNC]
$$\frac{\exists i_0 \in I\ \exists n_0 \in N\ \exists j_0 \in J_{n_0} : l^{n_0}_{j_0} = l_{i_0} \wedge S^{n_0}_{j_0} \not\leqslant S_{i_0}}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \mathscr{A}[\bigoplus_{j\in J_n} !l^n_j\langle S^n_j\rangle.T^n_j]^{n\in N}}$$

[N-CONT-ASYNC]
$$\frac{\exists i_0 \in I\ \exists n_0 \in N\ \exists j_0 \in J_{n_0} : l^{n_0}_{j_0} = l_{i_0} \wedge T_{i_0} \not\leqslant (\mathscr{A}{\downarrow}n_0)[T^{n_0}_{j_0}]^{n_0}}{\bigoplus_{i\in I} !l_i\langle S_i\rangle.T_i \not\leqslant \mathscr{A}[\bigoplus_{j\in J_n} !l^n_j\langle S^n_j\rangle.T^n_j]^{n\in N}}$$

[N-BRA-ASYNC]
$$\frac{\& \notin T}{T \not\leqslant \&_{i\in I} ?l_i(S_i).T_i}$$

Table 15. Negation of asynchronous subtyping rules.

of the asynchronous subtyping implies incompleteness of the synchronous subtyping.

We show completeness for asynchronous subtyping following the three steps described in § 3. In the third step we need to add two queues for exchanging messages, see the proof of Theorem 5.4. The proofs are more delicate than in the synchronous case since the asynchronous subtyping makes the shapes of types less structured. The first difficulty is to define the negation $\not\leqslant_{\mathsf{a}}$ inductively and prove that it implies the non-derivability of $\leqslant_{\mathsf{a}}$. The second difficulty is to catch the error states arising after an unbounded number of message enqueues, since rule [SUB-PERM-ASYNC] 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 $\mathbf{P}(u,T)$, is defined as in Definition 3.1, but for the case of $\mathbf{P}_!(u,l,S,T)$, which is now:

$$\mathbf{P}_!(u,l,S,T) \stackrel{\text{def}}{=} (\nu ab)(u!l\langle a\rangle.\mathbf{P}(u,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing)$$

For example if $T = !l_1\langle \mathtt{end}\rangle \oplus !l_2\langle !l_3\langle \mathtt{end}\rangle.\mathtt{end}\rangle.\mathtt{end}$, then

$$\begin{aligned}
\mathbf{P}(a,T) &= \mathbf{P}_!(a,l_1,\mathtt{end},\mathtt{end}) \oplus \mathbf{P}_!(a,l_2,!l_3\langle \mathtt{end}\rangle.\mathtt{end},\mathtt{end})\\
&= (\nu bb')(a!l_1\langle b\rangle.\mathbf{P}(a,\mathtt{end}) \mid \mathbf{P}(b',\mathtt{end}) \mid bb' \blacktriangleright \emptyset \mid b'b \blacktriangleright \emptyset) \oplus\\
&\quad (\nu cc')(a!l_2\langle c\rangle.\mathbf{P}(a,\mathtt{end}) \mid \mathbf{P}(c',?l_3(\mathtt{end}).\mathtt{end}) \mid\\
&\qquad\qquad cc' \blacktriangleright \emptyset \mid c'c \blacktriangleright \emptyset)\\
&= (\nu bb')(a!l_1\langle b\rangle.\mathbf{0} \mid \mathbf{0} \mid bb' \blacktriangleright \emptyset \mid b'b \blacktriangleright \emptyset) \oplus\\
&\quad (\nu cc')(a!l_2\langle c\rangle.\mathbf{0} \mid c'?l_3(x).(\mathbf{P}(c',\mathtt{end}) \mid \mathbf{P}(x,\mathtt{end})) \mid\\
&\qquad\qquad cc' \blacktriangleright \emptyset \mid c'c \blacktriangleright \emptyset)\\
&= (\nu bb')(a!l_1\langle b\rangle.\mathbf{0} \mid \mathbf{0} \mid bb' \blacktriangleright \emptyset \mid b'b \blacktriangleright \emptyset) \oplus\\
&\quad (\nu cc')(a!l_2\langle c\rangle.\mathbf{0} \mid c'?l_3(x).(\mathbf{0} \mid \mathbf{0}) \mid cc' \blacktriangleright \emptyset \mid c'c \blacktriangleright \emptyset)\\
&\equiv (\nu bb')(a!l_1\langle b\rangle \mid bb' \blacktriangleright \emptyset \mid b'b \blacktriangleright \emptyset) \oplus\\
&\quad (\nu cc')(a!l_2\langle c\rangle \mid c'?l_3(x) \mid cc' \blacktriangleright \emptyset \mid c'c \blacktriangleright \emptyset)
\end{aligned}$$

Similarly to Lemma 3.2 we get:

Lemma 5.2. $\vdash_{\mathsf{a}} \mathbf{P}(u,T) \triangleright \{u : T\}$.

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:

$$[\,]^{n'}\!\downarrow n \;=\; \begin{cases} [\,]^{n} & \text{if } n = n', \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$(\&_{i\in I}?l_i(S_i).\mathscr{A}_i)\!\downarrow n \;=\; \begin{cases} ?l_{i_0}(S_{i_0}).(\mathscr{A}_{i_0}\!\downarrow n) & \text{if } [\,]^n \text{ occurs in } \mathscr{A}_{i_0}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For example $(?\mathsf{r}(S_r).[\,]^1 \& ?\mathsf{s}(S_s).[\,]^2)\!\downarrow 2 = ?\mathsf{s}(S_s).[\,]^2$.

The negation rules of asynchronous subtyping are the rules of Table 7 *excluding* rule [N-SELBRA-SYNC] plus the rules of Table 15. Rule [N-LABEL-ASYNC] deals with the case that the selection cannot find a matching label inside the n_0-th hole. Rule [N-EXCH-ASYNC] considers a mismatch between carried types inside the n_0-th hole. Rule [N-CONT-ASYNC] considers a mismatch between continuation types, always inside the n_0-th hole, but preceded by the inputs needed to reach that hole. These three rules correspond to the rules [N-LABEL SEL], [N-EXCH SEL] and [N-CONT SEL] of Table 7, respectively, when the context $\mathscr{A}$ is just one hole. Rule [N-BRA-ASYNC] assures that a type without branchings cannot be a subtype of a branching type. We write $T \not\leqslant_{\mathsf{a}} S$ if $T \not\leqslant S$ is generated by the rules in Table 15 and Table 7 excluding rule [N-SELBRA-SYNC].

For example, by rule [N-CONT-ASYNC],

$$!\mathsf{m}\langle S_m\rangle.(?\mathsf{r}(S_r).T_r \& ?\mathsf{s}(S_s).\mathtt{end}) \not\leqslant_{\mathsf{a}} ?\mathsf{r}(S_r).!\mathsf{m}\langle S_m\rangle.T_r \& ?\mathsf{s}(S_s).!\mathsf{m}\langle S_m\rangle.!\mathsf{p}\langle S_p\rangle.T_p$$

since $?\mathsf{r}(S_r).T_r \& ?\mathsf{s}(S_s).\mathtt{end} \not\leqslant_{\mathsf{a}} ?\mathsf{r}(S_r).T_r \& ?\mathsf{s}(S_s).!\mathsf{p}\langle S_p\rangle.T_p$.

We can prove that $\not\leqslant_{\mathsf{a}}$ is the negation of the asynchronous subtyping, by observing that the premises of rule [SUB-PERM-ASYNC] correspond to the rules of Table 15.

Lemma 5.3. *$S \leqslant_{\mathsf{a}} T$ is not derivable if and only if $S \not\leqslant_{\mathsf{a}} T$ is derivable.*

Completeness can now be shown:

Theorem 5.4 (Completeness for asynchronous subtyping)**.** *The asynchronous subtyping relation $\leqslant_{\mathsf{a}}$ is complete for the asynchronous calculus.*

Proof. We prove that $T \not\leqslant_{\mathsf{a}} S$ implies

$$(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing) \to_{\mathsf{a}}^{*} \mathtt{error}$$

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

1. For rule [N-LABEL-ASYNC] we get

   $$T = \bigoplus_{i\in I} !l_i\langle T_i'\rangle.T_i,\;\; S = \mathscr{A}[\bigoplus_{j\in J_n} !l_j'^{\,n}\langle S_j'^{\,n}\rangle.S_j^n]^{n\in N},$$

   and there are $i_0 \in I, n_0 \in N$ such that for all $j \in J_{n_0}$ we get $l_j'^{\,n_0} \neq l_{i_0}$. We show by induction on $\mathscr{A}$ that $T \not\leqslant_{\mathsf{a}} S$ implies

   $$(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright h \mid ab \blacktriangleright \varnothing) \to_{\mathsf{a}}^{*} \mathtt{error}$$

   for an arbitrary queue h.

   (a) If $\mathscr{A}$ is just one hole, then $S = \bigoplus_{j\in J} !l_j'\langle S_j'\rangle.S_j$, where $J = J_{n_0}$, and $\overline{S} = \&_{j\in J} ?l_j'(S_j').\overline{S_j}$. We get

   $$\begin{aligned} &(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright h \mid ab \blacktriangleright \varnothing) \\ &\to_{\mathsf{a}} (\nu ab)(\mathbf{P}_!(a,l_{i_0},T_{i_0}',T_{i_0}) \mid \textstyle\sum_{j\in J} \mathbf{P}_?(b,l_j',S_j',\overline{S_j}) \\ &\qquad \mid ba \blacktriangleright h \mid ab \blacktriangleright \varnothing) \to_{\mathsf{a}} (\nu ab)((\nu cd)(\mathbf{P}(a,T_{i_0}) \mid \mathbf{P}(d,\overline{T_{i_0}'}) \\ &\qquad \mid dc \blacktriangleright \varnothing \mid cd \blacktriangleright \varnothing \\ &\qquad \mid \textstyle\sum_{j\in J} b?l_j'(x).(\mathbf{P}(b,\overline{S_j}) \mid \mathbf{P}(x,S_j')) \\ &\qquad \mid ba \blacktriangleright h \mid ab \blacktriangleright\; l_{i_0}\langle c\rangle)) \\ &\to_{\mathsf{a}} \mathtt{error} \end{aligned}$$

   by rule [ERR-MISM-ASYNC].

   (b) If $\& \in \mathscr{A}$, let $\mathscr{A} = \&_{k\in K} ?l_k^*(S_k^*).\mathscr{A}_k[\,]^{n\in N_k}$, where $\bigcup_{k\in K} N_k = N$. Then

   $$\overline{S} \;=\; \bigoplus_{k\in K} !l_k^*\langle S_k^*\rangle.\overline{\mathscr{A}_k}[\&_{j\in J_n} ?l_j'^{\,n}(S_j'^{\,n}).\overline{S_j}^{\,n}]^{n\in N_k}.$$

   Let $k_0 \in K$ be such that $n_0 \in N_{k_0}$ and

   $$V = \overline{\mathscr{A}_{k_0}}[\&_{j\in J_n} ?l_j'^{\,n}(S_j'^{\,n}).\overline{S_j}^{\,n}]^{n\in N_{k_0}}.$$

   We get

   $$\begin{aligned} &(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright h \mid ab \blacktriangleright \varnothing) \\ \to_{\mathsf{a}}\; &(\nu ab)(\mathbf{P}(a,T) \mid (\nu cd)(b!l_{k_0}^*\langle c\rangle.\mathbf{P}(b,V) \mid \mathbf{P}(d,\overline{S_{k_0}^*}) \\ &\qquad \mid dc \blacktriangleright \varnothing \mid cd \blacktriangleright \varnothing) \mid ba \blacktriangleright h \mid ab \blacktriangleright \varnothing) \\ \to_{\mathsf{a}}\; &(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,V) \mid (\nu cd)(\mathbf{P}(d,\overline{S_{k_0}^*}) \\ &\qquad \mid dc \blacktriangleright \varnothing \mid cd \blacktriangleright \varnothing \mid ba \blacktriangleright h \cdot l_{k_0}^*\langle c\rangle \mid ab \blacktriangleright \varnothing)) \end{aligned}$$

   and by induction
   $(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,V) \mid ba \blacktriangleright h \cdot l_{k_0}^*\langle c\rangle \mid ab \blacktriangleright \varnothing) \to_{\mathsf{a}}^{*} \mathtt{error}$

2. For rule [N-BRA-ASYNC] we get $\& \notin T$ and $S = \&_{i\in I} ?l_i(S_i).T_i$. Then $\overline{S} = \bigoplus_{i\in I} !l_i\langle S_i\rangle.\overline{T_i}$.

   $$\begin{aligned} &(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{S}) \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing) \\ \to_{\mathsf{a}}\; &(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}_!(b,l_{i_0},S_{i_0},\overline{T_{i_0}}) \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing) \\ \to_{\mathsf{a}}\; &C[(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{T_{i_0}}) \mid ba \blacktriangleright l_{i_0}\langle c\rangle \mid ab \blacktriangleright \varnothing)] \end{aligned}$$

   where $C[\,] = (\nu cd)([\,] \mid \mathbf{P}(d,\overline{S_{i_0}}) \mid dc \blacktriangleright \varnothing \mid cd \blacktriangleright \varnothing)$.

   It is easy to verify that $a \notin \varphi(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{T_{i_0}}) \mid ab \blacktriangleright \varnothing)$ since $\& \notin T$, and $\mathsf{fpv}(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{T_{i_0}}) \mid ab \blacktriangleright \varnothing) = \emptyset$, so by [ERR-ORPH-MESS-ASYNC] we get

   $(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{T_{i_0}}) \mid ba \blacktriangleright l_{i_0}\langle c\rangle \mid ab \blacktriangleright \varnothing) \to_{\mathsf{a}} \mathtt{error}$.

   Then by [ERR-CONTEXT]

   $C[(\nu ab)(\mathbf{P}(a,T) \mid \mathbf{P}(b,\overline{T_{i_0}}) \mid ba \blacktriangleright l_{i_0}\langle c\rangle \mid ab \blacktriangleright \varnothing)] \to_{\mathsf{a}}^{*} \mathtt{error}$. □

6. Denotational Preciseness

In λ-calculus types are usually interpreted as subsets of the domains of λ-models [2, 18]. *Denotational preciseness* of subtyping is then:

$$T \leqslant S \text{ if and only if } [\![T]\!] \subseteq [\![S]\!],$$

using $[\![\,]\!]$ 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.

$$[\![T]\!]_* = \{P \;\mid\; \vdash_* P \triangleright \{a : T\}\}$$

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

$$\vdash_* \mathbf{P}_*(a,T) \triangleright \{a : S\} \text{ implies } T \leqslant_* S,$$

where $\mathbf{P}_*(a,T)$ is the synchronous characteristic process if $* = \mathsf{s}$ and the asynchronous characteristic process if $* = \mathsf{a}$. If we could derive $\vdash_* \mathbf{P}_*(a,T) \triangleright \{a : S\}$ with $T \not\leqslant_* S$, then the processes

$$(\nu ab)(\mathbf{P}_{\mathsf{s}}(a,T) \mid \mathbf{P}_{\mathsf{s}}(b,\overline{S})) \text{ and}$$

$$(\nu ab)(\mathbf{P}_{\mathsf{a}}(a,T) \mid \mathbf{P}_{\mathsf{a}}(b,\overline{S}) \mid ba \blacktriangleright \varnothing \mid ab \blacktriangleright \varnothing)$$

could be typed. Theorems 3.4 and 5.4 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 $T \not\leqslant_* S$, then $\mathbf{P}_*(a,T) \in [\![T]\!]_*$, but $\mathbf{P}_*(a,T) \notin [\![S]\!]_*$.

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 rôle 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 [32, 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 [6] for a π-calculus with a patterned 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, 7].

Subtyping of Mostrous Note that our subtyping relation differs from that defined in [27] only for the premises $\& \in \mathscr{A}$ and $\& \in T_i$ in rule [SUB-PERM-ASYNC]. As a consequence T is a subtype of S when $T = \mu\mathbf{t}.!l\langle T'\rangle.\mathbf{t}$ and $S = \mu\mathbf{t}.!l\langle T'\rangle.?l'(S').\mathbf{t}$ (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 $C = (\nu ab)([\,] \mid Q \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright\varnothing)$ where

$$Q = \mathtt{def}\ Y(x) = b!l\langle x\rangle.b?l'(y).Y\langle x\rangle\ \mathtt{in}\ (\nu cc')(Y\langle c\rangle).$$

Then we can derive $C[a : S] \triangleright \emptyset$. Let

$$P = \mathtt{def}\ Z(z) = a!l\langle z\rangle.Z\langle z\rangle\ \mathtt{in}\ (\nu dd')(Z\langle d\rangle).$$

Then $P \triangleright \{a : T\}$. We get

$$C[P] \to_{\mathsf{a}}^{*} (\nu ab)(\nu cc')(P \mid Q \mid ab\blacktriangleright\varnothing \mid ba\blacktriangleright l\langle c\rangle) \to_{\mathsf{a}} \mathtt{error}$$

by rule [ERR-ORPH-MESS-ASYNC], since $a \notin \varphi(P \mid Q \mid ab\blacktriangleright\varnothing)$ and $\mathsf{fpv}(P \mid Q \mid ab\blacktriangleright\varnothing) = \emptyset$.

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 $\mu\mathbf{t}.!l\langle T\rangle.\mathbf{t} \not\leqslant \mu\mathbf{t}.?l'(S).\mathbf{t}$. There is no context C which is safe for all processes with one channel typed by $\mu\mathbf{t}.?l'(S).\mathbf{t}$ and no process P with one channel typed by $\mu\mathbf{t}.!l\langle T\rangle.\mathbf{t}$ 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 calculus. 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 ($\langle T, \overline{T}\rangle$ 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. $\mathtt{succ}(n)$) 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 [1]. 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 [23]. Such a characterisation would give preciseness for the many type systems which are instances of [23].

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 [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 rôle since a programmer can adjust a subtyping relation to loosen or tighten subtypings 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. 2010LHT4KM 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

[1] F. Barbanera and U. de Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, pages 155-164. ACM, 2010.

[2] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4): 931-940, 1983.

[3] J. Blackburn, I. Hernandez, J. Ligatti, and M. Nachtigal. Completely subtyping iso-recursive types. Technical Report CSE-071012, University of South Florida, 2012.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems*, 34(2):8:1–8:78, 2012.

[5] L. Cardelli. Amber. In *Combinators and functional programming languages*, volume 242 of *LNCS*, pages 21–47. Springer, 1986.

[6] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. *Theoretical Computer Science*, 398(1–3):217–242, 2008.

[7] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of session types. In *PPDP*, pages 219–230. ACM, 2009.

[8] R. Demangeon and K. Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *LNCS*, pages 280-296. Springer, 2011.

[9] M. Dezani-Ciancaglini and S. Ghilezan. Preciseness of subtyping on intersection and union types. In *RTATLCA*, volume 8560 of *LNCS*, pages 194–207. Springer, 2014.

[10] M. Dezani-Ciancaglini, U. de' Liguoro, and A. Piperno. A filter model for concurrent lambda-calculus. *SIAM Journal on Computing*, 27(5): 1376–1419, 1998.

[11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of ACM*, 55(4):1–64, 2008.

[12] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[13] S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[14] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.

[15] M. Goto, R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science*, 2014. To appear.

[16] M. Gouda, E. Manning, and Y. Yu. On the progress of communication between two finite state machines. *Information and Control*, 63:200–216, 1984.

[17] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

[18] J. R. Hindley. The completeness theorem for typing lambda-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[19] D. Hirschkoff, J.-M. Madiot, and D. Sangiorgi. Name-passing calculi: from fusions to preorders and types. In *LICS*, pages 378-387. IEEE Computer Society, 2013.

[20] K. Honda and N. Yoshida. On reduction-based process semantics. *Theorical Computer Science*, 151(2):437–486, 1995.

[21] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[22] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[23] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.

[24] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transaction on Programming Languages and Systems*, 21(5):914–947, 1999.

[25] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer, 1991.

[26] R. Milner and D. Sangiorgi. Barbed bisimulation. In *ICALP*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[27] D. Mostrous. *Session Types in Concurrent Calculi: Higher-Order Processes and Objects*. PhD thesis, Imperial College London, 2009.

[28] D. Mostrous and N. Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.

[29] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.

[30] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

[31] L. Padovani. Fair subtyping for open session types. In *ICALP*, volume 7966 of *LNCS*, pages 373–384. Springer, 2013.

[32] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[33] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):376–385, 1996.

[34] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[35] Scribble. Scribble Project homepage. `http://scribble.github.io/`.

[36] V. T. Vasconcelos. Fundamentals of session types. In *SFM*, volume 5569 of *LNCS*, pages 158–186. Springer, 2009.

[37] V. T. Vasconcelos. Session types for linear multithreaded functional programming. In *PPDP*, pages 1–6. ACM, 2009.

[38] N. Yoshida, V. T. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *FMCO*, volume 5751 of *LNCS*, pages 226–246. Springer, 2009.