Session-Based Communication Optimisation for Higher-Order Mobile Processes[⋆]

Dimitris Mostrous and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. In this paper we solve an open problem posed in our previous work on asynchronous subtyping [12], extending the method to higher-order session communication and functions. Our system provides two complementary methods for communication code optimisation, mobile code and asynchronous permutation of session actions, within processes that utilise structured, typed communications. In order to prove transitivity of our coinductive subtyping relation, we uniformly deal with type-manifested asynchrony, linear functional types, and contravariant components in higher-order communications. For the runtime system we propose a new compact formulation that takes into account stored higher-order values with open sessions, as well as asynchronous commutativity. In spite of the enriched type structures, we construct an algorithmic subtyping system, which is sound and complete with respect to the coinductive subtyping relation. The paper also demonstrates the expressiveness of our typing system with an e-commerce example, where optimised processes can interact respecting the expected sessions.

1 Introduction

Sessions [7, 16] have emerged as a tractable and expressive theoretical substrate, which offers direct language and protocol support [9, 17, 18] for high-level, type-safe and uniform abstraction for a wide range of communication patterns. *Session types* enable static validation assuring both *type* and *communication-safety* — not only is the value of each message correctly typed, but the sequence of messages are sent and received according to the scenario specified by the session type, precluding communication mismatch. Session primitives can be smoothly integrated with traditional *subtyping* of object and functional languages, to obtain a more flexible behavioural composition [5]. Our recent work [12] developed a new subtyping, *asynchronous subtyping*, that characterises compatibility between classes of permutations of communications within asynchronous protocols, offering much greater flexibility. However, an open problem remained: *how to uniformly introduce communication optimisations in the presence of code mobility [11], incorporating higher-order sessions and functions into the asynchronous subtyping [12, § 6]*. This is the question we address in this paper.

Higher-Order Processes with Asynchronous Sessions. We develop a session typing system for the Higher-order π-calculus [15], an amalgamation of call-by-value λ-calculus and π-calculus, extending [11]. Code mobility is facilitated by sending not just

[⋆] The work is supported by EPSRC GR/T03208, GR/T03215 and IST2005-015905 MOBIUS.

ground values and channels, but also abstracted processes that can be received and activated locally, reducing the number of transmissions of remote messages. The simplest code mobility operations are sending a thunked process $\ulcorner P \urcorner$ via channel s (denoted as $s!\langle\ulcorner P \urcorner\rangle$), and receiving and running it by applying the unit (denoted as $s?(x).x\,()$). In our calculus, communications are always within a *session*, established when accept and receive processes synchronise on a shared channel:

$$a(x).x!\langle 5\rangle.x!\langle\mathtt{true}\rangle.x?(y).(y() \mid R) \mid \overline{a}(x).x?(z_1).x?(z_2).x!\langle\ulcorner P \urcorner\rangle$$

resulting in a fresh session, consisting two channels s and $\overline{s}$, each private to one of the two processes, and their associated queues initialised to be empty:

$$(\nu\, s)(s!\langle 5\rangle.s!\langle\mathtt{true}\rangle.s?(y).(y() \mid R) \mid \overline{s}?(z_1).\overline{s}?(z_2).\overline{s}!\langle\ulcorner P \urcorner\rangle \mid s\!:\!\varepsilon \mid \overline{s}\!:\!\varepsilon)$$

To avoid conflicts, an output on a channel s (resp. $\overline{s}$) places the value on the *dual* queue $\overline{s}$ (resp. s), while an input on s reads from s (resp. for $\overline{s}$). Thus, after two steps the outputs of 5 and `true` are placed on queue $\overline{s}$ as follows:

$$(\nu\, s)(s?(y).(y() \mid R) \mid \overline{s}?(z_1).\overline{s}?(z_2).\overline{s}!\langle\ulcorner P \urcorner\rangle \mid s\!:\!\varepsilon \mid \overline{s}\!:\!5\cdot\mathtt{true})$$

and in two more steps the right process receives and reduces to $\overline{s}!\langle\ulcorner P \urcorner\{5/z_1\}\{\mathtt{true}/z_2\}\rangle$. Similarly the next step transmits the thunked process, and R can interact with P locally. The session type of $\overline{s}$, $S = ?[\mathsf{nat}].?[\mathsf{bool}].![H]$ (where H is the type of $\ulcorner P \urcorner$), guarantees that values are received following the order specified by S.

Asynchronous Communication Optimisation with Code Mobility Suppose the size of P is very large and it does not contain z_1 and z_2. Then the right process might wish to start transmission of P to $s : \varepsilon$ concurrently without waiting for the delivery of 5 and `true`, since the sending is non-blocking. Thus we send $\ulcorner P \urcorner$ ahead as in $\overline{s}!\langle\ulcorner P \urcorner\rangle.\overline{s}?(z_1).\overline{s}?(z_2).\mathbf{0}$. The interaction with the left process is *safe* as the outputs are ordered in an exact complementary way. However the optimised code is not composable with the other party by the original session system [16] since it cannot be assigned S. To make this optimisation valid, we proposed the *asynchronous subtyping* in [12] by which we can refine a protocol to maximise asynchrony without violating the session. For example, in the above case, $S' = ![H].?[\mathsf{nat}].?[\mathsf{bool}]$ is an asynchronous subtype of S, hence the resulting optimisation is typable.

The idea of this subtyping is intuitive and the combination of two kinds of optimisations is vital for typing many practical protocols [17, 18] and parallel algorithms [13], but it requires subtle formal formulations due to the presence of higher-order code. The linear functional typing developed in [11] permits to send a value that contains free session channels: for example, not only message $\overline{s}!\langle\ulcorner s'?(x).s'!\langle x\rangle\urcorner\rangle$ (for $\overline{s}!\langle\ulcorner P \urcorner\rangle$), but also one which contains its own session $\overline{s}!\langle\ulcorner\overline{s}?(x).\overline{s}!\langle x\rangle\urcorner\rangle$ is typable (if R conforms with the dual session like $R = s!\langle 7\rangle.s?(z).\mathbf{0}$). The first message can go ahead correctly, but the permutation of the second message (as $\overline{s}!\langle\ulcorner P \urcorner\rangle$) violates safety since the input action $\overline{s}?(x)$ will appear in parallel with $\overline{s}?(z_1).\overline{s}?(z_2)$, creating a race condition, as seen in:

$$(\nu\, s)(\overline{s}?(x).\overline{s}!\langle x\rangle \mid R \mid \overline{s}?(z_1).\overline{s}?(z_2).\mathbf{0} \mid s\!:\!\varepsilon \mid \overline{s}\!:\!5\cdot\mathtt{true})$$

(Identifiers)	$u, v, w ::= x, y, z$	variables		$k ::= x, y, z$	variables
	$\mid\ a, b, c$	shared channels		$\mid\ s, \bar{s}$	session channels

(Terms)			(Values)	
P, Q, R	$::= V$	value	$V ::= u, v, w$	shared
	$\mid\ u(x).P$	server	$\mid\ k$	linear
	$\mid\ \overline{u}(x).P$	client	$\mid\ ()$	unit
	$\mid\ k?(x).P$	input	$\mid\ \lambda(x{:}U).P$	abstraction
	$\mid\ k!\langle V\rangle.P$	output	$\mid\ \mu(x{:}U \to T).\lambda(y{:}U).P$	recursion
	$\mid\ k \triangleright \{l_1{:}P_1, \ldots, l_n{:}P_n\}$	branching		
	$\mid\ k \triangleleft l.P$	selection	(Message Values)	
	$\mid\ P \mid Q$	parallel	$h ::= l$	label
	$\mid\ (\nu\, a : \langle S\rangle)\, P$	restriction	$\mid\ V$	
	$\mid\ (\nu\, s)\, P$	restriction		
	$\mid\ P\, Q$	application	(Abbreviations)	
	$\mid\ \mathbf{0}$	nil process	$\ulcorner P \urcorner \overset{\text{def}}{=} \lambda(x{:}\mathsf{unit}).P \quad (x \notin \mathsf{fv}(P))$	thunk
	$\mid\ s{:}\vec{h}$	queue	$\mathit{run} \overset{\text{def}}{=} \lambda x.(x\,())$	run

Fig. 1. Syntax

This paper shows that the combination of two optimisations is indeed possible by establishing soundness and communication-safety, subsuming the original typability from [11]. The technical challenge is to prove the transitivity of the asynchronous subtyping integrated with higher-order (linear) function types and session-delegation, since the types now appear in arbitrary contravariant positions [12]. Another challenge is to formulate a runtime typing system which handles both stored higher-order code with open sessions and the asynchronous subtyping. We demonstrate all facilities of type-preserving optimisations proposed in this paper by using an e-commerce scenario. A full version, containing omitted definitions and proofs, is available from [1].

2 The Higher-Order π-Calculus with Asynchronous Sessions

2.1 Syntax and Reduction

The calculus is given in Fig. 1, based on the π-calculus augmented with asynchronous session primitives and the call-by-value λ-calculus. Except for recursion and message queues for asynchronous communications [8], all constructs are from the synchronous Higher-Order calculus with sessions [11]. A session is initiated over a *shared channel* and communications belonging to a session are performed via two fresh end-point channels specific to that session, called *session channels* or *queue endpoint channels*, used to distinguish the two end points, taking a similar approach to [5, 20]. The *dual* of a queue endpoint s is denoted by $\bar{s}$, and represents the other endpoint of the same session. The operation is self-inverse hence $\bar{\bar{s}} = s$. We write $\vec{V}$ for a potentially empty vector $V_1 \ldots V_n$. Types, given later, are denoted by U, T and $\langle S\rangle$, but type annotations are often omitted.

For values, we have shared and linear identifiers, unit, abstraction and recursion. For terms, we have prefixes for declaring session connections, $u(x).P$ for servers and

$$
\begin{array}{llll}
\text{(beta)} & (\lambda x.P)V \longrightarrow P\{^{V}\!/_{x}\} & \text{(rec)} & (\mu y.\lambda x.P)V \longrightarrow P\{^{V}\!/_{x}\}\{^{\mu y.\lambda x.P}\!/_{y}\} \\
\text{(send)} & s!\langle V\rangle.P \mid \overline{s}:\vec{h} \longrightarrow P \mid \overline{s}:\vec{h}\cdot V & \text{(get)} & s?(x).P \mid s:V\cdot\vec{h} \longrightarrow P\{^{V}\!/_{x}\} \mid s:\vec{h} \\
\text{(sel)} & s \lhd l.P \mid \overline{s}:\vec{h} \longrightarrow P \mid \overline{s}:\vec{h}\cdot l & \text{(bra)} & s \rhd \{l_1:P_1,\ldots,l_n:P_n\} \mid s:l_m\cdot\vec{h} \longrightarrow P_m \mid s:\vec{h} \\
 & & & \qquad\qquad (1 \leq m \leq n)
\end{array}
$$

$$
\text{(conn)}\quad a(x).P \mid \overline{a}(z).Q \longrightarrow (\nu\, s)\,(P\{^{s}\!/_{x}\} \mid Q\{^{\overline{s}}\!/_{z}\} \mid s:\varepsilon \mid \overline{s}:\varepsilon) \quad s,\overline{s}\text{ fresh}
$$

$$
\text{(app-l)}\ \frac{P \longrightarrow P'}{PQ \longrightarrow P'Q} \qquad \text{(app-r)}\ \frac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'} \qquad \text{(par)}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}
$$

$$
\text{(resc)}\ \frac{P \longrightarrow P'}{(\nu\, a)P \longrightarrow (\nu\, a)P'} \qquad \text{(ress)}\ \frac{P \longrightarrow P'}{(\nu\, s)P \longrightarrow (\nu\, s)P'} \qquad \text{(str)}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}
$$

Fig. 2. Reduction

$\overline{u}(x).P$ for clients. Session communications are performed using the next four primitives: input $k?(x).P$, output $k!\langle V\rangle.P$, branching $k \rhd \{l_1:P_1,\ldots,l_n:P_n\}$ (often written as $k \rhd \{l_i : P_i\}_{i\in I}$ with index set I) which offers alternative interaction patterns, and selection $k \lhd l.P$ which chooses an available branch. $(\nu\, a : \langle S\rangle)P$ restricts (and binds) a channel a to the scope of P. Similarly, $(\nu\, s)\,P$ binds s and $\overline{s}$, making them private to P. $s:\vec{h}$ is a *message queue*, also called *buffer*, representing ordered messages in transit with destination s (which may be considered as a network pipe in a TCP-like transport). Queues and session restrictions appear only at runtime. A *program* is a process which does not contain runtime terms. Other primitives are standard. We often omit $\mathbf{0}$.

The *bindings* are induced by $(\nu\, a:\langle S\rangle)P$, $(\nu\, s)P$, $u(x).P$, $\overline{u}(x).P$, $\lambda x.P$ and $\mu y.\lambda x.P$. The derived notions of bound and free identifiers, alpha equivalence and substitution are standard. We write $\mathsf{fv}(P)/\mathsf{fn}(P)$ for the set of free variables/channels, respectively.

By using recursion, we can represent infinite behaviours of processes such as, e.g. the definition agent def or $!u(y).P$ in [7, 10, 11, 20]. For example the replication $!u(y).P$ in [11] can be defined as $u(x).(\mu y.\lambda z.(P \mid z(x).yz))u$ with $x \notin \mathsf{fv}(P)$.

The single-step call-by-value reduction relation, denoted $\longrightarrow$, is a binary relation from closed terms to closed terms, defined by the rules in Fig. 2. The rules are from those of the HOπ-calculus [11] combined with asynchronous session communications from [8]. Rule (conn) establishes a new session between server and client via shared name u, generating two fresh session channels and the associated two empty queues (ε denotes the empty string). Rules (send) and (sel) respectively enqueue a value and a label at the tail of the queue for a dual endpoint $\overline{s}$. Rules (get) and (bra) dequeue, from the head of the queue, a value or label. (get) substitutes value V for x in P, while (bra) selects the corresponding m-branch. Since (conn) provides a queue for each channel, these rules say that a sending action is never blocked (asynchrony) and that two messages from the same sender to the same channel arrive in the sending order (order preservation). Other rules are standard. A session channel s and $\overline{s}$ can be sent and received (when $V = k$), with which various protocols are expressed, allowing complex nested and private structured communications. This interaction is called *higher-order session passing* (delegation). We use the standard structure rules [10] $\equiv$ such as $(\nu s)\,P \,|\, Q \equiv (\nu s)\,(P \,|\, Q)$ if $s, \bar{s} \notin \mathsf{fn}(Q)$ (see [11]). "$\twoheadrightarrow$" denote the multi-step reductions defined as $(\equiv \cup \rightarrow)^*$.

2.2 Example: Optimised Business Protocol with Code Mobility

We show a business/financial protocol interaction from [17, 18] which integrates the two kinds of type-safe optimisations. We extend the scenario from [11] to highlight the expressiveness gained using the new method. Fig. 3 draws the sequencing of actions modelling a hotel booking through a process `Agent`. On the left `Client` behaves dually to `Agent`; on the right, an optimised `MClient` utilises type-safe asynchronous behaviour.

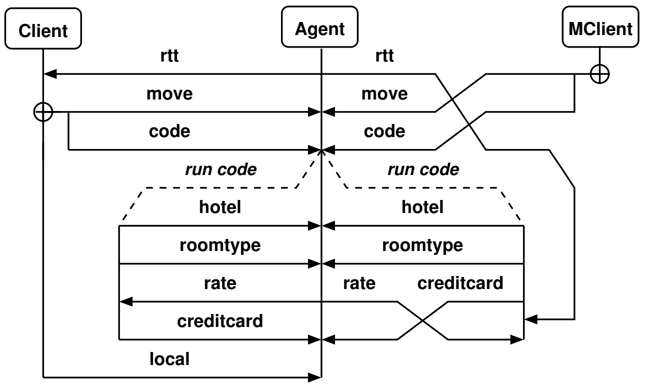

Fig. 3. Standard (left) and Optimised (right) Interaction for Hotel Booking

The `Agent` behaves the same towards both clients: initially it calculates the round-trip time (RTT) of communication (`rtt`) and sends it; it then offers to the other party the option to consider the RTT and either send *mobile code* to interact with the `Agent` on its location, or to continue the protocol with each executing remotely their behaviour. When mobile code (after choice move) is received, it is *run* by the `Agent` completing the transaction on behalf of the client, in a sequence of steps. The behaviour of `Client` is straightforward and complementary to `Agent`, but `MClient` has special requirements: it represents a mobile device with limited processing power, and irrespective of the RTT it always sends mobile code; moreover, it does not care about money, and provides the creditcard number (`card`) before finding out the `rate`.

To represent this optimised scenario, we start from the process for `Agent`:

$$\begin{aligned}\texttt{Agent} &= a(x).x!\langle\texttt{rtt}\rangle.x \rhd \{\mathsf{move} : x?(code).(\mathsf{run}\; code \mid Q), \mathsf{local} : Q\}\\ Q &= x?(hotel).x?(roomtype).x!\langle\texttt{rate}\rangle.x?(creditcard)\ldots\end{aligned}$$

The session is initiated over a, then the `rtt` is sent, then the choices `move` and `local` are offered. If the first choice is made then the received code is run in parallel to the process

Q which continues the agent's session, performing optimisation by code mobility. As expected, `Client` has dual behaviour:

$$\texttt{Client} = \overline{a}(x).x?(rtt).x \lhd \mathsf{move}.x!\langle\ulcorner x!\langle\texttt{ritz}\rangle.x!\langle\texttt{suite}\rangle.x?(rate).x!\langle\texttt{card}\rangle\ldots\urcorner\rangle$$

A more interesting optimisation is given by `MClient` which at first may seem to disagree with the intended protocol:

$$\texttt{MClient} = \overline{a}(x).x \lhd \mathsf{move}.x!\langle\ulcorner x!\langle\texttt{ritz}\rangle.x!\langle\texttt{suite}\rangle.x!\langle\texttt{card}\rangle.x?(rtt).x?(rate)\ldots\urcorner\rangle$$

After the session is established, it eagerly sends its choice move, ignoring `rtt`, followed by a thunk that will continue the session; and another important point is that in the mobile code the output of the `card` happens before `rtt` and `rate` are received.

Even without subtyping, the typing of sessions in the HOπ-calculus poses delicate conditions [11]; in the present system, we can further verify that the optimisation of `MClient` does not violate communications safety (but the similar example in § 1, $\overline{s}!\langle\ulcorner\overline{s}?(x).\overline{s}!\langle x\rangle\urcorner\rangle.\overline{s}?(z_1).\overline{s}?(z_2).\mathbf{0}$, must be untypable): when values are received they are always of the expected type, conforming to a new *subtyping* relation given in the next section.

3 Higher-Order Linear Types with Asynchronous Subtyping

3.1 Types

This section presents an asynchronous subtyping relation for the HOπ-calculus based on [12]. The syntax of the types is given below:

$$\begin{array}{llll}
\text{Term} & T & ::= & U \mid \diamond \\
\text{Value} & U & ::= & H \mid S \qquad \text{HO-value} \quad H \ ::= \ \mathsf{unit} \mid U \to T \mid U \multimap T \mid \langle S\rangle \\
\text{Session} & S & ::= & ![U].S \mid ?[U].S \mid \oplus[l_1 : S_1, \ldots, l_n : S_n] \mid \&[l_1 : S_1, \ldots, l_n : S_n] \\
 & & \mid & \mu\mathbf{t}.S \mid \mathbf{t} \mid \mathsf{end}
\end{array}$$

It is an integration of the types from the simply typed λ-calculus with linear functional types, $U \multimap T$, and the session types from the π-calculus. A linear type represents a function to be used exactly once. *Term types*, ranging over T, include all value types and the process type $\diamond$. *Session types* range over $S, S', \ldots$ *Higher-Order value types* consist of the unit type, the function types, the linear function types and the channel type $\langle S\rangle$, and *value types* consist of HO-value and session types. Note that linear types are attached only to function types. In the session types, $![U].S$ represents the output of a value typed by U followed by a session typed by S; $?[U]$ is its dual. $\oplus[l_1 : S_1, \ldots, l_n : S_n]$ is the selection type on which one of the labels l_i can be sent, with the subsequent session typed by S_i; $\&[l_1 : S_1, \ldots, l_n : S_n]$ is its dual called the branching type. $\mathbf{t}$ is a type variable and $\mu\mathbf{t}.S$ is a recursive type. We only consider contractive recursive types [20]. end denotes the termination of the session. We often write $\&[l_i : S_i]_{i\in I}$ and $\oplus[l_i : S_i]_{i\in I}$ and $\ulcorner T\urcorner$ for $\texttt{unit} \to T$ and $\ulcorner T\urcorner^1$ for or $\texttt{unit} \multimap T$. The type end is often omitted.

Each session type S has a *dual* type, denoted by $\overline{S}$, which describes complementary behaviour. This is inductively defined as: $\overline{![U].S} = ?[U].\overline{S}$, $\overline{\oplus[l_1 : S_1, \ldots, l_n : S_n]} =$

$$
\begin{array}{llll}
(\mathsf{OI}) & ![U].?[U'].S \ll ?[U'].![U].S & (\mathsf{SI}) & \oplus[l_j : ?[U].S_j]_{j\in J} \ll ?[U].\oplus[l_j : S_j]_{j\in J} \\
(\mathsf{OB}) & ![U].\&[l_j : S_j]_{j\in J} \ll \&[l_j : ![U].S_j]_{j\in J} & (\mathsf{SB}) & \oplus[l_i : \&[l'_j : S_{ij}]_{j\in J}]_{i\in I} \ll \&[l'_j : \oplus[l_i : S_{ij}]_{i\in I}]_{j\in J}
\end{array}
$$

$$
(\mathsf{Tr})\ \frac{S_1 \ll S_2 \quad S_2 \ll S_3}{S_1 \ll S_3} \qquad (\mathsf{CB})\ \frac{\forall i \in I.\ S_i \ll S'_i}{\&[l_i : S_i]_{i\in I} \ll \&[l_i : S'_i]_{i\in I}} \qquad (\mathsf{CI})\ \frac{S \ll S'}{?[U].S \ll ?[U].S'}
$$

$$
(\mathsf{CO})\ ![U].S \ll ![U].S \quad (\mathsf{CS})\ \oplus[l_i : S_i]_{i\in I} \ll \oplus[l_i : S_i]_{i\in I} \quad (\mathsf{E})\ \mathsf{end} \ll \mathsf{end} \quad (\mathsf{M})\ \mu\mathbf{t}.S \ll \mu\mathbf{t}.S
$$

Fig. 4. Top Level Asynchronous Action Rules

$\overline{\&[l_1 : S_1, ..., l_n : S_n]} = \oplus[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}]$, $\overline{?[U].S} = ![U].\overline{S}$, $\overline{\&[l_1 : S_1, \dots, l_n : S_n]} = \oplus[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}]$, $\overline{\mathbf{t}} = \mathbf{t}$, $\overline{\mu\mathbf{t}.S} = \mu\mathbf{t}.\overline{S}$ and $\overline{\mathsf{end}} = \mathsf{end}$.

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, $![U_1]$ and $?[U_2]$ in $![U_1].?[U_2].\mu\mathbf{t}.![U_3].\mathbf{t}$ are top-level, but $![U_3]$ in the same type is not). In the above type, $![U_1]$ is *the head* since it appears as the left-most occurrence of the top-level actions in S (note that $?[U_2]$ is not the head). We write *Type* for the collection of all closed types.

3.2 Higher-Order Asynchronous Subtyping

This subsection studies a theory of asynchronous session subtyping: reordered communications, even higher-order and mobile, can preserve the faithfulness to the other dual party. Fig. 4 defines the axioms for partial permutation of top-level actions for closed types, denoted $\ll$. $S \ll S'$ is read: S is an *action-asynchronous subtype* of S', and means S is more asynchronous than (or more optimised than) S'. We write $S \gg S'$ for $S' \ll S$. A permutation of two inputs or two outputs is not allowed since it violates type-safety. Suppose $P = s!\langle 2\rangle.s!\langle \mathtt{true}\rangle.s?(x).\mathbf{0}$ and $Q = \overline{s}?(y).\overline{s}?(z).\overline{s}!\langle y+2\rangle.\mathbf{0}$. These processes interact correctly. If we permute the outputs of P to get $P' = s!\langle \mathtt{true}\rangle.s!\langle 2\rangle.s?(x).\mathbf{0}$, then the parallel composition $(P' \mid Q)$ causes a type-error. Similarly the reverse direction of $(\mathsf{OI}, \mathsf{OB}, \mathsf{SI}, \mathsf{SB})$ causes a deadlock, losing progress in session s. For example, consider exchanging $s!\langle \mathtt{true}\rangle$ and $s?(z)$ in $P_1 = s!\langle \mathtt{true}\rangle.s?(z).\mathbf{0}$, and $Q_1 = \overline{s}?(y).\overline{s}!\langle 2\rangle.\mathbf{0}$. Note that partial permutation is only applied to finite parts of the top-level actions *without* unfolding recursive types.

To handle recursive types in asynchronous subtyping, we need to generalise the unfolding function defined in [5] since $\ll$ might be applicable to a type after unfolding of recursions under some guarded prefixes. The definition is based on [12].

Definition 3.1 (*n*-**time unfolding).**

$$
\begin{array}{ll}
\mathsf{unfold}^0(S) = S \text{ for all } S & \mathsf{unfold}^{1+n}(S) = \mathsf{unfold}^1(\mathsf{unfold}^n(S)) \\
\mathsf{unfold}^1(![U].S) = ![U].\mathsf{unfold}^1(S) & \mathsf{unfold}^1(\oplus[l_i : S_i]_{i\in I}) = \oplus[l_i : \mathsf{unfold}^1(S_i)]_{i\in I} \\
\mathsf{unfold}^1(?[U].S) = ?[U].\mathsf{unfold}^1(S) & \mathsf{unfold}^1(\&[l_i : S_i]_{i\in I}) = \&[l_i : \mathsf{unfold}^1(S_i)]_{i\in I}
\end{array}
$$

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

For any recursive type S, $\mathsf{unfold}^n(S)$ is the result of inductively unfolding the top level recursion up to a fixed level of nesting. Because our recursive types are contractive, $\mathsf{unfold}^n(S)$ terminates.

We now introduce the main definition of the paper, asynchronous communication subtyping for the HOπ-calculus. First, let us define:

$$(H,H')^{\circledast} = (H,H') \qquad (S,S')^{\circledast} = (S',S) \qquad (\diamond,\diamond)^{\circledast} = (\diamond,\diamond)$$

which is used to adjust for the different variance of functional and session types.

Definition 3.2 (Asynchronous Subtyping). 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:

1. If $T_1 = \diamond$, then $T_2 = \diamond$.
2. If $T_1 = \mathsf{unit}$, then $T_2 = \mathsf{unit}$.
3. If $T_1 = U_1 \to T_1'$, then $T_2 = U_2 \to T_2'$ or $T_2 = U_2 \multimap T_2'$ with $(U_2,U_1)^{\circledast} \in \mathfrak{R}$ and $(T_1',T_2')^{\circledast} \in \mathfrak{R}$.
4. If $T_1 = U_1 \multimap T_1'$, then $T_2 = U_2 \multimap T_2'$ with $(U_2,U_1)^{\circledast} \in \mathfrak{R}$ and $(T_1',T_2')^{\circledast} \in \mathfrak{R}$.
5. If $T_1 = \langle S_1 \rangle$, then $T_2 = \langle S_2 \rangle$ and $(S_1,S_2) \in \mathfrak{R}$ and $(S_2,S_1) \in \mathfrak{R}$.
6. If $T_1 = \mathsf{end}$, then $\mathsf{unfold}^n(T_2) = \mathsf{end}$.
7. If $T_1 = ![U_1].S_1$, then $\mathsf{unfold}^n(T_2) \gg ![U_2].S_2$, $(U_1,U_2)^{\circledast} \in \mathfrak{R}$ and $(S_1,S_2) \in \mathfrak{R}$.
8. If $T_1 = ?[U_1].S_1$, then $\mathsf{unfold}^n(T_2) = ?[U_2].S_2$, $(U_2,U_1)^{\circledast} \in \mathfrak{R}$ and $(S_1,S_2) \in \mathfrak{R}$.
9. If $T_1 = \oplus[l_i : S_{1i}]_{i \in I}$, then $\mathsf{unfold}^n(T_2) \gg \oplus[l_j : S_{2j}]_{j \in J}$, $I \subseteq J$ and $\forall i \in I.(S_{1i},S_{2i}) \in \mathfrak{R}$.
10. If $T_1 = \&[l_i : S_{1i}]_{i \in I}$, then $\mathsf{unfold}^n(T_2) = \&[l_j : S_{2j}]_{j \in J}$, $J \subseteq I$ and $\forall j \in J.(S_{1j},S_{2j}) \in \mathfrak{R}$.
11. If $T_1 = \mu\mathbf{t}.S$, then $(\mathsf{unfold}^1(T_1),T_2) \in \mathfrak{R}$.

As standard, 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 $(T_1,T_2) \in \mathfrak{R}$.

The integration of the subtyping of higher-order (linear) functions and asynchronous sessions requires a careful formulation: (1,2,6) are standard identity rules. (3) says the unlimited function can be used as the linear function. Note that the reverse is unsafe: suppose $f = \lambda x.k!\langle x \rangle$ with a linear type $\mathsf{nat} \multimap \diamond$. If we apply the reverse direction, $\lambda(y{:}\mathsf{nat} \to \diamond).(y\,1 \mid y\,2)f$ becomes typable, destroying the linearity of session k.

In (3), when U_i is a session type, we use the relation $(S_1,S_2)^{\circledast} = (S_2,S_1)$ to swap the tuple. The session types are dualised since the session channel is going to be used in a process in a contravariant manner.[1] To see this condition, suppose process $P = (\lambda(x:S).x!\langle 2 \rangle.x?(y).\mathbf{0})\,s$ with $S = ![\mathsf{nat}].?[\mathsf{bool}].\mathsf{end}$. Then P can safely interact with $Q = \bar{s}!\langle \mathtt{true} \rangle.\bar{s}?(z).\mathbf{0}$. For P to be composable with Q, s in P has a dual type of $\bar{s}$ in Q, which is $S' = ?[\mathsf{bool}].![\mathsf{nat}].\mathsf{end}$. Hence we must have $S \to \diamond \leqslant_c S' \to \diamond$, with $S \leqslant_c S'$ where the subtyping ordering of session channels is covariant. The case T_i is a session type is similarly explained. (4) is similar.

[1] The original session typing system uses a judgement "$\Gamma \vdash P : \Sigma$" where Γ is a shared (standard) environment and Σ is a mapping from a session channel to a session type. This means: P accesses the session channels specified at most by Σ. Contrarily, in our typing system defined in the next section, Σ appears in the left-side position, so that we need to dualise the session types for subtyping, cf. [19].

(5) says the shared channel type is invariant (as is the standard session types [5, 7, 12]). In (7), 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 similar reasons with (3), we swap the ordering if they are session types. For the input in (8), we do not require $\gg$, since, by definition of $\ll$, if the input appears at the top level in S, then it does so in all S' such that $S \ll S'$. The definitions of selection and branching subsume the traditional session branching/selection subtyping. In (9), selection is defined similarly to output since a label appearing in T_1 must be included in T_2; dually, in (10), branching is defined like input and any branch of T_2 must be included in T_1. Finally (11) forces T_1 to be unfolded until it reaches a guarded type.

More examples can be found in § 4.3. We conclude this section with the main theorem for $\leqslant_c$. Since now types include higher-order function types and session delegations with a combination of n-time unfolding and permutation, the proof of transitivity of $\leqslant_c$ requires a family of relations to *connect* two relations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, for which we use the transitivity connection $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$.

Lemma 3.3. *If $S_1 \leqslant_c S_2$ and $S'_1 \ll \mathsf{unfold}^n(S_1)$ then $S'_1 \leqslant_c S_2$.*

From the above lemma we have that whenever $S_1 \,\mathfrak{R}\, S_2$ for type simulation $\mathfrak{R}$, then for some n-times unfolding of S_1, and after applying a sequence of permutations to obtain S'_1 such that $S'_1 \ll \mathsf{unfold}^n(S_1)$, there exists a type simulation $\mathfrak{R}'$ such that $S'_1 \,\mathfrak{R}'\, S_2$.

We use this fact below: given $S_1 \,\mathfrak{R}\, S_2$, we obtain a simulation (the union of $\mathfrak{R}'$) for each level n of unfolding of S_1, relating each possible permutation S'_1 of $\mathsf{unfold}^n(S_1)$ and S_2.

Definition 3.4 (Transitivity Connection). When $S_1 \,\mathfrak{R}\, S_2$ for type simulation $\mathfrak{R}$, we define the asynchrony relation of S_1 and S_2 as:

$$\mathcal{A}(S_1, S_2) = \bigcup_{n \in \mathbb{N}} \left\{ (S'_1, S'_2) \mid \mathsf{unfold}^n(S_1) \gg S \;\wedge\; S \,\mathfrak{R}'\, S_2 \;\wedge\; \mathfrak{R}' \subseteq \leqslant_c \;\wedge\; (S'_1, S'_2) \in \mathfrak{R}' \right\}$$

Then, for type simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, we define their transitivity connection $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ as the smallest relation such that whenever $S_1 \,\mathfrak{R}_1\, S_2$ and $S_2 \,\mathfrak{R}_2\, S_3$, we have $\mathcal{A}(S_2, S_3) \subseteq \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$.

For type simulation $\mathfrak{R}$ with $S_1 \,\mathfrak{R}\, S_2$, $\mathcal{A}(S_1, S_2)$ (hence $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$) is also a type simulation. Using this property, we have:

Theorem 3.5 (Preorder). *$\leqslant_c$ is a preorder.*

Proof. Reflexivity is easy. For transitivity, we assume $(T_1, T_2) \in \mathfrak{R}_1$ and $(T_2, T_3) \in \mathfrak{R}_2$ for simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, to find a simulation $\mathfrak{R}$ such that $(T_1, T_3) \in \mathfrak{R}$. Define $\mathfrak{R}$ as:

$$\mathfrak{R} = \mathfrak{R}_{12} \cdot \mathfrak{R}_{21} \cup \mathfrak{R}_{21} \cdot \mathfrak{R}_{12} \qquad \text{with} \qquad \mathfrak{R}_{ij} = \mathfrak{R}_i \cup \mathbf{trc}(\mathfrak{R}_j, \mathfrak{R}_i)$$

We have $(T_1, T_3) \in \mathfrak{R}_1 \cdot \mathfrak{R}_2 \subseteq \mathfrak{R}_{12} \cdot \mathfrak{R}_{21} \subseteq \mathfrak{R}$; we then prove that $\mathfrak{R}$ is a type simulation, observing the fact that each $\mathfrak{R}_{ij}$ above is a type simulation, as the union of type simulations, see [1].

(Common)

$$\text{(Shared)}\ \frac{H \neq U \multimap T}{\Gamma, u{:}H; \emptyset; \emptyset \vdash u : H} \qquad \text{(Session)}\ \frac{}{\Gamma; k{:}S; \emptyset \vdash k : S} \qquad \text{(LVar)}\ \frac{}{\Gamma, x : U \multimap T; \emptyset; \{x\} \vdash x : U \multimap T} \qquad \text{(Base)}\ \frac{}{\Gamma; \emptyset; \emptyset \vdash () : \mathtt{unit}}$$

(Function) **(Process)**

$$\text{(Abs)}\ \frac{\Gamma, x{:}H; \Sigma; \mathcal{L} \vdash P : T \quad \text{if } H = U \multimap T' \text{ then } x \in \mathcal{L}}{\Gamma; \Sigma; \mathcal{L} \setminus x \vdash \lambda(x{:}H).P : H \to T} \qquad \text{(Abs}_S\text{)}\ \frac{\Gamma; \Sigma, x{:}S; \mathcal{L} \vdash P : T}{\Gamma; \Sigma; \mathcal{L} \vdash \lambda(x{:}S).P : S \to T} \qquad \text{(Sub)}\ \frac{\Gamma; \Sigma; \mathcal{L} \vdash P : H \quad \Sigma \leqslant_c \Sigma' \quad H \leqslant_c H'}{\Gamma; \Sigma'; \mathcal{L} \vdash P : H'}$$

$$\text{(Recursion)}\ \frac{\Gamma, x{:}U \to T; \emptyset; \emptyset \vdash \lambda(y{:}U).P : U \to T}{\Gamma; \emptyset; \emptyset \vdash \mu(x{:}U \to T).\lambda(y{:}U).P : U \to T} \qquad \text{(App)}\ \frac{\Gamma; \Sigma_1; \mathcal{L}_1 \vdash P : U \multimap T \quad \Gamma; \Sigma_2; \mathcal{L}_2 \vdash Q : U \quad \text{if } U = U' \to T' \text{ then } \Sigma_2 = \mathcal{L}_2 = \emptyset}{\Gamma; \Sigma_1, \Sigma_2; \mathcal{L}_1, \mathcal{L}_2 \vdash PQ : T}$$

Fig. 5. Selected Linear Session Typing

4 Asynchronous Higher-Order Session Typing

4.1 A Typing System for Programs

We first introduce the linear Higher-Order typing system for programs (terms which do not contain queues and session-restrictions). We define two environments:

$$\Gamma ::= \emptyset \mid \Gamma, u : H \qquad\qquad \Sigma ::= \emptyset \mid \Sigma, k : S$$

Γ is a finite mapping, associating HO value types to identifiers. Σ is a finite mapping from session channels to session types. In addition, we use a finite set of linear variables ranged over $\mathcal{L}$, $\mathcal{L}'$, ... to ensure linear usage of function terms that may contain session channels. Σ, Σ' and $\mathcal{L}, \mathcal{L}'$ denote disjoint-domain unions. $\Gamma, u{:}U$ means $u \notin \mathrm{dom}(\Gamma)$.

Then the typing judgement takes the shape:

$$\Gamma; \Sigma; \mathcal{L} \vdash P : T$$

which is read: under a global environment Γ, a term P has a type T with session usages described by Σ and linear variables specified by $\mathcal{L}$. We say the judgement is *well-formed* if $\mathrm{dom}(\Gamma) \supseteq \mathcal{L}$ and $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma) = \emptyset$. The typing system is given in Fig. 5. In each rule, we assume the environments of the consequence are defined.

We focus on the rules which differ from [11]. In the first group, **(Common)**, (Shared) is an introduction rule for identifiers with shared types; (Session) is for session channels. (LVar) is for linear variables. The second group, **(Function)**, comes from the typed linear λ-calculus with recursive types. All rules are identical except the addition of (Recursion). In (Abs), the premise side-condition ensures that the formal parameter x, to be substituted with the received function, appears in the linear variables. In the conclusion, we remove x from the function environment. (Abs$_S$) is an abstraction rule for session channels. (Recursion) forbids the use of any *free* linear identifier (by the condition $\Sigma = \mathcal{L} = \emptyset$) because it is repeatedly used. (App) is the rule for application; the side condition ensures that when the right term is of shared function type, it is required not to have free session channels or linear variables. The conclusion says that P and Q's session environments and linear variable sets are disjoint. The final group, **(Process)**, is for processes. The only different rule from [11] is (Sub) which now uses $\leqslant_c$; we write $\Sigma \leqslant_c \Sigma'$ when $\mathrm{dom}(\Sigma) = \mathrm{dom}(\Sigma')$ and for all $k\!:\!S \in \Sigma$, we have $k\!:\!S' \in \Sigma'$ with $S \leqslant_c S'$. The rest of the rules and their explanations can be found in [11].

4.2 A Typing System for Runtime

Session Remainder Type soundness is established by also typing the queues created during the execution of a well-typed initial program. We track the movement of linear functions and channels to and from the queue to ensure that linearity is preserved, and we check that endpoints continue to have dual types up to asynchronous subtyping after each use. To analyse the intermediate steps precisely, we utilise a *session remainder* $S - \vec{\tau} = S'$ which subtracts the vector $\vec{\tau}$ of queue types ($\tau ::= U \mid l$) of the values stored in a queue from the complete session type S of the queue, obtaining a remaining session S'. The rules are formalised below:

$$\begin{array}{llllll}
\textsf{(Empty)} & S - \varepsilon = S & \textsf{(Get)} & S - \vec{\tau} = S' & \Rightarrow & ?[U].S - U\vec{\tau} = S' \\
 & & \textsf{(Put)} & S - \vec{\tau} = S' & \Rightarrow & ![U].S - \vec{\tau} = ![U].S' \\
 & & \textsf{(Branch)} & S_k - \vec{\tau} = S' \wedge k \in I & \Rightarrow & \&[l_i : S_i]_{i \in I} - l_k\vec{\tau} = S' \\
 & & \textsf{(Select)} & \forall i \in I \,.\, S_i - \vec{\tau} = S'_i & \Rightarrow & \oplus[l_i : S_i]_{i \in I} - \vec{\tau} = \oplus[l_i : S'_i]_{i \in I}
\end{array}$$

When S' is end, then the session has been completed; otherwise it is not closed yet.

(Empty) is a base rule. (Get) takes an input prefixed session type $?[U].S$ and subtracts the type U at the head of the queue, then returns the remainder S' of the rest of the session S minus the tail $\vec{\tau}$ of the queue type. (Put) disregards the output action type of the session and calculates the remainder S' of $S - \vec{\tau}$, which is returned prefixed with the original output giving $![U].\vec{\tau}$. Therefore the output is not consumed. (Branch) is similar with (Get), but it only records the remainder of the k-th branch with respect to a stored label l_k. Dually, (Select) records the remainder of all selection paths.

A Typing System for Runtime We first extend the session environment as follows:

$$\Delta ::= \Sigma \mid \Delta, s : \vec{\tau} \mid \Delta, s : (S, \vec{\tau})$$

The typing judgement is also extended with $\Gamma;\Sigma;\mathcal{L} \vdash l : l$ which is used for typing any labels appearing in a session queue. Δ contains usage information for queues ($s : \vec{\tau}$) in a term, so that the cumulative result can be compared with the expected session type; for this we use the pairing ($s : (S, \vec{\tau})$) that combines the usage of a channel and the sequence of types already on its queue. We identify $(S, \vec{\tau})$ and $(\vec{\tau}, S)$.

We define a composition operation $\odot$ on Δ-environments, used to obtain the paired usages for channels and queues:

$$\Delta_1 \odot \Delta_2 = \{s\!:\!(\Delta_1(s), \Delta_2(s)) \mid s \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \cup \Delta_1 \backslash \mathsf{dom}(\Delta_2) \cup \Delta_2 \backslash \mathsf{dom}(\Delta_1)$$

The typing rules for runtime are listed in Fig. 6. (Label) types a label in a queue, while (Queue) forms a sequence of the types of the values in a queue: we ensure the disjointness of session environments of values, and apply a weakening for end (Σ_0) for closure

$$\text{(Label)}\ \frac{}{\Gamma;\emptyset;\emptyset \vdash l : l} \qquad \text{(Queue)}\ \text{if } \tau_i = U \to T \text{ then } \Sigma_i = \emptyset\ \frac{\Gamma;\Sigma_i;\emptyset \vdash h_i : \tau_i \quad i \in 1..n \quad \Sigma_0 = \{\vec{s} : \vec{\mathsf{end}}\}}{\Gamma;(\Sigma_0,..,\Sigma_n) \odot s : \tau_1..\tau_n;\emptyset \vdash s : h_1..h_n : \diamond} \qquad \text{(Par)}\ \frac{\Gamma;\Delta_{1,2};\mathcal{L}_{1,2} \vdash P_{1,2} : \diamond}{\Gamma;\Delta_1 \odot \Delta_2;\mathcal{L}_1,\mathcal{L}_2 \vdash P_1 \mid P_2 : \diamond}$$

$$\text{(New}_s\text{)}\ \frac{\Gamma;\Delta,s:(S_1,\vec{\tau}_1),\bar{s}:(S_2,\vec{\tau}_2);\emptyset \vdash P : \diamond \quad S_i - \vec{\tau}_i = S'_i \quad i \in 1,2 \quad S'_1 \leqslant_c \overline{S'_2}}{\Gamma;\Delta;\emptyset \vdash (\nu s)P : \diamond} \qquad \text{(New)}\ \frac{\Gamma, a:\langle S\rangle;\Delta;\mathcal{L} \vdash P : \diamond}{\Gamma;\Delta;\mathcal{L} \vdash (\nu a:\langle S\rangle)P : \diamond}$$

Fig. 6. Runtime Typing

under the structure rules. (Par) composes processes, including queues, and records the session usage by $\odot$; this rule subsumes (Par) for programs. (New$_s$) is the main rule for typing the two endpoint queues of a session. Types S_1 and S_2 can be given to the queues s and $\bar{s}$ when the session remainders S'_1 and S'_2 of $S_1 - \vec{\tau}_1$ and $S_2 - \vec{\tau}_2$ are dual session types *up to asynchronous subtyping*; more precisely, S'_1 must be a subtype of the dual of S'_2, written $S'_1 \leqslant_c \overline{S'_2}$. Since the session is compatible, we can restrict s. Note that in all runtime systems, the set of linear variables is empty.

4.3 Typing the Optimised Mobile Business Protocol

Using the program and runtime typing systems, we can now type the hotel booking example in § 2.2, in the presence of asynchronous optimisation for higher-order mobility. `Agent` and standard `Client` can be typed with:

$$S_{\texttt{Agent}} = ![\texttt{int}].\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\texttt{Agent}}\ ,\ \mathsf{local} : S'_{\texttt{Agent}}]$$
$$\text{with} \quad S'_{\texttt{Agent}} = ?[\texttt{string}].?[\texttt{string}].![\texttt{double}].?[\texttt{int}].\mathsf{end} \quad \text{and} \quad S_{\texttt{client}} = \overline{S_{\texttt{Agent}}}$$

We then type `MClient` by using the rules in Fig. 5 and [11].

$$S_{\texttt{MClient}} = \oplus[\mathsf{move} : ![\mathsf{unit} \multimap \diamond].![\texttt{string}].![\texttt{string}].![\texttt{int}].?[\texttt{int}].?[\texttt{double}].\mathsf{end}]$$

Applying Def. 3.2 we verify that $S_{\texttt{MClient}} \leqslant_c \overline{S_{\texttt{Agent}}}$ (and $S_{\texttt{MClient}} \leqslant_c S_{\texttt{Client}}$). Then using typing rules (Acc,Req) we can type both `MClient` and `Agent` with $a : \langle \overline{S_{\texttt{Agent}}} \rangle \in \Gamma$, after applying (Sub) on the premises of (Req) typing the body of `MClient`.

We now demonstrate runtime typing; after three reduction steps of `MClient` | `Agent` we can have this configuration:

$$(\nu s)(\bar{s} \triangleright \{\mathsf{move} : \bar{s}?(code).(run\ code \mid \ldots),\ \mathsf{local} : \ldots\} \mid s{:}\texttt{rtt} \mid \bar{s}{:}\mathsf{move}\cdot\ulcorner s!\langle\texttt{ritz}\rangle\ldots\urcorner)$$

with $\bar{s}$ as the `Agent`'s queue. Both queues contain values including the linear higher-order code sent by `MClient` (which became **0** after this output). Using (Queue, Label) we type $\bar{s}{:}\mathsf{move}\cdot\ulcorner s!\langle\texttt{ritz}\rangle\ldots\urcorner$ with session environment $\{s : S'_{\texttt{MClient}}, \bar{s} : \mathsf{move}\cdot\mathsf{unit} \multimap \diamond\}$ where $S'_{\texttt{MClient}}$ comes from typing the HO code containing s, and:

$$S'_{\texttt{MClient}} = ![\texttt{string}].![\texttt{string}].![\texttt{int}].?[\texttt{int}].?[\texttt{double}].\mathsf{end}$$

and similarly we type $s\!:\!\mathtt{rtt}$ with $\{s : \mathtt{int}\}$. The $\mathtt{Agent}\ \overline{s} \triangleright \{\mathsf{move} : \ldots, \mathsf{local} : \ldots\}$ is typed with (Bra) under session environment $\{\overline{s} : \&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}]\}$. The above session environments can be synthesised using $\odot$ to obtain:

$$\{s : (S'_{\mathtt{MClient}}, \mathtt{int}), \overline{s} : (\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}], \mathsf{move} \cdot \mathsf{unit} \multimap \diamond)\}$$

Now we use the rules in § 4.2 to calculate the session remainder of each queue:

$$S'_{\mathtt{MClient}} - \mathtt{int} = ![\mathtt{string}].![\mathtt{string}].![\mathtt{int}].?[\mathtt{double}].\mathsf{end}$$
$$\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}] - \mathsf{move} \cdot \mathsf{unit} \multimap \diamond = S'_{\mathtt{Agent}}$$

and we have $![\mathtt{string}].![\mathtt{string}].![\mathtt{int}].?[\mathtt{double}].\mathsf{end} \leqslant_c \overline{S'_{\mathtt{Agent}}}$. Finally, we can apply (New_s) and complete the derivation. We can also check that the similar example in § 1, $\overline{s}!\langle \ulcorner \overline{s}?(x).\overline{s}!\langle x \rangle \urcorner \rangle.\overline{s}?(z_1).\overline{s}?(z_2).\mathbf{0}$, is untypable since we cannot compose the session environments which include $\overline{s}$ both in the sent thunk and in the continuation.

5 Communication Safety and Algorithmic Subtyping

5.1 Type Soundness and Communication Safety

This section studies the key properties of our typing system. First we show that typed processes enjoy subject reduction and communication safety.

We begin by introducing *balanced environments* which specify the conditions of composable environments of runtime processes.

Definition 5.1 (Balanced Δ). $\mathsf{balanced}(\Delta)$ holds if whenever $\{s : (S_1, \vec{\tau}_1), \overline{s} : (S_2, \vec{\tau}_2)\} \subseteq \Delta$ with $S_1 - \vec{\tau}_1 = S'_1$ and $S_2 - \vec{\tau}_2 = S'_2$, then $S'_1 \leqslant_c \overline{S'_2}$.

The definition is based on (New_s) in the runtime typing system (Fig. 6): intuitively, all subprocesses generated from an initial typable program should conform to the balanced condition. We next define the ordering between the session environments which abstractly represents an interaction at session channels.

Definition 5.2 (Δ Ordering). Recall $\odot$ defined in § 4.2. We define $\Delta \sqsubseteq_s \Delta'$ as follows:

$$s : ?[U].S \odot s : U\vec{\tau} \sqsubseteq_s s : S \odot s : \vec{\tau} \qquad s : \&[l_i : S_i]_{i \in I} \odot s : l_k\vec{\tau} \sqsubseteq_s s : S_k \odot s : \vec{\tau} \quad k \in I$$
$$s : ![U].S \odot \overline{s} : \vec{\tau} \sqsubseteq_s s : S \odot \overline{s} : \vec{\tau}U \qquad s : \oplus[l_i : S_i]_{i \in I} \odot \overline{s} : \vec{\tau} \sqsubseteq_s s : S_k \odot \overline{s} : \vec{\tau}l_k \quad k \in I$$
$$s : \mu\mathbf{t}.S \odot s' : \vec{\tau} \sqsubseteq_s s : S' \odot s' : \vec{\tau}' \quad \text{if} \quad s : S[\mu\mathbf{t}.S/\mathbf{t}] \odot s' : \vec{\tau} \sqsubseteq_s s : S' \odot s' : \vec{\tau}'$$
$$\Delta \odot \Delta_1 \sqsubseteq_s \Delta \odot \Delta_2 \quad \text{if} \quad \Delta_1 \sqsubseteq_s \Delta_2 \text{ and } \Delta \odot \Delta_1 \text{ defined}$$

Note that if $\Delta_1 \sqsubseteq_s \Delta_2$ and $\Delta \odot \Delta_1$ is defined, then $\Delta \odot \Delta_2$ is defined; and if $\mathsf{balanced}(\Delta)$ and $\Delta \sqsubseteq_s \Delta'$ then $\mathsf{balanced}(\Delta')$. Then by the standard substitution lemmas, we have:

Theorem 5.3 (Type Soundness).

1. *Suppose* $\Gamma; \Delta; \mathcal{L} \vdash P : \diamond$. *Then* $P \equiv P'$ *implies* $\Gamma; \Delta; \mathcal{L} \vdash P' : \diamond$.
2. *Suppose* $\Gamma; \Delta; \emptyset \vdash P : T$ *with* $\mathsf{balanced}(\Delta)$. *Then* $P \longrightarrow P'$ *implies* $\Gamma; \Delta'; \emptyset \vdash P' : T$ *and either* $\Delta = \Delta'$ *or* $\Delta \sqsubseteq_s \Delta'$.

We now formalise communication-safety (which subsumes the usual type-safety). First, an *s-queue* is a queue process $s:\vec{h}$. An *s-input* is a process of the shape $s?(x).P$ or $s \triangleright \{l_i : P_i\}_{i\in I}$. An *s-output* is a process $s!\langle V\rangle.P$ or $s \triangleleft l.P$. Then, an *s-process* is an *s-queue*, *s-input* or *s-output*. Finally, an *s-redex* is a parallel composition of either an *s-input* and non-empty *s-queue*, or an *s-output* and $\bar{s}$-*queue*.

Definition 5.4 (Error Process). We say P is an *error* if $P \equiv (\nu\,\vec{a})(\nu\,\vec{s})(Q \mid R)$ where Q is one of the following: (a) a $|$-composition of two *s-processes* that does not form either an *s-redex* or an *s-input* and an empty *s-queue*; (b) an *s-redex* consisting an *s-input* and *s-queue* such that $Q = s?(x).Q' \mid s:l_k\vec{h}$ or $Q = s \triangleright \{l_i : P_i\}_{i\in I} \mid s:V\vec{h}$; (c) an *s-process* for $s \in \vec{s}$ with $\bar{s}$ not free in R or Q; (d) a prefixed process or application containing an *s-queue*.

The above says that a process is an error if (a) it breaks the linearity of s by having e.g. two *s-inputs* in parallel; (b) there is communication-mismatch; (c) there is no corresponding opponent process for a session; or (d) it encloses a queue under prefix, thus making it unavailable. As a corollary of Theorem 5.3, we achieve the following general communication-safety theorem, subsuming the case that P is an initial program.

Theorem 5.5 (Communication Safety). *If* $\Gamma;\Delta;\mathcal{L} \vdash P : \diamond$ *with* $\mathsf{balanced}(\Delta)$*, then* P *never reduces into an error.*

5.2 Algorithmic Higher-Order Asynchronous Subtyping

This subsection proposes an algorithmic subtyping, extending the method from [12, § 3]. While the inclusion of the higher-order sessions and functional types complicates the proof of soundness, the basic idea of the rules and proofs stays as before. First we prove the decidability of $S \ll S'$, introducing the rewriting rule $S \overset{!}{\mapsto} S'$ which moves the output action to the head (using $\ll$ in the reverse direction). Similarly for $S \overset{\oplus}{\mapsto} S'$. For a simple example, let $S_0 = \&[l_1 :?[U_1].![U_2].\mathsf{end},\ l_2 :![U_2].\mathsf{end}]$. Then $S_0 \overset{!}{\mapsto} \&[l_1 : ![U_2].?[U_1].\mathsf{end},\ l_2 :![U_2].\mathsf{end}] \overset{!}{\mapsto} ![U_2].\&[l_1 :?[U_1].\mathsf{end}, l_2 : \mathsf{end}]$ by applying $(\mathsf{OI},\mathsf{OB})$ in Fig. 4 in the reverse direction. Since $\overset{!}{\mapsto}$ and $\overset{\oplus}{\mapsto}$ terminate, and $S \ll S'$ can be decomposed into a finite sequence of $S \overset{\pi_1}{\mapsto} \cdots \overset{\pi_n}{\mapsto} S'$ $(n \geq 0)$ with $\pi_i \in \{!, \oplus\}$, the decidability of $\ll$ is straightforward. Using this relation, we can define the derivability of judgement $\Sigma \vdash T \leqslant T'$ where Σ is a sequence of assumed goals in the subtyping derivation. We list only the key output rule which is used together with the standard output rule [4, 5]:

$$\text{(Out)}\ \frac{\Sigma \vdash ![U_1].S_1 \leqslant ![U_2].\mathcal{T}[S_{2h}]^{h\in H} \quad \mathcal{T}[![U_2].S_{2h}]^{h\in H} \overset{!}{\mapsto} ![U_2].\mathcal{T}[S_{2h}]^{h\in H} \quad S_1 \bowtie \mathcal{T}[S_{2h}]^{h\in H}}{\Sigma \vdash ![U_1].S_1 \leqslant \mathcal{T}[![U_2].S_{2h}]^{h\in H}}$$

where $\mathcal{T}[S_h]^{h\in H}$ represents a h-hole context; and $T \bowtie T'$ means that T and T' have the same session constructors under matching recursions; and labels in each type are distinct. This rule reads: we fix the subtype and apply $\overset{!}{\mapsto}$ to place $![U_2]$ to the head; then we can use the standard output rule. As an example, let $S_1 = \&[l_1 :?[U_1].\mathsf{end}, l_2 : \mathsf{end}]$. Then we can derive $![U_2].S_1 \leqslant S_0$ (S_0 is given above) by using (Out). The algorithm is applied to the initial goal $\emptyset \vdash T \leqslant T'$. Then using the same method developed in [5, 12], we can prove the subtyping algorithm always terminates. We conclude this section with the following theorem (see [1]):

Theorem 5.6 (Soundness and Completeness of the Algorithmic Subtyping). *For all closed types T and T' with $T \bowtie T'$, $T \leqslant_c T'$ if and only if $\emptyset \vdash T \leqslant T'$.*

6 Related and Future Work

The asynchronous subtyping has been firstly studied in [12] for multiparty session types [8]; this work does not support neither *higher-order sessions* (delegations) nor *code mobility* (higher-order functions). Both of these features provide powerful abstractions for structured distributed computing; delegation is the key primitive in our implementation of session types in Java [9] and web service protocols [17, 18], to which we can now apply our theory for flexible optimisation. The proof of the transitivity in this paper requires more complex construction of the transitive closure $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ (Definition 3.4) than the one in [12] due to the higher-order constructs. In spite of the richness of the type structures, we proposed more compact runtime typing and proved communication safety in the presence of higher-order code, which is not presented in [12]. Note that our new typing system *subsumes* the previous linear typing system in [11], demonstrating a smooth integration of two kinds of type-directed optimisation.

Coinductive subtyping of recursive session types is first studied in [5], adapting the standard methods for the IO-subtyping in the π-calculus [14]. The system of [5] does not provide any form of asynchronous permutation, thus does not need the nested n-times unfolding (Definition 3.1). Our transitivity proof and the algorithmic subtyping are more involved than [5] due to the incorporation with n-time unfolding and higher-order functions.

Our treatment of runtime typing, specifically our method for typing session queues and the use of *session remainders*, is more compact than previous asynchronous session works [2, 3, 8] where they use the method of *rolling-back* messages – the head type of a queue typing *moves* to the prefix of the session type of a process using the queue, and then compatibility is checked on the constructed types. Our method is simpler, as we remove type elements appearing in a queue from its typing. On the other hand, our queue typing is more similar to that of the functional language in [6], where smaller types are obtained after *matching* with buffer values. Our method works with queue types rather than with values directly, hence it can be extended smoothly to handle asynchronous optimisation, which is not treated in [6]. For example we allow a type consisting an output followed by an input action to be reduced with a type corresponding to the input, leaving the output prefix intact. Using a more delicate composition between values and queue typing, our system enables linear mobile code to be stored in the queues.

We intend to integrate the improved methods from this work back to our original subtyping method for multiparty sessions [12], extending it to higher-order multiparty sessions. Another direction is *progress* [2], by which we mean deadlock-free execution of multiple interleaved sessions: in the presence of higher-order code mobility, this extension is challenging since it requires tracking dependencies inside mobile code. For example, if $s!\langle\ulcorner P \urcorner\rangle$ is blocked, the sessions inside $\ulcorner P \urcorner$ are also blocked. On the other hand, we postulate that asynchronous subtyping does not introduce deadlock to a deadlock-free supertype, as outputs and selections can only be done in advance (partial commutativity), satisfying even stricter input dependencies than those required by the dual session of the supertype.

References

1. On-line Appendix of this paper. `www.doc.ic.ac.uk/~mostrous/hopiasync`.
2. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
3. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.
4. 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.
5. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Info.*, 42(2/3):191–225, 2005.
6. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types, October 2008. Submitted for publication.
7. 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, 1998.
8. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
9. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
10. R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
11. D. Mostrous and N. Yoshida. Two Session Typing Systems for Higher-Order Mobile Processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.
12. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009. available from `www.doc.ic.ac.uk/~mostrous/asyncsub`.
13. The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/intermediate/main.htm.
14. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.
15. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.
16. 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-Verlag, 1994.
17. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.
18. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.
19. N. Yoshida. Channel dependency types for higher-order mobile processes. In *POPL '04*, pages 147–160. ACM Press, 2004. Full version available at www.doc.ic.ac.uk/~yoshida.
20. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisit. In *SecRet'06*, volume 171(3) of *ENTCS*, pages 127–151. Elsevier, 2007.