Global Principal Typing in Partially Commutative Asynchronous Sessions

Dimitris Mostrous[1], Nobuko Yoshida[1], and Kohei Honda[2]

[1] Department of Computing, Imperial College London
[2] Department of Computer Science, Queen Mary, University of London

Abstract. We generalise a theory of multiparty session types for the π-calculus through asynchronous communication subtyping, which allows partial commutativity of actions with maximal flexibility and safe optimisation in message choreography. A sound and complete algorithm for the subtyping relation, which can calculate conformance of optimised end-point processes to an agreed global specification, is presented. As a complementing result, we show a type inference algorithm for deriving the principal global specification from end-point processes which is minimal with respect to subtyping. The resulting theory allows a programmer to choose between a top-down and a bottom-up style of communication programming, ensuring the same desirable properties of typable processes.

1 Introduction

Programs which communicate by asynchronous message passing are abundant in critical computing scenes, from a simple web-service application between two parties to a global financial network hosting thousands of nodes and billions of messages per year. The design of such programs, which may be developed in geographically disparate sites, demands a clear high-level specification of their conversation structure, against which participating programs can be validated (*conformance*). Further such specifications may change during development (*refinement*), and might even need to be synthesised from individual endpoint programs, against which updated programs can be further validated (*synthesis of global specifications*).

This paper develops a new theory of multiparty session types [1, 2, 4, 5, 13, 23], which can handle uniformly these three concerns by seamlessly integrating the *top-down* and *bottom-up* strategies for the development of communication-centred software. The methodology for distributed programming put forward in [1, 13] centres on the concept of a *global type* which plays the role of type signature for distributed communications, presenting an abstract high-level description of the protocol that all the participants have to honour when an actual conversation takes place. Building on this framework, we propose the following two strategies for communication programming:

Top-Down Approach: Once this signature G is agreed upon by all parties as a global protocol, a local protocol from each party's viewpoint (*local type* T_i) is generated as a projection of G to each party. Then each local type T_i can be *locally refined* to, say, T_i', possibly giving a more *optimised* protocol, realised as a program, say, P_i. If all the resulting local programs $P_1,..,P_n$ can be type-checked against refined $T_1',..,T_n'$, then they are automatically guaranteed to interact properly, without communication mismatch or getting stuck inside a session, precisely following the intended scenario.

Bottom-Up Approach: In this case the programmers may work based on an informal understanding of shared conversation structures, which, after appropriate development, will get reified into a formal global protocol by synthesis of local behaviours of all end-point programs: first, a type T_i is inferred from each program P_i, then a new global specification is synthesised from $T_1,..,T_n$. If this specification is validated to satisfy certain conditions, $P_1,..,P_n$ are guaranteed to interact properly. This process can be repeated incrementally, using a succession of synthesised types as *globally refined* protocols.

This paper presents a general and rigorous foundation for these two approaches and their seamless integration, based on multiparty session types. For the automatic refinement, we introduce *asynchronous communication subtyping* over local types, which allows permutation of actions to increase efficiency, while ensuring type-soundness and communication-safety. As an example, suppose we are using an asynchronous communication transport where the message order is preserved but the sending is non-blocking, as in TCP. Let us assume the following three simple processes:

$$P_1 \stackrel{\text{def}}{=} t?(y_1);s!\langle 5\rangle;s!\langle \text{apple}\rangle;Q_1 \quad P_2 \stackrel{\text{def}}{=} b?(y_2);t!\langle 7\rangle;Q_2 \quad P_3 \stackrel{\text{def}}{=} s?(z_1);s?(z_2);Q_3$$

where $s?(y)$ is an input and $s!\langle 5\rangle$ is an output via channel s; and ";" is sequential composition. Then first P_2 gets the value at b; then P_2 sends 7 to P_1; finally P_1 sends 5 and "apple" to P_3 preserving the order. We note that P_3 is blocked until b is fired at P_2. To execute P_3 ahead, P_1 might be locally optimised since y_1 does not bind the subsequent outputs at s. We can similarly optimise P_2. The resulting processes given below still preserve linearity and proper communication structures.

$$P_1' \stackrel{\text{def}}{=} s!\langle 5\rangle;s!\langle \text{apple}\rangle;t?(y_1);Q_1 \quad P_2' \stackrel{\text{def}}{=} t!\langle 7\rangle;b?(y_2);Q_2 \quad P_3 \stackrel{\text{def}}{=} s?(z_1);s?(z_2);Q_3$$

Asynchronous communication subtyping specifies safe permutations of actions, by which we can refine a local protocol to maximise asynchrony without violating the global protocol. For example, in the above case, P_1' is given local type $s!\langle \mathsf{nat}\rangle;s!\langle \mathsf{string}\rangle;t?\langle \mathsf{nat}\rangle;T$ which is a subtype of $t?\langle \mathsf{nat}\rangle;s!\langle \mathsf{nat}\rangle;s!\langle \mathsf{string}\rangle;T$ projected from the global type. Hence optimisations can be checked locally. The idea of this subtyping is intuitive, but it requires delicate formal formulations due to the presence of recursive types and branching/selection session types, whose combinations are vital for typing many practical protocols [12, 22]. This subtlety is because type-permutations *affect the structures of session types*, which makes straightforward constructions following the preceding literature [10, 20] inapplicable. Intuitively, because partial commutativity is defined between *a sequence of actions*, it may require *more than one unfolding* of recursive types to find a match. However, this calculation can be made automatic by an *algorithmic subtyping* which completely characterises the semantic notion of subtyping and can be used to effectively (in)validate conformance of an optimised local type to a global type.

For the bottom-up strategy, we formulate *principal global typing* by which we can synthesise the most general global type from untyped endpoint programs, or can check they can have no global type, i.e. their protocols are incompatible. The framework uses graph-shaped global types which generalise the original syntactic global types, extending typability. Asynchronous communication subtyping plays a central role in the synthesis process. We demonstrate the use of the theory for the two strategies by providing correctness arguments for the development of a distributed parallel algorithm. A full version, containing more examples and detailed proofs, is available from `http://www.doc.ic.ac.uk/~mostrous/asyncsub`.

2 Asynchronous Multiparty Sessions

Syntax. We use the π-calculus for multiparty sessions from [13], omitting polyadicity and delegation for simplicity. We use the following base sets: *shared names* or *names*, written $a,b,x,y,z,\dots$; *session channels* or *channels*, written $s,t,\dots$; *labels* (functioning like labels in labelled records), written $l,l',\dots$; and *process variables*, written $X,Y,\dots$. For hiding, we use n for either a single shared name or a vector of session channels. Then *processes* $(P,Q\dots)$ and *expressions* $(e,e',\dots)$ are given below:

$$\begin{array}{rcll@{\qquad}rcll}
P & ::= & \overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P & \text{request} & & \mid & P \mid Q & \text{parallel}\\
& \mid & a_{[\mathtt{p}]}(\tilde{s}).P & \text{acceptance} & & \mid & \mathbf{0} & \text{inaction}\\
& \mid & s!\langle e\rangle;P & \text{sending} & & \mid & (\nu\, n)P & \text{hiding}\\
& \mid & s?(x);P & \text{reception} & & \mid & \mathsf{def}\ D\ \mathsf{in}\ P & \text{recursion}\\
& \mid & s \lhd l;P & \text{selection} & & \mid & X\langle \tilde{e}\tilde{s}\rangle & \text{process call}\\
& \mid & s \rhd \{l_i\colon P_i\}_{i\in I} & \text{branching} & & \mid & s:\tilde{h} & \text{message queue}\\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{conditional} & h & ::= & l \mid v & \text{message values}\\
e & ::= & v \mid e\ \mathtt{and}\ e' \mid \mathtt{not}\ e \cdots & \text{expressions} & D & ::= & \{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I} & \text{declaration}\\
v & ::= & a \mid \mathsf{true} \mid \mathsf{false} \cdots & \text{values} & & & &
\end{array}$$

$\overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P$ initiates, through a shared name a, a new session with other participants, each of shape $a_{[\mathtt{p}]}(\tilde{s}).Q$ with $2 \le \mathtt{p} \le n$. The (bound) s_i in vector $\tilde{s}$ are session channels used in the session. We call p, q,... (natural numbers) the *participants* of a session. Session communications (which take place inside an established session) are performed by the sending and receiving of a value; and by selection and branching (the former chooses one of the branches offered by the latter). $s:\tilde{h}$ is a *message queue* representing ordered messages in transit $\tilde{h}$ with destination s (which may be considered as a network pipe in a TCP-like transport). The rest of the syntax is standard from [13]. We often omit $\mathbf{0}$, and unimportant arguments of sending/receiving, e.g. $s!\langle\rangle$ and $s?();P$.

Operational semantics. Some selected rules of reduction $P \to P'$ are given below:

$$\begin{array}{l}
\overline{a}_{[2..\mathtt{n}]}(\tilde{s}).P_1 \mid a_{[2]}(\tilde{s}).P_2 \mid \cdots \mid a_{[\mathtt{n}]}(\tilde{s}).P_{\mathtt{n}} \;\to\; (\nu\,\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_{\mathtt{n}} \mid s_1:\emptyset \mid \ldots \mid s_m:\emptyset)\\
s!\langle e\rangle;P \mid s:\tilde{h} \to P \mid s:\tilde{h}\cdot v \quad (e \downarrow v) \qquad s?(x);P \mid s:v\cdot\tilde{h} \;\to\; P[v/x] \mid s:\tilde{h}\\
s \lhd l;P \mid s:\tilde{h} \to P \mid s:\tilde{h}\cdot l \qquad\qquad\qquad s \rhd \{l_i\colon P_i\}_{i\in I} \mid s:l_j\cdot\tilde{h} \;\to\; P_j \mid s:\tilde{h} \qquad (j \in I)
\end{array}$$

The first rule describes the initiation of a new session among n participants that synchronise over the shared name a. After the initiation, they will share the private m fresh session channels s_i and the associated m empty queues ($\emptyset$ denotes the empty string). The output rules enqueue a value and a label, respectively ($e \downarrow v$ denotes the evaluation of e to v). The input rules perform the complementary operations. Processes are considered modulo structural equivalence, $\equiv$, defined by the standard rules [13].

Global types. A *global type*, written $G, G', ..$, describes the whole conversation scenario of a multiparty session as a type signature [13].

$$\begin{array}{lclllll} \text{Global } G & ::= & \mathsf{p} \to \mathsf{p}' \colon k \langle U \rangle ; G' & \text{values} & \mid \mu \mathbf{t}.G & \text{recursive} \\ & \mid & \mathsf{p} \to \mathsf{p}' \colon k \{ l_j \colon G_j \}_{j \in J} & \text{branching} & \mid \mathbf{t} & \text{variable} \\ & \mid & G, G' & \text{parallel} & \mid \mathsf{end} & \text{end} \\ \text{Value } U & ::= & \mathsf{bool} \mid \mathsf{nat} \mid \cdots \mid G & & & \end{array}$$

Type $\mathsf{p} \to \mathsf{p}' \colon k \langle U \rangle ; G'$ says that participant p sends a message of type U to channel k (represented as a finite natural number) received by participant p' and then interactions described in G' take place. *Value types* range over U, and are either global types for shared names, or base values. Type $\mathsf{p} \to \mathsf{p}' \colon k \{ l_j \colon G_j \}_{j \in J}$ says that participant p invokes one of the l_i labels on channel k (at participant p'), then interactions described in G_j take place. Type $\mu \mathbf{t}.G$ is for recursive protocols, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e. they only occur under values or branches. We assume G in value types is closed, i.e. without free type variables. Type end represents the termination of a session. We often omit end and identify "G, end" and "end, G" with G. We stipulate that each channel can only be used among two parties (but maybe repeatedly), one party using it for input/branching while the other party for output/selection.[3]

Local types. Local session types type-abstract sessions from each endpoint's view.

$$\begin{array}{lclllll} \text{Local } T & ::= & k! \langle U \rangle ; T & \text{send} & \mid k \& \{ l_i \colon T_i \}_{i \in I} & \text{branching} \\ & \mid & k? \langle U \rangle ; T & \text{receive} & \mid \mu \mathbf{t}.T \mid \mathbf{t} & \text{recursion} \\ & \mid & k \oplus \{ l_i \colon T_i \}_{i \in I} & \text{selection} & \mid \mathsf{end} & \text{end} \end{array}$$

Type $k! \langle U \rangle$ expresses the sending to k of a value of type U. Type $k? \langle U \rangle$ is its dual input. Type $k \oplus \{ l_i \colon T_i \}_{i \in I}$ represents the transmission to k of a label l_i chosen in the set $\{ l_i \mid i \in I \}$, followed by the communications described by T_i. Type $k \& \{ l_i \colon T_i \}_{i \in I}$ is its dual. The remaining types are standard. We say a type is *guarded* if it is neither a recursive type nor a type variable. (An occurrence of) a type constructor *not* under a recursive prefix in a recursive type is called *top-level action* (for example, $k_1! \langle U_1 \rangle$ and $k_2! \langle U_2 \rangle$ in $k_1! \langle U_1 \rangle ; k_2! \langle U_2 \rangle ; \mu \mathbf{t}.k_3! \langle U_3 \rangle ; \mathbf{t}$ are top-level, but $k_3! \langle U_3 \rangle$ in the same type is not). k is *the head of* T if k appears at the left-most occurrence of the top-level actions in T (e.g. $k_1! \langle U_1 \rangle$ is the head of the above type). The relation between global and local types is formalised by *projection*, written $G \restriction \mathsf{p}$ (called *projection of* G *onto* p), defined as in [13]. For example, $(\mathsf{p} \to \mathsf{p}' \colon k \langle U \rangle ; G') \restriction \mathsf{p} = k! \langle U \rangle ; (G' \restriction \mathsf{p})$, $(\mathsf{p} \to \mathsf{p}' \colon k \langle U \rangle ; G') \restriction \mathsf{p}' = k? \langle U \rangle ; (G' \restriction \mathsf{p}')$ and $(\mathsf{p} \to \mathsf{p}' \colon k \langle U \rangle ; G') \restriction \mathsf{q} = (G' \restriction \mathsf{q})$. We write *Type* for the collection of all closed local types.

3 Asynchronous Partially Commutative Sessions

3.1 Asynchronous Communication Subtyping: Top-Level Actions

This section introduces and studies a basic theory of asynchronous session subtyping. Figure 1 defines the axioms for partial permutation of top-level actions for closed types,

[3] This condition dispenses with the need for linearity-check to ensure well-formedness [1].

$$\begin{array}{lrcl}
(\mathsf{OI}) & k!\langle U\rangle;k'?\langle U'\rangle;T & \ll & k'?\langle U'\rangle;k!\langle U\rangle;T \\
(\mathsf{OB}) & k!\langle U\rangle;k'\&\{l_j:T_j\}_{j\in J} & \ll & k'\&\{l_j:k!\langle U\rangle;T_j\}_{j\in J} \\
(\mathsf{SI}) & k\oplus\{l_j:k'?\langle U\rangle;T_j\}_{j\in J} & \ll & k'?\langle U\rangle;k\oplus\{l_j:T_j\}_{j\in J} \\
(\mathsf{SB}) & k\oplus\{l_i:k'\&\{l'_j:T_{ij}\}_{j\in J}\}_{i\in I} & \ll & k'\&\{l'_j:k\oplus\{l_i:T_{ij}\}_{i\in I}\}_{j\in J} \\
(\mathsf{OO}) & k!\langle U\rangle;k'!\langle U'\rangle;T & \ll & k'!\langle U'\rangle;k!\langle U\rangle;T \\
(\mathsf{II}) & k?\langle U\rangle;k'?\langle U'\rangle;T & \ll & k'?\langle U'\rangle;k?\langle U\rangle;T \\
(\mathsf{SO}) & k\oplus\{l_i:k'!\langle U\rangle;T_i\}_{i\in I} & \ll & k'!\langle U\rangle;k\oplus\{l_i:T_i\}_{i\in I} \\
(\mathsf{OS}) & k'!\langle U\rangle;k\oplus\{l_i:T_i\}_{i\in I} & \ll & k\oplus\{l_i:k'!\langle U\rangle;T_i\}_{i\in I} \\
(\mathsf{SS}) & k\oplus\{l_i:k'\oplus\{l'_j:T_{ij}\}_{j\in J}\}_{i\in I} & \ll & k'\oplus\{l'_j:k\oplus\{l_i:T_{ij}\}_{i\in I}\}_{j\in J}
\end{array}$$

$$(\mathsf{Tr})\ \frac{T_1\ll T_2 \quad T_2\ll T_3}{T_1\ll T_3} \qquad (\mathsf{CO})\ \frac{T\ll T'}{k!\langle U\rangle;T\ll k!\langle U\rangle;T'} \qquad (\mathsf{CI})\ \frac{T\ll T'}{k?\langle U\rangle;T\ll k?\langle U\rangle;T'}$$

$$(\mathsf{CB})\ \frac{\forall i\in I.\ T_i\ll T'_i}{k\&\{l_i:T_i\}_{i\in I}\ll k\&\{l_i:T'_i\}_{i\in I}} \qquad (\mathsf{CS})\ \frac{\forall i\in I.\ T_i\ll T'_i}{k\oplus\{l_i:T_i\}_{i\in I}\ll k\oplus\{l_i:T'_i\}_{i\in I}} \qquad \begin{array}{l}(\mathsf{E})\ \mathsf{end}\ll\mathsf{end}\\ (\mathsf{M})\ \mu\mathbf{t}.T\ll\mu\mathbf{t}.T\end{array}$$

Fig. 1. Action Asynchronous Subtyping Rules ($(\mathsf{BI},\mathsf{IB},\mathsf{BB})$ are omitted)

denoted $\ll$. We assume $k\neq k'$ for all the axioms. $T\ll T'$ is read: T is an *action-asynchronous subtype* of T', and means T is more asynchronous than (or more optimised than) T'. We write $T\gg T'$ for $T'\ll T$.

A partial permutation is applied only to finite parts of the top-level actions (*without* unfolding recursive types); see Proposition 7. Note that we *cannot* exchange an input and output in the reverse direction of (OI) even for different channels. Consider: $P=s?();r!\langle\rangle$ and $Q=s!\langle\rangle;r?()$. These processes interact correctly. If we permute the output and input of Q, we get $Q'=r?();s!\langle\rangle$. Then the parallel composition $(P\mid Q')$ causes deadlock, losing progress. For the same reason, the reverse direction of $(\mathsf{OB},\mathsf{SI},\mathsf{SB})$ is not allowed. By combining these input and output permutation rules, we can achieve a flexible local refinement for communications. For example, suppose $R=s?(x);r?(y);t!\langle 1\rangle;t'!\langle y\rangle$ typed by $T_R=s?\langle\mathsf{file}\rangle;r?\langle\mathsf{bool}\rangle;t!\langle\mathsf{nat}\rangle;t'!\langle\mathsf{bool}\rangle;\mathsf{end}$. We might wish to receive the (small) value via r first, and immediately forward to t', then receive the (larger) file at s in the end: we can obtain $S=r?(y);t'!\langle y\rangle;t!\langle 1\rangle;s?(x)$ typed by $T_S=r?\langle\mathsf{bool}\rangle;t'!\langle\mathsf{bool}\rangle;t!\langle\mathsf{nat}\rangle;s?\langle\mathsf{file}\rangle;\mathsf{end}$, transformed from T_R (i.e. $T_S\ll T_R$) by using a combination of $(\mathsf{OO},\mathsf{OI},\mathsf{II})$.

3.2 Asynchronous Communication Subtyping: Recursive Types

For handling recursive types in asynchronous subtyping, we extend the coinductive method in [20, § 2.3] and [10, § 3.3]. In particular, we need to modify the unfolding function for recursive types since $\ll$ might be applicable to *a sequence of types* after unfolding of recursions under guarded prefixes. The resulting definition integrates $\ll$ with the traditional session subtyping [10, 13]. For any recursive type T, $\mathsf{unfold}^n(T)$ is the result of inductively unfolding the first recursion (even under guarded types) up to a fixed level of nesting.

Definition 1 **(n-time unfolding).**

$\mathsf{unfold}^0(T) = T$ for all T $\quad$ $\mathsf{unfold}^{1+n}(T) = \mathsf{unfold}^1(\mathsf{unfold}^n(T))$

$\mathsf{unfold}^1(k!\langle U\rangle;T) = k!\langle U\rangle;\mathsf{unfold}^1(T)$ $\quad$ $\mathsf{unfold}^1(k\oplus\{l_i\colon T_i\}_{i\in I}) = k\oplus\{l_i\colon \mathsf{unfold}^1(T_i)\}_{i\in I}$

$\mathsf{unfold}^1(k?\langle U\rangle;T) = k?\langle U\rangle;\mathsf{unfold}^1(T)$ $\quad$ $\mathsf{unfold}^1(k\&\{l_i\colon T_i\}_{i\in I}) = k\&\{l_i\colon \mathsf{unfold}^1(T_i)\}_{i\in I}$

$\mathsf{unfold}^1(\mu\mathbf{t}.T) = T[\mu\mathbf{t}.T/\mathbf{t}]$ $\quad$ $\mathsf{unfold}^1(\mathbf{t}) = \mathbf{t}$ $\quad$ $\mathsf{unfold}^1(\mathsf{end}) = \mathsf{end}$

We also use $\mathsf{unfold}^n(U)$ which is defined as $\mathsf{unfold}^n(T)$ above. [4]

For example, $\mathsf{unfold}^2(k?\langle U\rangle;\mu\mathbf{t}.k'!\langle U'\rangle;\mathbf{t}) = k?\langle U\rangle;k'!\langle U'\rangle;k'!\langle U'\rangle;\mu\mathbf{t}.k'!\langle U'\rangle;\mathbf{t}$. Note that, because our recursive types are contractive, $\mathsf{unfold}^n(T)$ terminates. We can now introduce the central notion of asynchronous communication subtyping.

Definition 2. A relation $\mathfrak{R} \in \mathit{Type}\times\mathit{Type}$ is an asynchronous type simulation if $(T_1,T_2) \in \mathfrak{R}$ implies the following conditions:

- If $T_1 = \mathsf{end}$, then $\mathsf{unfold}^n(T_2) = \mathsf{end}$.
- If $T_1 = k!\langle U_1\rangle;T_1'$, then $\mathsf{unfold}^n(T_2) \gg k!\langle U_2\rangle;T_2'$, $(T_1',T_2') \in \mathfrak{R}$ and $(U_1,U_2) \in \mathfrak{R}$.
- If $T_1 = k?\langle U_1\rangle;T_1'$, then $\mathsf{unfold}^n(T_2) \gg k?\langle U_2\rangle;T_2'$, $(T_1',T_2') \in \mathfrak{R}$ and $(U_2,U_1) \in \mathfrak{R}$.
- If $T_1 = k\oplus\{l_i : T_{1i}\}_{i\in I}$, then $\mathsf{unfold}^n(T_2) \gg k\oplus\{l_j : T_{2j}\}_{j\in J}$, $I\subseteq J$ and $\forall i\in I.(T_{1i},T_{2i}) \in \mathfrak{R}$.
- If $T_1 = k\&\{l_i : T_{1i}\}_{i\in I}$, then $\mathsf{unfold}^n(T_2) \gg k\&\{l_j : T_{2j}\}_{j\in J}$, $J\subseteq I$ and $\forall j\in J.(T_{1j},T_{2j}) \in \mathfrak{R}$.
- If $T_1 = \mu\mathbf{t}.T$, then $(\mathsf{unfold}^1(T_1),T_2) \in \mathfrak{R}$.

where a type simulation of $(U_1,U_2) \in \mathfrak{R}$ is defined as the standard bisimulation (since U is invariant).[5] The coinductive subtyping relation $T_1 \leqslant_c T_2$ (read: T_1 is an *asynchronous subtype* of T_2) is defined when there exists a type simulation $\mathfrak{R}$ with $(T_1,T_2) \in \mathfrak{R}$.

An output of T_1 can be simulated after applying asynchronous optimisation $\gg$ to the unfolded T_2. We also need to ensure object type U_1 is a subtype of U_2. For the input, we ensure U_2 is a subtype of U_1. The definitions of selection and branching subsume the traditional session branching/selection subtyping.[6] In selection a label appearing in T_1 must be included in T_2; dually, in branching a subtype T_1 must cover all branches declared in T_2. For a value type, $U_1 \leqslant_c U_2$ implies $U_2 \leqslant_c U_1$ by definition. We show examples to justify our subtyping.

Example 3. Below we write $k!$ for $k!\langle U\rangle$ and $k?$ for $k?\langle U\rangle$, omitting U.

1. Let $T_1 = \mu\mathbf{t}.k?;k'!;\mathbf{t}$, $T_2 = \mu\mathbf{t}.k'!;k?;\mathbf{t}$. Then we can prove $T_2 \leqslant_c T_1$ using the simulation $\mathfrak{R} = \{(T_2,T_1),\ (k'!;k?;T_2,T_1),\ (k?;T_2,k?;T_1)\}$. T_2 represents more optimal communications than T_1 since it can output messages at k' without waiting.
2. Let $T_2' = k'!;T_1$ which means first sending a signal at k' then repeating input-output actions. Then $T_2' \leqslant_c T_1$ by taking $\mathfrak{R} = \{(T_2',T_1),\ (T_1,k?;T_1),\ (k?;T_2',k?;T_1)\}$ as a simulation closure. Note also $T_2 \leqslant_c T_2'$ and $T_2' \leqslant_c T_2$.

[4] In [10], $\mathsf{unfold}(T)$ repeatedly unfolds consecutive top-level recursion until a guarded type is obtained. In our definition, $\mathsf{unfold}^1(T)$ expands a single recursion, not only top-level but also under guarded types.

[5] Note that G is invariant like standard channel types $\hat{}[\tilde{T}]$ [10].

[6] We follow the subtyping relation in [7, 13] whose ordering is reversed from [10] since in our judgement the session environment is declared on the right-hand side of a process.

3. Let $T_4' = k_1!;k_2!;T_3$ with $T_3 = \mu\mathbf{t}.k_3?;k_1!;k_4?;k_2!;\mathbf{t}$ and $T_4 = \mu\mathbf{t}.k_1!;k_3?;k_2!;k_4?;\mathbf{t}$. These types are extended from T_2', T_1 and T_2 with two signal messages at the top level. Then $T_4' \leqslant_c T_3$. To simulate T_4', we require nested unfold for T_3. More exactly, the intermediate type $k_1!;k_4?;k_2!;T_3$ can be simulated by $k_4?;T_3$ if T_3 unfolds and $k_1!$ under recursion appears at the top-level. Similarly for $T_4' \leqslant_c T_4$.
4. Take $T_5 = \mu\mathbf{t}_1.k_1!;\mu\mathbf{t}_2.k_1!;\&\{l_1 : k_2?;k_1!;\mathbf{t}_1,\ l_2 : k_1!;\mathbf{t}_2\}$ and let $T_6 = \mu\mathbf{t}_1.\mu\mathbf{t}_2.\&\{l_1 : k_2?;k_1!;\mathbf{t}_1,\ l_2 : k_1!;\mathbf{t}_2\}$. Then $T_5 \leqslant_c T_6$. This example is proved similarly to (3).

Note that none of the above subtyping relations, except $T_2 \leqslant_c T_2'$ and $T_2' \leqslant_c T_2$, can be derived without including $\ll$ in the typed simulation.

Before we prove that $\leqslant_c$ is a preorder, we show that there are connecting simulations relating the components of two subtyping relations. We write $T_1 \,\mathfrak{R}_1\, T_2$ for $(T_1, T_2) \in \mathfrak{R}_1$.

Lemma 4. *If $T_1 \,\mathfrak{R}_1\, T_2$ and $T_2 \,\mathfrak{R}_2\, T_3$ for type simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$ then there exists a type simulation $\mathfrak{R}_3$ such that if* $\mathsf{unfold}^n(T_2) \gg T_2'$, *then $T_2' \,\mathfrak{R}_3\, T_3$.*

Definition 5 (Transitivity connection). For simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, we say $\mathfrak{R}_3$ (from the condition in Lemma 4) is a transitivity connection of $T_1 \,\mathfrak{R}_1\, T_2$ and $T_2 \,\mathfrak{R}_2\, T_3$. We write $\mathbf{trc}(T_1 \,\mathfrak{R}_1\, T_2 \,\mathfrak{R}_2\, T_3)$ for $\mathfrak{R}_3$. We define $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ as the smallest relation such that if $(T_1, T_2) \in \mathfrak{R}_1$ and $(T_2, T_3) \in \mathfrak{R}_2$, then $\mathbf{trc}(T_1 \,\mathfrak{R}_1\, T_2 \,\mathfrak{R}_2\, T_3) \subseteq \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$.

From the definition, $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ only contains type simulations, and as the union of these it is also a type simulation. Note that the smallest relation exists, by set inclusion of relation pairs, containing all the transitivity connections of elements in $\mathfrak{R}_1/\mathfrak{R}_2$. For example, $\mathfrak{R}_3 = \mathbf{trc}(T_1 \,\mathfrak{R}_1\, T_2 \,\mathfrak{R}_2\, T_3)$ does not contain $(k!\langle U\rangle;\mathsf{end}, k?\langle U\rangle;\mathsf{end})$, which cannot be a member of any type simulation; and by set inclusion, it is smaller than $\mathfrak{R}_3 \cup (k!\langle U\rangle;\mathsf{end}, k?\langle U\rangle;\mathsf{end})$.

Theorem 6. *The relation $\leqslant_c$ is a preorder.*

Proof. Using as standard the relation $\{(T,T) \mid T \in \mathit{Type}\}$, we prove $\leqslant_c$ is reflexive. For transitivity of $\leqslant_c$, suppose $T_1 \leqslant_c T_2 \leqslant_c T_3$ and let $\mathfrak{R}_1$ and $\mathfrak{R}_2$ be type simulations with $(T_1, T_2) \in \mathfrak{R}_1$ and $(T_2, T_3) \in \mathfrak{R}_2$. To show $T_1 \leqslant_c T_3$ we need to find a type simulation $\mathfrak{R}$ such that $(T_1, T_3) \in \mathfrak{R}$. Define $\mathfrak{R}$ as $(\mathfrak{R}_1 \cdot \mathfrak{R}_2) \cup (\mathfrak{R}_1 \cdot \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2))$. Clearly $(T_1, T_3) \in \mathfrak{R}$, and it remains to show that $\mathfrak{R}$ is a type simulation. For any $(T, T'') \in \mathfrak{R}$, there are two cases (relations above), and six subcases (simulation rules). For $(U, U') \in \mathfrak{R}$, the result is easy as U types are invariant. We only show one of the most interesting cases.

Suppose $(T, T'') \in \mathfrak{R}_1 \cdot \mathfrak{R}_2$ and $T = k!\langle U_1\rangle;T_1$. Then there exists $(T, T') \in \mathfrak{R}_1$ and $(T', T'') \in \mathfrak{R}_2$. By the definition of type simulation, we have $\mathsf{unfold}^n(T') \gg k!\langle U_1'\rangle;T_1'$ and $(U_1, U_1') \in \mathfrak{R}_1$ and $(T_1, T_1') \in \mathfrak{R}_1$. Let $\mathbf{trc}(T \,\mathfrak{R}_1\, T' \,\mathfrak{R}_2\, T'') = \mathfrak{R}_3 \subseteq \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$, then by Lemma 4 we obtain $(k!\langle U_1'\rangle;T_1', T'') \in \mathfrak{R}_3$, and by the definition of simulation we have $\mathsf{unfold}^m(T'') \gg k!\langle U_1''\rangle;T_1''$ and $(U_1', U_1'') \in \mathfrak{R}_3$ and $(T_1', T_1'') \in \mathfrak{R}_3$. Finally, by the definition of $\mathfrak{R}_1 \cdot \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$, $(U_1, U_1'') \in \mathfrak{R}$ and $(T_1, T_1'') \in \mathfrak{R}$ as required. Other cases are similar. $\square$

3.3 Algorithmic Asynchronous Subtyping

The algorithmic subtyping of session types is studied in [10, § 5.1]. Due to the incorporation of asynchronous permutation and n-time unfolding in the type simulation, we need the bound of unfolding for constructing a terminating algorithm. We first list some selected rewriting rules $\overset{k}{\mapsto}$ which move the types with channel k to the head applying the rules of $\gg$ in Figure 1 in the reverse direction.

$$(\mathrm{OI})\quad k'?\langle U'\rangle;k!\langle U\rangle;T \overset{k}{\mapsto} k!\langle U\rangle;k'?\langle U'\rangle;T \qquad (\mathrm{Tr})\ \frac{T_1 \overset{k}{\mapsto} T_2 \quad T_2 \overset{k}{\mapsto} T_3}{T_1 \overset{k}{\mapsto} T_3}$$

$$(\mathrm{CO})\ \frac{T \overset{k}{\mapsto} T}{k'!\langle U\rangle;T \overset{k}{\mapsto} k'!\langle U\rangle;T'} \qquad (\mathrm{CB})\ \frac{T_j \overset{k}{\mapsto} T_j'}{k'\&\{l_1:T_1,..,l_j:T_j,..\} \overset{k}{\mapsto} k'\&\{l_1:T_1,..,l_j:T_j',..\}}$$

We omit the similar rules for (OB–SS), (CI,CS), which are defined similarly to (OI) and (CO,CB). Note that we do not define $\overset{k}{\mapsto}$ for (E) and (M). (CO,CB) are for congruency. For a simple example, let $T_0 = k\oplus\{l_1 : k_1?\langle U_1\rangle;k_2!\langle U_2\rangle;\mathsf{end},\ l_2 : k_2!\langle U_2\rangle;\mathsf{end}\}$. Then $T_0 \overset{k_2}{\mapsto} k\oplus\{l_1 : k_2!\langle U_2\rangle;k_1?\langle U_1\rangle;\mathsf{end},\ l_2 : k_2!\langle U_2\rangle;\mathsf{end}\} \overset{k_2}{\mapsto} k_2!\langle U_2\rangle;k\oplus\{k_1?\langle U_1\rangle;\mathsf{end}, l_2 : \mathsf{end}\}$ by (CS,OS). We can easily show $\overset{k}{\mapsto}$ is confluent and terminates, and $T \overset{k}{\mapsto} T'$ implies $T' \ll T$. We can also prove if $T \ll T'$, then we always have $T' \overset{k_1}{\mapsto} \cdot \overset{k_2}{\mapsto} \cdots \overset{k_n}{\mapsto} T$ where $k_1k_2..k_n$ are a (possibly empty) subsequence of channels occurring at the top-level in T with this order (e.g., $k_1k_2k_3k_4$ if $k_1!;k_2\oplus\{l_1 : k_3!,\ l_2 : k_4!\}$). Hence:

Proposition 7. *Given T and T', $T \ll T'$ is decidable.*

The derivability of judgement $\Sigma \vdash T \leqslant T'$ is defined in Figure 2 where Σ is a sequence of assumed instances of the subtyping relation. We use n-hole type contexts ($\mathscr{T}, \mathscr{T}', ...$) where $[\ ]^{h\in H}$ denotes a hole with index h.

$$\mathscr{T} \ ::=\ [\ ]^{h\in H} \mid k!\langle U\rangle;\mathscr{T} \mid k?\langle U\rangle;\mathscr{T} \mid k\oplus\{l_i : \mathscr{T}_i\}_{i\in I} \mid k\&\{l_i : \mathscr{T}_i\}_{i\in I}$$

For example, with $H = \{1,2\}$ and $\mathscr{T} = k\oplus\{l_1 : k_1?\langle U_1\rangle;[\,]^{1\in H},\ l_2 : [\,]^{2\in H}\}$, we have $\mathscr{T}[T_i]^{i\in H} = k\oplus\{l_1 : k_1?\langle U_1\rangle;T_1,\ l_2 : T_2\}$. A hole in $\mathscr{T}$ does not appear under recursion since $\ll$ permutes top-level actions only. We also use (1) function $\mathsf{top}(T)$ which returns the channel at the head of T and (2) function $\mathsf{depth}\langle k,T\rangle$ to calculate how many unfoldings are needed for k to appear at the top-level. If k does not appear in T, $\mathsf{depth}\langle k,T\rangle$ is undefined. When $\mathsf{depth}\langle k,T\rangle$ is defined, $\mathsf{depth}\langle k,T\rangle$ is finite.

$$\begin{array}{l}
\mathsf{top}(\mathsf{end}) = \bullet \quad \mathsf{top}(k?\langle U\rangle;T) = \mathsf{top}(k!\langle U\rangle;T) = \mathsf{top}(k\&\{l_i\colon T_i\}_{i\in I}) = \mathsf{top}(k\oplus\{l_i\colon T_i\}_{i\in I}) = k\\
\mathsf{depth}\langle k,T\rangle = 0 \quad \text{if } \mathsf{top}(T) = k \quad \mathsf{depth}\langle \bullet,\mathsf{end}\rangle = 0\\
\mathsf{depth}\langle k,k'?\langle U\rangle;T\rangle = \mathsf{depth}\langle k,k'!\langle U\rangle;T\rangle = \mathsf{depth}\langle k,T\rangle \quad k\neq k'\\
\mathsf{depth}\langle k,k'\&\{l_i\colon T_i\}_{i\in I}\rangle = \mathsf{depth}\langle k,k'\oplus\{l_i\colon T_i\}_{i\in I}\rangle = \max_{i\in I}(\mathsf{depth}\langle k,T_i\rangle) \quad k\neq k'\\
\mathsf{depth}\langle k,\mu\mathbf{t}.T\rangle = \mathsf{depth}\langle k,T[\mu\mathbf{t}.T/\mathbf{t}]\rangle + 1 \qquad \mathsf{depth}\langle \bullet,\mu\mathbf{t}.T\rangle = \mathsf{depth}\langle \bullet,T[\mu\mathbf{t}.T/\mathbf{t}]\rangle + 1
\end{array}$$

In Figure 2, [ASMP,END] are standard. In [OUT], we fix the subtype and apply $\overset{k}{\mapsto}$ to place $k!\langle U\rangle$ to the top level. Then we check the tail of the result of rewriting $\mathscr{T}[T'_{2h}]^{h\in H}$ is a subtype of T_1 (the rule subsumes the case $k!\langle U\rangle$ already at the top level). Rule [SEL]

$$[\textsc{Asmp}]\,\frac{T \leqslant T' \in \Sigma}{\Sigma \vdash T \leqslant T'} \qquad [\textsc{End}]\,\frac{-}{\Sigma \vdash \mathsf{end} \leqslant \mathsf{end}}$$

$$[\textsc{Out}]\,\frac{\Sigma \vdash U_1 \leqslant U_2 \quad \Sigma \vdash T_1 \leqslant \mathscr{T}[T'_{2h}]^{h\in H} \quad \mathscr{T}[k!\langle U_2\rangle;T_{2h}]^{h\in H} \overset{k}{\mapsto} k!\langle U_2\rangle;\mathscr{T}[T'_{2h}]^{h\in H}}{\Sigma \vdash k!\langle U_1\rangle;T_1 \leqslant \mathscr{T}[k!\langle U_2\rangle;T_{2h}]^{h\in H}}$$

$$[\textsc{Sel}]\,\frac{\forall i \in I.\Sigma \vdash T_i \leqslant \mathscr{T}[T''_{ih}]^{h\in H} \;\; \mathscr{T}[k\oplus\{l_i : T'_{ih}\}_{i\in J}]^{h\in H} \overset{k}{\mapsto} k\oplus\{l_i : \mathscr{T}[T''_{ih}]^{h\in H}\}_{i\in J} \; I\subseteq J}{\Sigma \vdash k\oplus\{l_i : T_i\}_{i\in I} \leqslant \mathscr{T}[k\oplus\{l_i : T'_{ih}\}_{i\in J}]^{h\in H}}$$

$$[\textsc{RecL}]\,\frac{\Sigma, \mu\mathbf{t}.T \leqslant T' \vdash \mathsf{unfold}^1(\mu\mathbf{t}.T) \leqslant T'}{\Sigma \vdash \mu\mathbf{t}.T \leqslant T'} \qquad [\textsc{RecR}]\,\frac{\begin{array}{c} n = \mathsf{depth}\langle \mathsf{top}(T), T'\rangle \quad n \geq 1 \\ \Sigma, T \leqslant T' \vdash T \leqslant \mathsf{unfold}^n(T')\end{array}}{\Sigma \vdash T \leqslant T'}$$

Fig. 2. Algorithmic Subtyping Rules

is similarly defined. Rule [RecL] is standard. Rule [RecR] unfolds T' until a type with the same channel as the top of T appears at the top-level. The rule for input/branching is defined like [Out]/[Sel], respectively.

The rules give an algorithm for checking *the algorithmic subtyping relation* $\leqslant$ (by reading these rules from upwards). As usual, [Asmp] should always be used if it is applicable, and when both [RecL] and [RecR] are applicable, [RecL] is used in preference to [RecR]. Similarly, other rules are applied in preference to [RecR], which can only be applied if the top of T does not appear at the top level of T'. As an example, let $T_1 = k\oplus\{l_1 : k_1?\langle U_1\rangle;\mathsf{end}\}$. Then we can derive $k_2!\langle U_2\rangle;T_1 \leqslant T_0$ (T_0 is given above) by using [Out]. At the top level, the algorithm is applied to the initial goal $\emptyset \vdash T \leqslant T'$ (which we often write $T \leqslant T'$).

Lemma 8. *1. The subtyping algorithm always terminates.*
2. If $T \leqslant_c T'$ then the algorithm does not return false when applied to $\Sigma \vdash T \leqslant T'$.

The proof uses techniques related to those developed in [10]; the main differences are, for the proof of (1), we have to take the subterms up to $\ll$ with the finite number of unfolding when we argue the size of Σ cannot increase without bound. However since $\ll$ does not change the size of the judgement (defined in [10, Lemma 10]), we can prove (1). The proof of (2) is standard from (1).

Theorem 9 (Soundness and Completeness of the Algorithmic Subtyping). *For all closed types T and T', $T \leqslant_c T'$ if and only if $T \leqslant T'$.*

The if-direction is by Lemma 8 (2) and the only-if direction by constructing a relation following [10, Theorem 4].

3.4 Local Asynchronous Commutative Session Typing

The type judgement for end-point processes is of the shape $\Gamma \vdash P \triangleright \Delta$ which reads: "under the environment Γ, process P has typing Δ" where environments are defined as:

$$\Gamma \;::=\; \emptyset \mid \Gamma, u : U \mid \Gamma, X : \tilde{U}\tilde{T} \qquad \Delta \;::=\; \emptyset \mid \Delta, \tilde{s} : \{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I}$$

A *sorting* $(\Gamma, \Gamma', ..)$ is a finite map from names to value types and from process variables to sequences of value types and session types. *Typing* $(\Delta, \Delta', ..)$ records linear usage of session channels. $T\,@\mathtt{p}$ is called *located type* which means T is a session type of the participant $\mathtt{p}$. In multiparty sessions, it assigns a family of located types to a vector of session channels. The typing system is identical with [13]: we only have to add the subsumption rule: i.e. $\Gamma \vdash P \triangleright \Delta$ and $\Delta \leqslant \Delta'$ then $\Gamma \vdash P \triangleright \Delta'$ where $\Delta \leqslant \Delta'$ is defined by pointwise application of $\leqslant$.

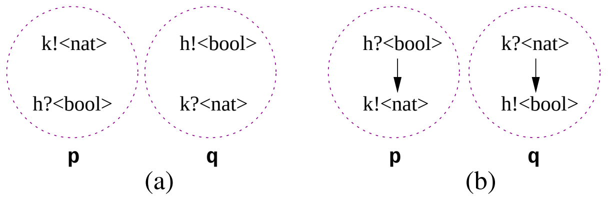
Theorem 10 (Subject Congruence and Reduction). *$\Gamma \vdash P \triangleright \emptyset$ and $P \equiv Q$ imply $\Gamma \vdash Q \triangleright \emptyset$; and $\Gamma \vdash P \triangleright \emptyset$ and $P \longrightarrow Q$ imply $\Gamma \vdash Q \triangleright \emptyset$.*

The proof follows the same routine as in [13], but we must take care that all permutations defined by $\ll$ do not affect the input-output causal dependencies of the global types. We can also obtain the other three key proprieties, communication-safety, session-fidelity and progress as stated in [13, § 5]. The rest of the paper can be read without knowing the details of a typing system.

4 Principal Global Typing through Graph-based Types

Why graph-based types. Let $P \stackrel{\text{def}}{=} a[\mathtt{p}](\tilde{s}).s_1!\langle 3\rangle; s_2?(x)$ and $Q \stackrel{\text{def}}{=} \overline{a}[\mathtt{p}](\tilde{s}).s_2!\langle \mathsf{true}\rangle; s_1?(y)$ where P is a participant named by $\mathtt{p}$ and Q is an initiator named by $\mathtt{q}$. Then P and Q are typable under essentially only two global types, $G = \mathtt{p} \to \mathtt{q} : k\langle \mathsf{int}\rangle; \mathtt{q} \to \mathtt{p} : h\langle \mathsf{bool}\rangle; \mathsf{end}$ and $G' = \mathtt{q} \to \mathtt{p} : h\langle \mathsf{bool}\rangle; \mathtt{p} \to \mathtt{q} : k\langle \mathsf{int}\rangle; \mathsf{end}$. Note the projection of G to $\mathtt{p}$ is $\leqslant$-minimal for P (i.e. other local types of P can be derived by subsumption): but this is not true for its projection to $\mathtt{q}$. Similarly G' does not give a minimal type for P. Thus there is no "best" global type for $P|Q$: this is because interactions between P and Q take place in a criss-crossing way: the syntax of global types, which can only represent tree-like causality, is too rigid to represent such a situation.

Local and global graphs. A *local graph* is a (finite or infinite) directed graph where each node, called *action*, is labelled by one of $k?\langle U\rangle$ (input), $k!\langle U\rangle$ (output), $k\&[l_i]_{i\in I}$ (branching), $k \oplus [l_i]_{i\in I}$ (selection) and $k \oplus l$ (label-output [1]); and, for edges: (1) each edge from $k\&[l_i]_{i\in I}$ or $k \oplus [l_i]_{i\in I}$ is labelled by one of $\{l_i\}$; and (2) $k \oplus l$ (resp. $k!\langle U\rangle$) has a unique outgoing edge, and its target is always an output/selection/label-output at k. A *global graph for participants* $\{\mathtt{p}_1, .., \mathtt{p}_n\}$, written $\mathscr{G}, \mathscr{G}', \ldots$, is a disjoint union of an $\{\mathtt{p}_1, .., \mathtt{p}_n\}$-indexed family of local graphs. Given $\mathscr{G}$ for $\{\mathtt{p}_1, .., \mathtt{p}_n\}$, its $\mathtt{p}_i$-*component* is the local graph in $\mathscr{G}$ indexed by $\mathtt{p}_i$. A node is *active* if it has no incoming edges.



(a) (b)

In (a) above, we show a global graph for P and Q given above, consisting of two local graphs (balloons labelled by $\mathtt{p}$ and $\mathtt{q}$), each with an output, an input and no edges. If we add an edge from input to output in each local graph, we get the global graph (b) for $a[\mathrm{p}](\tilde{s}).s_2?(x);s_1!\langle 3\rangle$ and $\overline{a}[\mathrm{p}](\tilde{s}).s_1?(y);s_2!\langle\mathsf{true}\rangle$, which now deadlocks.

Linearity, progress and coherence. We equip global graphs with a notion of reduction which abstracts that of processes. Below we write $\mathscr{E}[\,\cdot\,]..[\,\cdot\,]$ for a global graph with one or more holes, each of which is to be filled with a sub-graph of a local graph, such that all holes are active, i.e. have no incoming edges.

$$\mathscr{E}[k?\langle U\rangle][k!\langle U\rangle] \longrightarrow \mathscr{E}[\emptyset] \qquad \mathscr{E}[k\&[l_i:\mathscr{G}_i]_{i\in I}][k\oplus l_j] \longrightarrow \mathscr{E}[\mathscr{G}_j][\emptyset] \;\; (j\in I)$$
$$\mathscr{E}[k\oplus[l_i:\mathscr{G}_i]_{i\in I}] \longrightarrow \mathscr{E}[k\oplus l_i;\mathscr{G}_i]$$

Above $\emptyset$ is the empty graph. In each rule, the replacement in the hole(s) entails taking off *both* the old graph *and* all the outgoing edge(s) from it and filling the hole with the new graph. A reduction by the first two rules is called *communication at k*. In the second rule, $k\&[l_i:\mathscr{G}_i]_{i\in I}$ is the disjoint union of $k\&[l_i]_{i\in I}$ and $\{\mathscr{G}_i\}_{i\in I}$ together with, for each $i\in I$, l_i-labelled edges from $k\&[l_i]_{i\in I}$ to all the active actions in $\mathscr{G}_i$. In the third (from [1]), $k\oplus[l_i:\mathscr{G}_i]_{i\in I}$ is as $k\&[l_i:\mathscr{G}_i]_{i\in I}$ while $k\oplus l_i;\mathscr{G}_i$ is the disjoint union of $k\oplus l_i$ and $\mathscr{G}_i$ with edges from the former to the active output/selection/label-outputs at k in $\mathscr{G}_i$. A global graph $\mathscr{G}$ is *linear* when for each $\mathscr{G}'$ such that $\mathscr{G}\longrightarrow^*\mathscr{G}'$, if $\mathscr{G}'$ has two active actions at k, a reduction at k is possible, and no other active action shares k. A global graph $\mathscr{G}$ has *progress* when for each $\mathscr{G}'$ such that $\mathscr{G}\longrightarrow^*\mathscr{G}'$, either $\mathscr{G}'$ reduces or it is empty. Finally we say $\mathscr{G}$ is *coherent* when it is linear and has progress.

Coherent global graphs from local types. A local graph is constructed from a local type as the latter's regular tree representations. Given $\Delta=\{T_i@\mathrm{p}_i\}_{i\in I}$, this immediately gives the global graph for $\{\mathrm{p}_i\}_{i\in I}$, which we write $[\![\Delta]\!]$. The coherence of $[\![\Delta]\!]$ is decidable, as we outline below.

We first check Δ is *well-directed* in the sense that each channel in Δ is used by two and only two participants and moreover one of them uses it only for input/branching and the other only for output/selection/label-output, which can be checked by going through Δ once. For well-directed Δ, there is an algorithm to ensure linearity of $[\![\Delta]\!]$, by checking if each pair of participants in Δ are compatible in their type structures, closely following the algorithmic subtyping in § 3.2.

Through the validation of compatibility of Δ, we can equip $[\![\Delta]\!]$ with the additional *communication edges*, from each output/selection/label-output to its potentially interacting action(s), representing potential redexes. Using this added set of edges, we reduce the progress of $[\![\Delta]\!]$ to the acyclicity of its paths consisting of its local and communication edges, completely characterising progress under linearity. The acyclicity of $[\![\Delta]\!]$ is then reducible to that of its initial finite sub-graph. Because linearity and compatibility are equivalent under progress, we obtain:

Theorem 11 (complete algorithmic characterisation of coherence). *Let $\Delta=\{T_i@\mathrm{p}_i\}_{i\in I}$ be well-directed. Then the coherence of $[\![\Delta]\!]$ with Δ given as input is decidable.*

Principal global typing through global graphs. Any projectable global type G for participants say $\{\mathrm{p}_i\}_{i\in I}$ is equivalent to its projections $\Delta=\{(G\restriction\mathrm{p}_i)@\mathrm{p}_i\}_{i\in I}$, and because such Δ is immediately compatible and acyclic, we can regard G as a coherent global graph. This motivates the use of coherent global graphs instead of global types in the type discipline, presenting $[\![\Delta]\!]$ as Δ itself.[7] By replacing global types with coherent global graphs in types and typing rules, we obtain a new type discipline. We write $\Gamma \vdash_{\mathrm{g}} P \triangleright \Delta$ for typability in this new discipline (subsumption is consistent because if $[\![\Delta]\!]$ is coherent and Δ' is point-wise $\leqslant$-smaller than Δ then $[\![\Delta']\!]$ is also coherent). By identifying $\mathscr{G}$ as the corresponding coherent global graph, $\Gamma \vdash P \triangleright \Delta$ implies $\Gamma \vdash_{\mathrm{g}} P \triangleright \Delta$. Further, since linearity and progress of $[\![\Delta]\!]$ are reflected onto the dynamics of typed processes (precisely following the arguments in [13]), the typability in $\vdash_{\mathrm{g}}$ ensures communication safety and progress.

For the principal typing property, we add the $\leqslant$-least element $\perp$ to the set of local types; $\perp$ is also used as local graph occurring in global graphs (where intuitively $\perp$ denotes a placeholder for a local behaviour). The coherence and other notions for global graphs are defined ignoring $\perp$. Without loss of practical generality we assume each shared name say a has a fixed *arity* which is the number of participants for a potential session established through a; and that processes are type-annotated on bound variables and free object names in the standard way. Through local type inference [6, 15] using the point-wise join of coherent global types (calculated as in algorithmic subtyping), together with Theorem 11, we obtain a principal global typing property. Below we write $\Gamma' \leqslant \Gamma$ for $\mathsf{dom}(\Gamma') \subset \mathsf{dom}(\Gamma)$ and $\Gamma'(a) \leqslant \Gamma(a)$ for each $a \in \mathsf{dom}(\Gamma')$. We say P is *closed* if it has no free session channels nor free variables.

Theorem 12 (principal global typing). *Let P be closed.* (1) *The typability of P with respect to* $\vdash_{\mathrm{g}}$ *is decidable.* (2) *If P is typable then P has a* principal global typing Γ_0 *in the sense that* $\Gamma_0 \vdash_{\mathrm{g}} P \triangleright \emptyset$ *holds and moreover* $\Gamma \vdash_{\mathrm{g}} P \triangleright \emptyset$ *implies* $\Gamma_0 \leqslant \Gamma$.

5 Application: Double-Buffering Algorithm

This section illustrates the use of our type theory using the *double-buffering algorithm* [21], a basic distributed algorithm widely used in stream/media processing and high-performance and multicore computing, presenting how the two strategies discussed in Introduction can be applied through the theories presented in the previous sections.

The purpose of the double-buffering algorithm is to transform a large amount of data, where a series of chunks of data are transferred from a source (Source) to a transformer (called Kernel), gets processed there and delivered to a sink (Sink). Under potential temporal variations in processing and communication time, it is necessary to synchronise among these three parties through message passing. However a naive, and obviously safe, protocol leads to a highly sequential, non-optimal distributed algorithm. Thus it is beneficial to increase asynchrony of local programs without violating the shared protocol. We show the outline of an application of our theories to achieve this goal, starting from a sequential and safe global protocol to optimised local protocols through asynchronous communication subtyping, with a formal safety guarantee.

[7] To be precise, we regard Δ up to the type isomorphism corresponding to $\leqslant$; and we take off, from each branching type, its branches (if any) which never get invoked in any reduction path: such "garbage" branches are precisely identified during the validation of coherence.

Global Type : $G =$

$$\mu\mathbf{t}.(\ \mathsf{K} \to \mathsf{So} : r_1\langle\rangle;\ \ \mathsf{K} \to \mathsf{So} : r_2\langle\rangle;\ \ \mathsf{So} \to \mathsf{K} : s_1\langle U\rangle;\ \ \mathsf{So} \to \mathsf{K} : s_2\langle U\rangle;\ \ \mathsf{Si} \to \mathsf{K} : t_1\langle\rangle;\ \ \mathsf{Si} \to \mathsf{K} : t_2\langle\rangle;\ \ \mathsf{K} \to \mathsf{Si} : u_1\langle U\rangle;\ \ \mathsf{K} \to \mathsf{Si} : u_2\langle U\rangle;\mathbf{t})$$

Projected Local Type of Kernel :

$$T = \mu\mathbf{t}.r_1!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;\ r_2!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;\mathbf{t}$$

Local Type of Kernel :

$$T^\star = r_1!\langle\rangle;r_2!\langle\rangle;\ \mu\mathbf{t}.s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_1!\langle\rangle;\ s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;r_2!\langle\rangle;\mathbf{t}$$

```
Source:
$a[1](r_1 r_2 s_1 s_2 t_1 t_2 u_1 u_2)$.
$\mu X$.(
  .. // assign data to $y[1..n]$
  $r_1?(()); s_1!\langle y[1..n]\rangle$;
  .. // assign data to $y[1..n]$
  $r_2?(()) ;s_2!\langle y\rangle;X$)
```

```
Sink:
$a[2](r_1 r_2 s_1 s_2 t_1 t_2 u_1 u_2)$.
$\mu X$.(
  $t_1!\langle\rangle; u_1?(z)$;
  .. // print $z[1..n]$
  $t_2!\langle\rangle; u_2?(z)$;
  .. // print $z[1..n]$
  $X$)
```

```
Kernel:
$\overline{a}[1,2](r_1 r_2 s_1 s_2 t_1 t_2 u_1 u_2)$.
$r_1!\langle\rangle; r_2!\langle\rangle$;
$\mu X$.(
  $s_1?(x_A)$;
  .. // repeat:
  .. // $x_A[i]$ ::= $x_A[i]\oplus$key
  .. // key::= $x_A[i]$
  $t_1?(()); u_1!\langle x_A\rangle; r_1!\langle\rangle$;
  $s_2?(x_B)$;
  .. // repeat:
  .. // $x_B[i] = x_B[i] \oplus key$
  .. // $key = x_B[i]$
  $t_2?(()); u_2!\langle x_B\rangle; r_2!\langle\rangle; X$
)
```

Fig. 3. Double-Buffering Algorithm: Processes and Types

Top-down approach (1): global type. The development of programs starts from the global type G on the left-most column in Figure 3. So, K and Si denote participant names for Source, Kernel and Sink. U denotes a large int-array type. Assuming Kernel will use two channels and the associated arrays for potential parallelism, the global type G starts from a recursion, describing an infinite loop. In the loop, Kernel first notifies Source via $r_{1,2}$ that it is ready to receive data in its two channels ($s_{1,2}$, with signal at r_i saying s_i is ready); Source complies, sending two chunks of data sequentially via $s_{1,2}$. Then Kernel (internally processes data and) waits for Sink to inform (via $t_{1,2}$) that Sink is ready to receive data via $u_{1,2}$: upon receiving the signals, Kernel sends the two chunks of processed data to Sink. This protocol is sequential but is safe and deadlock-free.

Top-down approach (2): local type and its refinement. Just below the global type G, Figure 3 gives the local type T of Kernel as directly projected from the global type. Our purpose is to refine T so that (1) the new local protocol is more asynchronous, allowing overlap of communication and computation [9, 11]; and (2) it still conforms to G — Kernel with the new optimised protocol will safely interact with Source and Sink who conform to the original global type G. For this purpose the developer may come up with a more asynchronous $T^\star$, given in Figure 3 after T. In this refined protocol, Kernel notifies Source via both $r_{1,2}$, but only once before entering the loop, allowing Source to start its work. Now inside the loop, the refined protocol dictates Kernel first receives data via its first channel s_1 with Source, processes the data and sends out the result to Sink via its first channel u_1 with Sink and *immediately notifies Source via r_1 that it's ready in its first channel*, allowing Source to start sending data early. Kernel then repeats the same work for its second channels with Source and Sink. In this way, Kernel can process data it has already received in one channel while it is receiving data in the other, noting it can take time for large data to sent, transferred and received.

We now show this optimised local protocol is safe w.r.t. other participants conforming to G, through the asynchronous communication subtyping. The justification uses nested unfolding.We start from unfolding T once to match r_1, r_2 of $T^\star$ as $\mathsf{unfold}^1(T) = r_1!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_2!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T$. Then $r_1!\langle\rangle$ matches $T^\star$. To simulate $r_2!\langle\rangle$ of $T^\star$, $r_2!\langle\rangle$ is permuted by $\ll$. Let $T^\star = r_1!\langle\rangle;r_2!\langle\rangle;T^\star_R$. Thus $\mathsf{unfold}^1(T^\star_R)$ must be simulated by $T' = s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T$. However to simulate $r_1!\langle\rangle$ in $\mathsf{unfold}^1(T^\star_R)$, T *must be unfolded again* since the types in the guarded position of T' do *not* include $r_1!\langle\rangle$. By [RecR], it now suffices to solve the following:

$$r_1!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;r_2!\langle\rangle;T^\star_R \;\leqslant\; s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;\mathsf{unfold}^1(T).$$

For this we apply [In,Out] of $\leqslant$ with $\ll$, reaching the assumption in Σ in [Asmp].

Top-down approach (3): code development. Figure 3 depicts the skeleton of the three (final) programs which conform to the global type. All participants initiate the session at a in the first line. We only illustrate the behaviour of Kernel, considering a simple transformation for stream encryption. Kernel, after initialising its variables including the initial key value, signals to Source that its buffers are both empty, via r_1 and r_2: then enters the main loop, where it does the following: it first receives the datum at x_A via s_1, goes through the buffer taking the XOR element-wise with key, after which it waits for Sink's cue via t_1 (which may have already arrived asynchronously), and finally sends out the buffer content to Sink via u_1, and tells Source it's ready at A via r_1: then similarly works for the second buffer (given in the next column). Unlike Source and Sink, the behaviour of Kernel does not conform to the projection of G to Kernel (T): however $T^\star$ does type-abstract its behaviour directly and because $T^\star \leqslant T$ by the argument above, Kernel does type-check under T with subsumption, hence under G.

Integration with bottom-up approach. The development process described above can be effectively and seamlessly integrated with the "bottom-up" strategy discussed in Introduction through the type inference (synthesis) of global types in § 4, which allows developers to directly refine programs and to synthesise a new global protocol reflecting the refinement, incrementally validating compatibility. This added flexibility is useful since, in actual development, programmers may often directly work on programs rather than starting from refinement of local protocols.

Further examples showing the applicability of permutation of branching/selections to parallel algorithms [14] can be found in the full version.

6 Related Work

Branching/selection subtyping in session types is first studied in [10] for binary session types. We use their syntactic approach for defining a type-simulation, but a significant extension from their technique is needed due to the incorporation of $\ll$ and nested unfoldings, which makes the proof of transitivity delicate and challenging. An initial idea of asynchronous communication subtyping for binary sessions is presented in an unpublished manuscript [18], where the treatment for recursive types and branching/selection types is left open. A recent work in a technical report [17] demonstrates a subtyping rule similar to our (OI) rule is useful for an object calculus with asynchronous binary sessions, with an iso-recursive system. It is an interesting future work to extend to the

HOπ-calculus [16] where a careful formulation for the algorithmic subtyping would be required in the presence of arrow types. The top-down approach in multiparty session types is first studied in [13], but a local refinement (asynchronous subtyping) is not proposed there. The problem of synthesising a global specification from endpoint behaviours has been a lingering question since the inception of the notion of global descriptions for business protocols [24], being posed as an open problem in [1, 2, 13]. Inference of principal types is studied in [15] for binary session types (note in binary sessions the issue of global synthesis does not arise). The present work gives clear and general solutions to these extant technical problems.

In the context of multiparty session types, a typing system for a strong progress property is studied in [1]. Asynchronous communication subtyping can be smoothly applied to [1]. For delegations, the main definitions of $\ll$, $\leqslant_c$ and $\leqslant$ stay as the same, but proofs need to be revised to treat nesting types; and for the principal typing, $\perp$ should be added into a carried type in global graphs. The study of formal theories of contracts are studied in [8] using CCS-like processes as a type representation. The work [19] extends [8] with the treatment of asynchronous behaviours using orchestrators, through the use of bounded buffers that control message flows between a client and servers. Our own system in [7] developed a theory in which a global specification arises as a programming language itself.

Conformance and refinement based on agreement of service specifications is studied in [3], using a synchronous CCS-based calculus as a contract language, and testing-preorders to check subcontract compliance. Neither type-checking of end-point processes using projected contracts (in our case, Theorem 9) nor a bottom-up strategy is presented there. The work [5] proposes a distributed calculus with sessions, incorporating the merging of running sessions. Another work [23] presents a calculus for service orientations by extending the π-calculus with context-sensitive interactions, equipped with service and request primitives and local exceptions. These preceding works do not treat the main technical problems addressed in the present work – the asynchronous communication subtyping, type-based local refinement/conformance, and a derivation of the minimum global types, backed-up by the efficient type-checking and inference algorithms, ensuring strong safety properties based on the session type discipline.

Acknowledgements. We thank the reviewers for their useful comments, Gary Brown and Steve Ross-Talbot for discussions on the potential applications of the presented framework for software development, and Matthew Rawlings for discussions on the practical significance of asynchrony in financial protocols. We generalised $\ll$ with input commutativity following a suggestion by Raymond Hu. The work is partially supported by EPSRC GR/T03208, GR/T03215, EP/F002114, EP/F003757 and IST2005-015905 MOBIUS.

References

1. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
2. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.

3. M. Bravetti and G. Zavattaro. A theory for strong service compliance. In *COORDINATION'07*, volume 4467 of *LNCS*, pages 96–112. Springer, 2007.
4. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, volume 4829 of *LNCS*, pages 34–50, 2007.
5. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.
6. M. Carbone, K. Honda, and N. Yoshida. A theoretical basis of communication-centered concurrent programming. To appear as a WS-CDL working report, `www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf`.
7. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17, 2007.
8. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL*, pages 261–272, 2008.
9. D. Culler et al. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.
10. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
11. M. Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
14. T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2005.
15. L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION*, volume 5052 of *LNCS*, pages 216–231. Springer, 2008.
16. D. Mostrous and N. Yoshida. Two Sessions Typing Systems for Higher-Order Mobile Processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.
17. D. Mostrous and N. Yoshida. A Session Object Calculus for Structured Communication-Based Programming. Technical report, Imperial College London, 2008. `www.doc.ic.ac.uk/~mostrous`.
18. M. Neubauer and P. Thiemann. Session Types for Asynchronous Communication. Universität Freiburg, 2004.
19. L. Padovani. Contract-directed synthesis of simple orchestrators. In *CONCUR*, volume 5201 of *LNCS*, pages 131–146, 2008.
20. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
21. J. C. Sancho and D. J. Kerbyson. Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, April 14–18, 2008.
22. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
23. H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283, 2008.
24. Web Services Choreography Working Group. Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.