



Contents lists available at ScienceDirect

## Information and Computation

www.elsevier.com/locate/yinco



# Session typing and asynchronous subtyping for the higher-order $\pi$ -calculus

Dimitris Mostrous<sup>a,\*</sup>, Nobuko Yoshida<sup>b</sup>

<sup>a</sup> Departamento de Informática, Universidade de Lisboa, Portugal

<sup>b</sup> Department of Computing, Imperial College London, UK

## ARTICLE INFO

### Article history:

Received 11 December 2013

Received in revised form 24 October 2014

Available online xxxx

### Keywords:

Session types

The higher-order  $\pi$ -calculus

Asynchronous subtyping

Communication optimisation

Code mobility

Linear typing

## ABSTRACT

This paper proposes a session typing system for the higher-order  $\pi$ -calculus (the  $\text{HO}\pi$ -calculus) with asynchronous communication subtyping, which allows partial commutativity of actions in higher-order processes. The system enables two complementary kinds of optimisation, mobile code and asynchronous permutation of session actions, within processes that utilise structured, typed communications. Our first contribution is a session typing system for the  $\text{HO}\pi$ -calculus using techniques from the linear  $\lambda$ -calculus. Integration of arbitrary higher-order code mobility and sessions leads to technical difficulties in type soundness, because linear usage of session channels and completion of sessions are required. Our second contribution is to introduce an asynchronous subtyping system which uniformly deals with type-manifested asynchrony and linear functions. The most technical challenge for subtyping is to prove the transitivity of the subtyping relation. We also demonstrate the expressiveness of our typing system with an e-commerce example, where optimised processes can interact respecting the expected sessions.

© 2015 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

*The higher-order  $\pi$ -calculus with session types* In global computing environments, applications are executed across multiple distributed sites or devices. The use of mobile code is prominent in such environments, where several participants are synthesised by communication of not only passive values but also of runnable code: for example a service can be delegated to different participants, by sending either a channel via which it is accessible, or code that accesses it; and incoming code may transit through several devices that alter their computational behaviour or their data through interaction with it. Indeed, mobile code has become really pervasive at many levels. For example when we speak of “software updates,” we are in fact referring to mobile code, and we use it in mobile phone applications, operating systems, and all kinds of networked applications.

The Higher-Order  $\pi$ -calculus ( $\text{HO}\pi$ -calculus) [46] is a general formalism of interaction in which two kinds of mobility, name passing and process passing, are integrated in a simple and universal form: in this model, processes can be instantiated by names and other processes, just like a piece of mobile code is instantiated with local capability after migration. This additional expressiveness inherited from the  $\lambda$ -calculus provides a powerful basis for describing and analysing dynamic behaviour in global computing scenarios.

\* Corresponding author.

E-mail addresses: [dimitris@fc.ul.pt](mailto:dimitris@fc.ul.pt) (D. Mostrous), [n.yoshida@imperial.ac.uk](mailto:n.yoshida@imperial.ac.uk) (N. Yoshida).

<http://dx.doi.org/10.1016/j.ic.2015.02.002>

0890-5401/© 2015 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

As a type-theoretic foundation for highly structured communication protocols often found in distributed applications, this paper focuses on the notion of *sessions* and their types [48,54,24]. A session is a series of communications between two parties which form a meaningful logical unit, just like a web session between a browser and a server when a human user interacts with an e-commerce site. Session types model such interactions as an abstract structure of typed choice, inputs and outputs. The study of session typing systems is now wide-spread due to the need for structured communications in various scenarios in distributed computing. While many advanced session types for the  $\pi$ -calculus and programming languages have been studied, before our work [35] there existed no session typing systems for the  $\text{HO}\pi$ -calculus. Incorporation of sessions into this language offers a general theoretical basis for examining the interplay between two non-trivial features in communication-based programming, higher-order mobility and session-based structured interaction.

As the first contribution, this article establishes the first session type theory for the  $\text{HO}\pi$ -calculus which can statically validate the type safety of complex distributed scenarios with code mobility. In spite of their simple type syntax, the previous literature have shown that obtaining type soundness for session types is an intricate task because of delegation of sessions [54]. Preservation of typability becomes even more non-trivial in the presence of higher-order process passing, especially when the mobile processes contain free sessions.

*Higher-order processes with asynchronous sessions* We now outline technical challenges by examples. Code mobility in  $\text{HO}\pi$ -calculus is facilitated by sending not just 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:

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

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

$$(\nu s)(s!\langle 5 \rangle.s!\langle \text{true} \rangle.s?(y).(y() \mid R) \mid \bar{s}?(z_1).\bar{s}?(z_2).(\bar{s}!\langle \ulcorner P \urcorner \rangle \mid Q) \mid s:\epsilon \mid \bar{s}:\epsilon)$$

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

$$(\nu s)(s?(y).(y() \mid R) \mid \bar{s}?(z_1).\bar{s}?(z_2).(\bar{s}!\langle \ulcorner P \urcorner \rangle \mid Q) \mid s:\epsilon \mid \bar{s}:5 \cdot \text{true})$$

and in two more steps the right process receives the values and becomes  $\bar{s}!\langle \ulcorner P \urcorner \{5/z_1\} \{ \text{true}/z_2 \} \rangle \mid Q \{5/z_1\} \{ \text{true}/z_2 \}$ . Similarly the next step transmits the thunked process, and  $R$  can interact with  $P$  locally.

The session types  $S_1$  of  $s$  and  $S_2$  of  $\bar{s}$ :

$$S_1 = ![\text{nat}].![\text{bool}].?[U].\text{end} \quad S_2 = ?[\text{nat}].?[\text{bool}].![U].\text{end}$$

where  $U$  is the type of  $\ulcorner P \urcorner$ , have the property  $S_1 = \overline{S_2}$  derived from a duality relation on types, and this guarantees that values are communicated in a complementary order.

*Asynchronous communication optimisation with code mobility* The main idea of optimisation by message permutation, in the context of buffered communications, is that outputs can be performed in advance without affecting correctness with regards to the delayed inputs. This is based on the fact that there are two buffers per session (as there are two streams per socket in network programming) which means that we only need to preserve the relative order of outputs (resp. inputs) to avoid communication mismatches. In the previous example, suppose the size of  $P$  is very large and it does not contain  $z_1$  and  $z_2$ , for example because they appear in  $Q$  and the program is not optimised. Then, if  $\bar{s}$  does not appear in  $P$ , the right process might wish to start transmission of  $P$  to  $s:\epsilon$  concurrently without waiting for the delivery of 5 and  $\text{true}$  on  $\bar{s}:\epsilon$ . Thus, we can send  $\ulcorner P \urcorner$  ahead obtaining  $\bar{s}!\langle \ulcorner P \urcorner \rangle.\bar{s}?(z_1).\bar{s}?(z_2).Q$  where  $\bar{s}$  now has the type  $S'_2 = ![U].?[\text{nat}].?[\text{bool}].\text{end}$ . The interaction with the left process is still *safe* since both  $s$  and  $\bar{s}$  continue to receive the expected type of value and in the expected order, specifically  $s$  will receive  $U$  and  $\bar{s}$  will receive first  $\text{nat}$  then  $\text{bool}$ . However, the optimised code is not composable with the other party by the original session system [48] since it cannot be assigned  $S_2$  for  $\bar{s}$  which is the only type such that  $S_1 = \overline{S_2}$ . To make this optimisation valid, we proposed *asynchronous subtyping* in [37] by which we can refine a protocol to maximise asynchrony without violating the session. For example, in the above case,  $S'_2$  is an asynchronous subtype of  $S_2$ , written  $S'_2 \leq_c S_2$ , so the optimised process can also be assigned  $S_2$ , and can therefore compose with the left process as before. Unsafe optimisations, such as one where the left process sends values in a different order, first  $![\text{bool}]$  and then  $![\text{nat}]$ , are filtered out by the typing system, otherwise  $z_1$  of type  $\text{nat}$  would receive a value of type  $\text{bool}$ .

The idea of this subtyping is intuitive and the combination of two kinds of optimisations is vital for typing many practical protocols [50,23] and parallel algorithms [38], but it requires subtle formal formulations due to the presence of higher-order code. The linear functional typing permits to send a value that contains free session channels: for example,  $\bar{s}!\langle \ulcorner P \urcorner \rangle$  can be  $\bar{s}!\langle \ulcorner s'?(x).s'!\langle x \rangle \urcorner \rangle$  or even  $\bar{s}!\langle \ulcorner \bar{s}?(x).\bar{s}!\langle x \rangle \urcorner \rangle$  which contains its own session (if  $R$  conforms with the dual session, e.g.,

(Identifiers)		$u, v, w ::=$		$k ::=$	
		$x, y, z$	variables	$x, y, z$	variables
		$  a, b, c$	shared channels	$  s, \bar{s}$	session channels
(Terms)				(Values)	
$P, Q, R ::=$				$V ::=$	
	$V$	value			$u, v, w$ shared
	$  u(x).P$	server			$  k$ linear
	$  \bar{u}(x).P$	client			$  ()$ unit
	$  k?(x).P$	input			$  \lambda(x:U).P$ function
	$  k!(V).P$	output			$  \mu(x:U \rightarrow T).\lambda(y:U).P$ recursion
	$  k \triangleright \{l_1:P_1, \dots, l_n:P_n\}$	branching		(Message Values)	
	$  k \triangleleft l.P$	selection			$h ::=$
	$  P   Q$	parallel			$l$ label
	$  (va : \langle S \rangle) P$	restriction			$  V$
	$  (vs) P$	restriction		(Abbreviations)	
	$  P Q$	application			$\top \dashv \dashv \stackrel{\text{def}}{=} \lambda(x:\text{unit}).P \ (x \notin \text{fv}(P))$ thunk
	$  \mathbf{0}$	nil process			$\text{run} \stackrel{\text{def}}{=} \lambda x.(x())$ run
	$  k:\bar{h}$	queue			

Fig. 1. Syntax.

$R = s!(\bar{7}).s?(z).\mathbf{0}$ ). In the first case, we can permute the output  $\bar{s}!(\top \dashv \dashv)$  as explained, but in the second case it would be unsafe, since the input action  $\bar{s}?(x)$  from the thunk will appear in parallel with  $\bar{s}?(z_1).\bar{s}?(z_2).Q$ , creating a race condition, as seen in:

$$(vs)(\bar{s}?(x).\bar{s}!(x) \mid R \mid \bar{s}?(z_1).\bar{s}?(z_2).Q \mid s:\epsilon \mid \bar{s}:5 \cdot \text{true})$$

This article shows that the combination of two optimisations is indeed possible by establishing soundness and communication-safety. 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 positions, both covariantly and contravariantly. Moreover, the definitions are now exposed in detail. Another challenge is to formulate a runtime typing system which handles both stored higher-order code with open sessions and asynchronous subtyping. We demonstrate all aspects of type-preserving optimisations explained above by using e-commerce scenarios.

**Outline** This article is a full version of the extended abstracts published in two conference papers [35,36] and the first author's PhD thesis [32]. Here it includes the detailed definitions, expanded explanations, more detailed examples, and complete proofs. We have also updated the related work with recent literature. In the rest of the article, Section 2 defines the syntax, operational semantics, and demonstrates the combined use of sessions, code mobility and asynchronous optimisation with examples. Section 3 defines types and Section 4 introduces the asynchronous subtyping. Section 5 illustrates the typing system for programs and Section 6 extends it to the typing system for runtime processes. Section 7 proves the main theorems, type soundness and communication safety of the typed processes. Section 8 discusses related work and Section 9 concludes the article. Appendices A–C list the detailed definitions and proofs which are omitted from the main sections.

## 2. The higher-order $\pi$ -calculus with asynchronous sessions

The  $\text{HO}\pi$ -calculus with asynchronous sessions,  $\text{HOS}\pi$ , is a variant of the  $\text{HO}\pi$ -calculus [46]. There are two notable differences compared to [46]. First, in  $\text{HOS}\pi$  communications occur in the context of an initiated session synchronising two processes that perform a prescribed protocol. Second, communications are buffered in *message queues*, to realise asynchronous FIFO semantics.  $\text{HOS}\pi$  encompasses two types of mobility: name passing, with which dynamic communication topologies can be programmed, and code passing, where by transmitting processes a dynamic behaviour can be achieved. Note that the calculus is monadic, i.e., only one value is sent/received at each communication step, but this does not affect the results and serves for simplicity.

### 2.1. Syntax

The syntax of  $\text{HOS}\pi$  is given in Fig. 1. The calculus extends the  $\text{HO}\pi$  with a small kernel of session primitives: a way to initiate a session over a shared channel, a class of session names — which we call *endpoints* — used for communications within sessions, and primitives for offering and making choices indexed by labels.

**Identifiers** Variables range over  $x, y, z, \dots$ . Shared channel names, which are used only to initiate sessions (we describe this in detail further below), are ranged over  $a, b, c, \dots$ . We write  $u, v, w, \dots$  to represent shared identifiers, that is, those that are either variables or shared channel names. Session channels, ranged over  $s, \dots$  and  $\bar{s}, \dots$ , are the *endpoints* through which values are communicated *within* an established session (which as we shall see is always between exactly two processes).

The name  $\bar{s}$  denotes the *dual* of  $s$ , that is, if one process in a session uses  $s$ , the other process uses  $\bar{s}$ , and in this way each of the two processes possess a unique endpoint. This separation of endpoints is similar to the use of two *polarities* in [19,54]. We define duality to be idempotent, thus, we have that  $\bar{\bar{s}} = s$ . This property of endpoint names is used in the reduction semantics, where a communication is synchronised over the two endpoints of a session. We write  $k, k', k'', \dots$  for linear identifiers, consisting of variables and session channels.

**Values** We write  $V, V', W, \dots$  for those terms that may be used as values, that is, as the object of a communication or as the argument in function application. First, we have identifiers, shared and linear (as standard). Abstraction, written  $\lambda(x:U).P$ , encapsulates a process  $P$ , where  $x$  may occur free, into a function over  $x$  (with type annotation  $U$ ). This is the basic mechanism for the exchange of processes, and the unit  $()$  is useful when we wish to obtain a value from an arbitrary process  $P$ : take a variable  $x$  not free in  $P$ , then  $\lambda(x:\text{unit}).P$  is a value, usually referred to as a *thunk*, and abbreviated to  $\ulcorner P \urcorner$ . To reveal and execute a process within a thunk, we use the *run* function  $\lambda(y:\text{unit} \rightarrow \diamond).(y())$  which takes a thunk as argument and applies it to the unit value to obtain the hidden process.

To facilitate terms that exhibit infinitary behaviour, we introduce a recursive function constructor  $\mu(x:U \rightarrow T).\lambda(y:U).P$ . In this fixpoint representation, instances of the variable  $x$  within  $P$  represent the function itself.

**Terms** Terms range over  $P, Q, R, \dots$ . The main constructs are:

**Session initialisation**  $u(x).P$  and  $\bar{u}(x).Q$  are prefixed processes that may synchronise and commence a session. The interactions will adhere to the session type assigned to the shared identifier  $u$ , and since each session consists of two endpoints used in a complementary way, we distinguish the two different behaviours with respect to this type using  $u$  and  $\bar{u}$ . The bound variable  $x$  is a placeholder for a fresh session endpoint, initialised after the prefixes react to establish a session.

**Input and output**  $k?(x).P$  is the standard input prefixed process, with linear subject  $k$  and using  $x$  as a placeholder for the received value.  $k!(V).P$  is an output prefixed process, sending value  $V$  over session  $k$ .

**Branching and selection**  $k \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$  offers a set of label-indexed choices  $l_i : P_i$  on endpoint  $k$ , with a process continuation  $P_i$  corresponding to each label  $l_i$ . It is often written  $k \triangleright \{l_i : P_i\}_{i \in I}$  with index set  $I$ . The dual (or co-action) of a branch is a process ready to perform a selection  $k \triangleleft l.P$  where the chosen label is within the domain of the branch set. Essentially a branching is an input expecting a label and performing case analysis (which covers all cases) on this label to choose a continuation. Dually, a selection is an output of a label designating a choice. Clearly, it is undesirable to allow the empty set in branching, since no selection can be made (that is, there is no effective co-action), and henceforth we assume that there is at least one branch (and the respective indexing sets, when used, are non-empty).

**Fresh names** We write  $(\nu a : \langle S \rangle)P$  to denote a process  $P$  in which the shared channel  $a$  (typed by  $\langle S \rangle$ ) is unique. With  $(\nu s)P$  we denote that the two endpoints  $s$  and  $\bar{s}$  are unique in  $P$ , that is, no external process can perform a session action on either of these endpoints; this gives non-interference within a session.

**Message queues** A message queue  $s : \bar{h}$  provides access, via a session that uses  $s$ , to the ordered messages  $\bar{h}$ . It can be thought of as a network pipe in a TCP-like transport mechanism. The messages can be values, or labels which are required for selection and branching.

Other constructs are the nil process  $\mathbf{0}$ , parallel composition  $P \mid Q$ , and functional application  $PQ$ , which are standard from  $\pi$ -calculus and  $\lambda$ -calculus.

We often omit  $\mathbf{0}$  and some type annotations when not relevant.

The *bindings* are induced by  $(\nu a : \langle S \rangle)P$ ,  $(\nu s)P$ ,  $u(x).P$ ,  $\bar{u}(x).P$ ,  $k?(x).P$ ,  $\lambda(x:U).P$ , and  $\mu(x:U \rightarrow T).\lambda(y:U).P$ . The derived notions of bound and free identifiers, alpha equivalence and substitution are mostly standard. We write  $\text{fv}(P)/\text{fn}(P)$  for the set of free variables/names, respectively, extended to queue processes (which can contain labels) as follows; the complete definition is in Fig. A.13.

$$\text{fn}((\nu s)P) = \text{fn}(P) \setminus \{s, \bar{s}\} \quad \text{fn}(\mathbf{0}) = \emptyset \quad \text{fn}(s : h_1 \dots h_n) = (\cup_{i \in 1..n} \text{fn}(h_i)) \cup \{s\}$$

As usual, in all mathematical contexts we assume Barendregt's variable convention, that is, free and bound variables are always chosen to be different, and all bound variables are distinct; the same applies to names.

Note that queues and session restrictions appear only at our formalisation of *runtime* systems, since programmers do not normally write protocols with “open” sessions. Furthermore, we use the terminology *program* for a process which does not contain such runtime elements.

## 2.2. Reduction semantics

We define the standard structural congruence, denoted  $\equiv$ , as the smallest equivalence relation which is congruent with respect to the calculus constructors (parallel composition, name restriction, prefixes) and respects the axioms and rules in Fig. 2. The only non-standard rule is for *garbage collecting* queues from completed sessions:  $(\nu s)(s : \bar{s} : \epsilon) \equiv \mathbf{0}$ .

$$\begin{aligned}
P \equiv_{\alpha} Q &\Rightarrow P \equiv Q & P \mid Q &\equiv Q \mid P & (P \mid Q) \mid R &\equiv P \mid (Q \mid R) & P \mid \mathbf{0} &\equiv P \\
(\nu a : \langle S \rangle) P \mid Q &\equiv (\nu a : \langle S \rangle) (P \mid Q) & a \notin \text{fn}(Q) & & (\nu a : \langle S \rangle) (\nu s) P &\equiv (\nu s) (\nu a : \langle S \rangle) P \\
(\nu s) P \mid Q &\equiv (\nu s) (P \mid Q) & s, \bar{s} \notin \text{fn}(Q) & & (\nu a : \langle S \rangle) (\nu b : \langle S' \rangle) P &\equiv (\nu b : \langle S' \rangle) (\nu a : \langle S \rangle) P \\
(\nu s) (\nu s') P &\equiv (\nu s') (\nu s) P & (\nu a : \langle S \rangle) \mathbf{0} &\equiv \mathbf{0} & (\nu s) \mathbf{0} &\equiv \mathbf{0} & (\nu s) (s : \epsilon \mid \bar{s} : \epsilon) &\equiv \mathbf{0}
\end{aligned}$$

Fig. 2. Structural congruence.

$$\begin{aligned}
(\lambda(x : U).P)V &\longrightarrow P\{V/x\} & (\text{beta}) \\
(\mu y.\lambda x.P)V &\longrightarrow P\{V/x\}\{\mu y.\lambda x.P/y\} & (\text{rec}) \\
k!(V).P \mid \bar{k}:\vec{h} &\longrightarrow P \mid \bar{k}:\vec{h} \cdot V & (\text{send}) \\
k?(x).P \mid k:V \cdot \vec{h} &\longrightarrow P\{V/x\} \mid k:\vec{h} & (\text{recv}) \\
k \triangleleft l.P \mid \bar{k}:\vec{h} &\longrightarrow P \mid \bar{k}:\vec{h} \cdot l & (\text{sel}) \\
k \triangleright \{l_i : P_i\}_{i \in I} \mid k:l_m \cdot \vec{h} &\longrightarrow P_m \mid k:\vec{h} & (m \in I) \quad (\text{bra}) \\
a(x).P \mid \bar{a}(z).Q &\longrightarrow (\nu s) (P\{s/x\} \mid Q\{\bar{s}/z\} \mid s : \epsilon \mid \bar{s} : \epsilon) \quad (\star) & (\text{conn}) \\
&& (\star) s \notin \text{fn}(P, Q)
\end{aligned}$$

$$\begin{aligned}
\frac{P \longrightarrow P'}{PQ \longrightarrow P'Q} & \quad \frac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'} & (\text{app} - l, \text{app} - r) \\
\frac{P \longrightarrow P'}{(\nu s)P \longrightarrow (\nu s)P'} & \quad \frac{P \longrightarrow P'}{(\nu a : \langle S \rangle)P \longrightarrow (\nu a : \langle S \rangle)P'} & (\text{ress}, \text{resc}) \\
\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} & \quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} & (\text{par}, \text{str})
\end{aligned}$$

Fig. 3. Reduction.

The single-step call-by-value reduction, denoted  $\longrightarrow$ , is a binary relation from closed terms to closed terms, defined by the rules in Fig. 3. Rule (beta) is standard from the call-by-value  $\lambda$ -calculus. The case of (rec) is similar, with the added step of unfolding the recursive function, by substituting it in place of the variable  $y$  within the function body  $P$ .

Rule (conn) establishes a new session between two processes  $a(x).P$  and  $\bar{a}(z).Q$  ready to synchronise on  $a$ . The result of this rewriting is a parallel composition of the session bodies  $P$  and  $Q$  with a fresh set of endpoints  $s$  and  $\bar{s}$  substituted for the session variables  $x$  and  $z$ , respectively. The side condition ensures that the new endpoints do not already appear free in either  $P$  or  $Q$ . The result contains empty queues corresponding to the session channels ( $\epsilon$  denotes the empty sequence).

Rules (send) and (sel) respectively enqueue a value or label at the tail of the queue for the dual endpoint  $k$ . When  $V$  is a function, we have *higher-order code passing*; when  $V$  is a session endpoint, we call it *higher-order session passing*. Rules (recv) and (bra) dequeue, from the head of the queue, a value or label. The rule (recv) substitutes value  $V$  for  $x$  in  $P$ , while (bra) selects the corresponding branch for index  $m$ . The received label  $l_m$  must be in the branch set as indicated by the side condition. Due to the self-inverse duality property of endpoints, if  $k = s$  then we have an output from  $\bar{s}$  to  $s$ , and if  $k = \bar{s}$ , the output is from  $s$  to  $\bar{s}$ .

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*).

In the remaining rules: (app – l) and (app – r) implement a left to right reduction order for functional application; (par) reduces the leftmost parallel process; (resc) and (ress) are standard and reduce a process under name hiding. The last rule, (str), introduces structural congruence [31] into the reduction relation. This is necessary for re-arranging terms to match reduction rules.

With “ $\twoheadrightarrow$ ” we denote the multi-step reduction defined as  $(\equiv \cup \longrightarrow)^*$ .

**Encoding replication** By using recursion, we can represent infinite behaviours of processes such as, e.g., the definition agent  $\text{def}$ , or the replication  $!u(x).P$  of [30,54,24,35]. Replication on a shared name, useful for defining persistent servers, can be encoded as follows:

$$!u(x).P \stackrel{\text{def}}{=} (\mu y.\lambda z.z(x).(P \mid yz))u \quad \text{taking } y, z \notin \text{fv}(P)$$

Hereafter when writing a replicated connection-prefixed process we shall mean that this encoding is used. Note that we did not (and by typing we cannot) replicate a session endpoint, since that would violate linearity. To validate the encoding, we can observe a reduction using a replicated connection  $!a(x).P$  and a suitable co-action  $\bar{a}(z).Q$ :

$$\begin{aligned}
!a(x).P \mid \bar{a}(z).Q &\longrightarrow a(x).(P \mid !a(x).P) \mid \bar{a}(z).Q \\
\longrightarrow (\nu s) (P\{s/x\} \mid !a(x).P \mid Q\{\bar{s}/z\}) &\equiv (\nu s) (P\{s/x\} \mid Q\{\bar{s}/z\}) \mid !a(x).P
\end{aligned}$$

Note that in the application of rule (conn), since  $x$  is bound in  $!a(x).P$ , the substitution  $\{s/x\}$  has no effect on this subterm. Once a connection is established via (conn), we can apply structural congruence  $\equiv$  to obtain a term where  $!a(x).P$  can react again; for this we used the fact that  $s$  and  $\bar{s}$  do not occur free in  $!a(x).P$ , which is ensured by the conditions of the previous reduction with (conn).

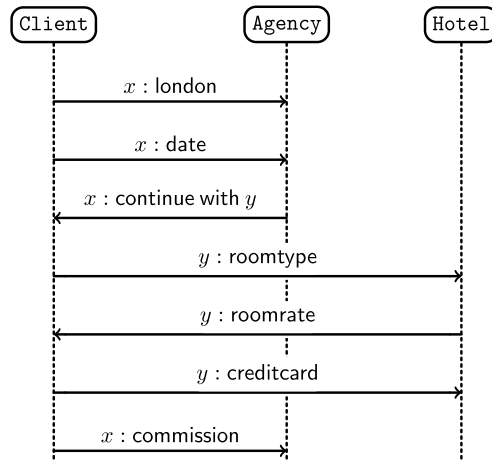


Fig. 4. Hotel booking.

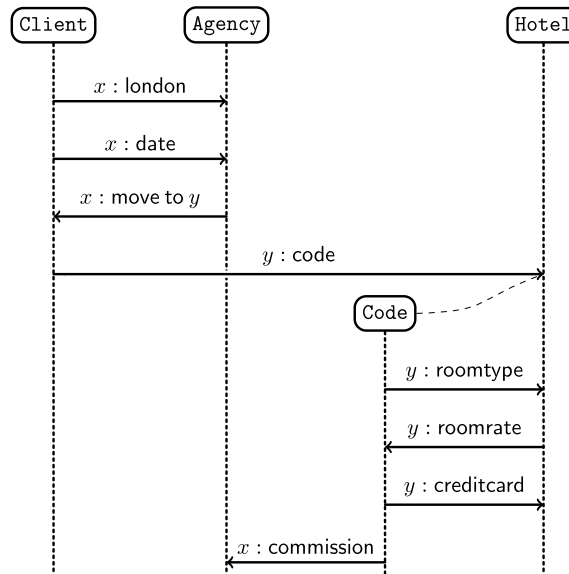


Fig. 5. Hotel booking with mobile code.

### 2.3. Example: business protocol with code mobility

We show a simple protocol which contains essential features by which we can demonstrate the expressivity of the code mobility and session primitives for the  $\text{HOS}\pi$ -calculus; it consists of a combination of code mobility, session delegation and branching. This extends a typical collaboration pattern that appears in many web service business protocols [23,8,47] to code mobility.

In Fig. 4, we show the sequence diagram for a protocol which models a hotel booking: first, Booking Agency and Client initiate interaction at session  $x$  over channel  $a$ ; then Client starts exchanging a series of information with Agency; during this initial communication, Agency calculates its Round Trip Time (RTT) between Client and Agency; Agency selects an appropriate Hotel and creates a new session  $y$  over channel  $b$  with that Hotel. If the RTT is short (Fig. 4), then Agency delegates to Client its part of the remaining activity with Hotel, by sending session channel  $y$  to Client; then Client and Hotel continue negotiations by message passing. If the RTT is long (Fig. 5), since many remote interactions increase the communication time as well as the danger of communication failures, Agency asks Client to send mobile code to the Hotel (via  $y$ ) which contains the communications pertaining to the Client's room plan and negotiation behaviour. Client sends the code to Hotel, then Hotel runs it locally, finishing a series of interactions in its location. Finally, Agency receives a commission fee (10 percent of the room rate) via session  $x$ , concluding the transaction.

The given scenario is straightforwardly encoded in our calculus, where session primitives make the structure of interactions clearer. Agency first initiates at  $a$  and starts the interactions with Client; then it initiates at  $b$  and establishes



session  $y$ ; next it invokes either label `cont` or label `move` in `Client` depending on the RTT and sends  $y$  (higher-order session passing) to it, and waits for completion of the transaction between `Client` and `Hotel` at  $x$  (“if-then-else” can be encoded using branching, and we use other base types and their operators).

$$\begin{aligned} \text{Agency} &\stackrel{\text{def}}{=} !a(x).x?(area).x?(date) \dots \bar{b}(y) . \\ &\quad \text{if } rtt < 100 \\ &\quad \text{then } x \triangleleft \text{cont} . x!\langle y \rangle . x?(z) . P \end{aligned} \quad (1)$$

$$\text{else } x \triangleleft \text{move} . x!\langle y \rangle . x?(z) . P \quad (2)$$

`Client` requests a service at  $a$  and starts a series of interactions with `Agency`, and either continues the remaining activity with `Hotel` or sends the code (a thunk in [Line 4](#)). Note that `Client` can safely send the commission fee back to `Agency` because the return message  $\bar{x}(z \times 0.1)$  which uses session channel  $x$  is embedded in the thunk.

$$\begin{aligned} \text{Client} &\stackrel{\text{def}}{=} \bar{a}(x).x!\langle \text{London} \rangle \dots \\ x \triangleright \{ &\quad \text{cont} : x?(y).y \triangleleft \text{cont}.y!\langle \text{roomtype} \rangle.y?(z) \dots x!\langle z \times 0.1 \rangle , \end{aligned} \quad (3)$$

$$\text{move} : x?(y).y \triangleleft \text{move}.y!\langle \ulcorner y!\langle \text{roomtype} \rangle.y?(z) \dots x!\langle z \times 0.1 \rangle \urcorner \rangle \} \quad (4)$$

`Hotel` performs the interactions with `Agency` and `Client` via a single session at  $y$  (by the facility of higher-order sessions). In [Line 6](#), the code sent by `Client` is run locally.

$$\begin{aligned} \text{Hotel} &\stackrel{\text{def}}{=} \\ !b(y).y \triangleright \{ &\quad \text{cont} : y?(z).y!\langle \text{roomrate}(z) \rangle . Q ; \end{aligned} \quad (5)$$

$$\text{move} : y?(code).(run\ code \mid y?(z).y!\langle \text{roomrate}(z) \rangle . Q) \} \quad (6)$$

This example demonstrates a couple of subtle points whose slight modification would violate the expected “complementarity” of session actions, leading to obvious violations of soundness. First, in [Line 4](#), if we send code which does not complete the session, e.g. if we have interactions at  $y$  (say  $y!\langle w \rangle$ ) *after* sending the thunk in [Line 4](#) of `Client`, the session at  $y$  will eventually appear in three threads (two in `Hotel`, one in `Client`), so values may get mixed up due to the non-determinism on  $y$ -actions. Secondly, in [Line 6](#), if we have two or more applications (say  $run\ code \mid run\ code$ ) instead of exactly one  $run\ code$ , we will end up with duplication of session endpoints (both  $y$  and  $x$ ). Finally, if the code is not activated in [Line 6](#) (for example if we use  $(\lambda x.0)code$  instead of  $run\ code$ ), the other end of the session,  $y?(z).y!\langle \text{roomrate}(z) \rangle . Q$ , will never find a matching output. Hence, the variable  $code$  must appear exactly once and become instantiated into a process exactly once. We type this example in [Section 6.2](#).

#### 2.4. Example: optimised business protocol with code mobility

We now show a business protocol which integrates the two kinds of type-safe optimisation: code mobility, by which a protocol can be executed at the location of the receiver, which is especially useful when latency is high; and also message re-ordering, which allows an implementation to perform outputs in advance, essentially permitting both participants of a session to send *at the same time*. We thus extend the previous protocol to highlight the behaviours that are possible using our methods. [Fig. 6](#) 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.

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 credit card number ( $card$ ) before finding out the rate.

To represent this optimised scenario, we start from the process for `Agent` (which is a simplification of `Agency`):

$$\text{Agent} = a(x).x!\langle rtt \rangle . x \triangleright \{ \text{move} : x?(code).(run\ code \mid Q) , \text{local} : Q \}$$

$$Q = x?(hotel).x?(roomtype).x!\langle rate \rangle . x?(creditcard) \dots$$

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:

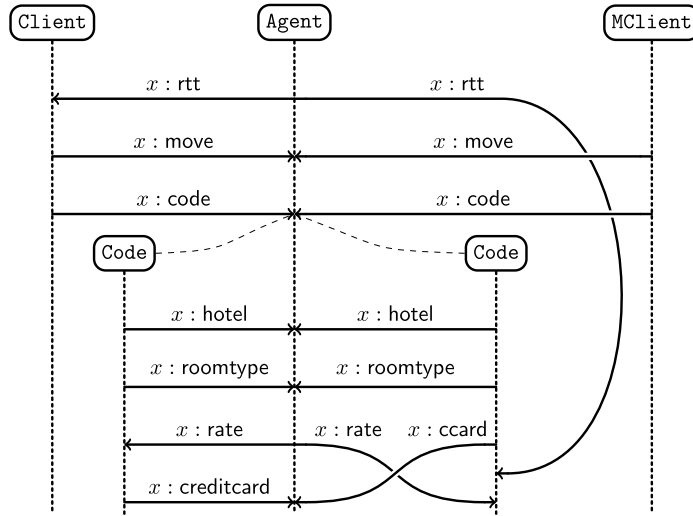


Fig. 6. Standard (left) and optimised (right) interaction for hotel booking.

Term		Session	
$T ::= U$	value	$S ::= ![U].S$	output
$  \diamond$	process	$  ?[U].S$	input
Value	$U ::= \text{unit}$	$  \oplus[l_1:S_1, \dots, l_n:S_n]$	selection
	$  U \rightarrow T$	$  \&[l_1:S_1, \dots, l_n:S_n]$	branching
	$  U \multimap T$	$  \mathbf{t}$	type variable
	$  \langle S \rangle$	$  \mu \mathbf{t}.S$	recursion
	$  S$	$  \text{end}$	ending

Fig. 7. Types.

$$\text{Client} = \bar{a}(x).x?(rtt).x \triangleleft \text{move}.x!(\langle x!(\text{ritz}).x!(\text{suite}).x?(rate).x!(card) \dots \rangle)$$

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

$$\text{MClient} = \bar{a}(x).x \triangleleft \text{move}.x!(\langle x!(\text{ritz}).x!(\text{suite}).x!(card).x?(rtt).x?(rate) \dots \rangle)$$

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.

As explained in the previous subsection, even without subtyping, the typing of sessions in the HO $\pi$ -calculus poses delicate conditions; in the present system, we can further verify that the optimisation of MClient does not violate communications safety: when values are received they are always of the expected type, conforming to a new *subtyping* relation given in Section 4. Optimisation by permutation is very delicate, for example as explained in the introduction we cannot optimise  $\bar{s}?(z_1).\bar{s}?(z_2).\bar{s}!(\langle \bar{s}!(5) \rangle).\mathbf{0}$  into  $\bar{s}!(\langle \bar{s}!(5) \rangle).\bar{s}?(z_1).\bar{s}?(z_2).\mathbf{0}$ , because the thunk in the first process contains the sender's session (on  $\bar{s}$ ) and a permutation to the left (before the inputs) will cause interference as explained in the previous example. In fact, the second term is untypable.

### 3. Higher-order linear types

This short section presents the syntax of the types, which combine linear functions, unrestricted functions, and session types.

#### 3.1. Types

The syntax of types is given on Fig. 7. It is an integration of the types from the simply typed  $\lambda$ -calculus with unit and the session types from the  $\pi$ -calculus. Term types range over  $T$ , and can be value types, ranging over  $U$ , or the process type  $\diamond$ . Value types consist of the unit type unit, the type  $U \rightarrow T$  of shared functions, the type  $U \multimap T$  of linear functions, the type  $S$  of sessions, and the shared channel type  $\langle S \rangle$  which enforces that sessions initiated on the corresponding channel will follow the protocol defined by  $S$ .



$$\frac{\overline{!U}.S = ?[U].\bar{S} \quad \overline{?U}.S = ![U].\bar{S} \quad \bar{t} = t \quad \overline{\mu t}.S = \mu t.\bar{S} \quad \text{end} = \text{end}}{\oplus[l_1:S_1, \dots, l_n:S_n] = \&[l_1:\bar{S}_1, \dots, l_n:\bar{S}_n] \quad \&[l_1:S_1, \dots, l_n:S_n] = \oplus[l_1:\bar{S}_1, \dots, l_n:\bar{S}_n]}$$

Fig. 8. Type duality.

The session types are defined inductively as follows. The type  $!U.S$  represents the sending of a value of type  $U$ , followed by the remaining session  $S$ . Dually, with  $?[U].S$  the action will be to receive a value of expected type at least  $U$ , followed by  $S$  as before. The selection type  $\oplus[l_1:S_1, \dots, l_n:S_n]$  signifies that one of the choices  $l_1, \dots, l_n$  will be made (operationally this is an output of a label), and depending on this label the corresponding session continuation chosen from  $S_1, \dots, S_n$  will take place. The co-type of selection is the branch type  $\&[l_1:S_1, \dots, l_n:S_n]$  corresponding to the reception of a label followed by the corresponding continuation type as in selection. Recursive session types are written  $\mu t.S$ , where the type variable  $t$  is bound and may occur free in  $S$ . We only consider contractive recursive types [18,54]. Practically, contractiveness of  $\mu t.S$  means that every free instance of  $t$  in  $S$  is *guarded* under at least one input, output, selection or branching constructor. For example  $\mu t.[\text{nat}].t$  is contractive, but  $\mu t.\mu t'.t$  is not. Moreover, we only consider *tail-recursive* session types, therefore types such as  $\mu t.[t].\text{end}$  are not well-formed. To indicate that a session is finished, we use the terminal end.

We write  $\mathcal{T}$  for the set of types.

**Abbreviated forms** We often write  $\&[l_i:S_i]_{i \in I}$  and  $\oplus[l_i:S_i]_{i \in I}$  for branching and selection types,  $\ulcorner T \urcorner$  for unit  $\rightarrow T$  and  $\urcorner T \urcorner$  for unit  $\leftarrow T$ . The terminal end is sometimes omitted.

**More general recursive types** Our restriction to tail-recursive types may cause a slight limitation with regards to expressiveness, and as noted in [3] there are safe processes that are not tail-recursive. For example, if we were to encode a data type such as a tree with elements of type  $T$ , we would need a type of the shape  $\mu t. \oplus [\text{tree} : ![\mathbf{t}].![\mathbf{t}].\text{end}, \text{leaf} : ![\mathbf{t}].\text{end}]$ . The first branch uses recursion to send the left and right subtrees to the client (which will have a dual type). However, we can easily lift the restriction without changing anything substantial with respect to subtyping, and with minor modifications to some definitions (e.g., Definition 4.1 which defines how recursive types are *unfolded*), and so it serves for simplicity.

### 3.2. Examples of types

Session types can encode many common interactions. For example the following type can be used to iterate through a list containing elements of type  $U$ :

$$\mu t. \oplus [\text{hasnext} : \&[\text{next} : ?[U].t, \text{finished} : \text{end}], \text{finished} : \text{end}]$$

The type describes the behaviour of the client process accessing the list: first a choice is made, either to query the list and discover if it has more elements, by choosing *hasnext*; or alternatively the choice *finished* can be made in which case the protocol reaches its end. If *hasnext* is chosen, then the list can respond by choosing *next*, after which the client can receive a value of type  $U$ . Moreover the type variable  $t$  signifies that at this point the protocol is repeated from the point of definition, that is, from the  $\mu$ -binder at the beginning. If the list replies by choosing *finished*, the protocol is complete.

Abstractions that contain running sessions must be used exactly once, which demonstrates the difference between linear and unrestricted functional types:

1.  $(\lambda(x:U).x \cdot ()) \cdot \ulcorner s!(5).\mathbf{0} \urcorner$

This term is safe, since the thunk which contains  $s$  is used exactly once within the function that receives it. To denote linear usage, the argument has type  $U = \text{unit} \rightarrow \diamond$ .

2.  $(\lambda(x:U').\mathbf{0}) \cdot \ulcorner \bar{a}(x).x!(5).\mathbf{0} \urcorner$

Although the function disappears after the application, this term is safe, because even if the thunk will not be used in the function, it does not contain any linear or session element that needs to be preserved. Hence, the argument must have type  $U' = \text{unit} \rightarrow \diamond$ . These examples are easy to check with the typing rules in Section 5.

**Duality** In the above example (Section 3.2) we show the type of the iterator, but not of the list. In fact the list's type can be obtained by *duality*. Each session type  $S$  has a *dual* type, denoted by  $\bar{S}$ , which describes *complementary* behaviour. This is inductively defined by the rules in Fig. 8. Essentially, dualisation interchanges input ( $?$ ) with output ( $!$ ), branching ( $\&$ ) with selection ( $\oplus$ ), leaving *end*, type variables and  $\mu$  binders unchanged. Duality is idempotent. Note that we do not need to define duality for other types such as function types, as these are never dualised.

### 4. Higher-order asynchronous subtyping

This section presents our theory of asynchronous session subtyping: reordered communications between two processes, in the presence of higher-order values and session mobility, can preserve the type-safety of the original protocol.

As we have seen in the introduction, asynchronous subtyping allows processes to perform output actions (which include selections) in advance within the same session, taking advantage of the underlying buffered model of communication.

Thus, we enable certain permutations of inputs with outputs. However, a permutation of two inputs or two outputs is not permissible because it can violate type-safety. Suppose:

$$P = s!(2).s!(\text{true}).s?(x).\mathbf{0} \text{ and } Q = \bar{s}(y).\bar{s}(z).\bar{s}!(y+2).\mathbf{0}.$$

These processes interact correctly. If we permute the outputs of  $P$  to get  $P' = s!(\text{true}).s!(2).s?(x).\mathbf{0}$ , then the parallel composition  $(P' \mid Q)$  causes a type-error. By duality, it is easy to understand why two inputs cannot be permuted. Moreover, an alteration in the relative order of inputs and outputs such that an input is done in advance may cause deadlock, losing progress in session  $s$ . For example, consider exchanging  $s!(\text{true})$  and  $s?(z)$  in  $P$ , obtaining:

$$P'' = s!(2).s?(x).s!(\text{true}).\mathbf{0} \text{ and } Q = \bar{s}(y).\bar{s}(z).\bar{s}!(y+2).\mathbf{0}.$$

Obviously  $(P'' \mid Q)$  ends with deadlock, since the two inputs (the second action on both  $P''$  and  $Q$ ) are blocked after the initial prefixes interact. The only way to optimise the communication within a session is to place outputs before inputs, for example:

$$P = s!(2).s!(\text{true}).s?(x).\mathbf{0} \text{ and } Q' = \bar{s}(y).\bar{s}!(y+2).\bar{s}(z).\mathbf{0}.$$

The communication in  $Q'$  is optimised and  $(P \mid Q')$  is type-safe.

#### 4.1. Asynchronous subtyping

We begin with some preliminary notions. An occurrence of a type constructor *not* under a recursive prefix in a recursive type is called a *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.

Consider the following types:

$$S_1 = ![U_1].?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$$

$$S_2 = \mu\mathbf{t}.?[U_2].![U_1].\mathbf{t}$$

Intuitively, we want to include  $S_1$  in the subtypes of  $S_2$ , because in the infinite expansion of the types any action of  $S_1$  can be matched to one in  $S_2$ . The first output  $![U_1]$  of  $S_1$  needs to be matched with a copy of the same output obtained after unrolling the recursion in  $S_2$  once, resulting in:

$$S'_2 = ?[U_2].![U_1].S_2$$

This unrolling is necessary because under the  $\mu$  binder every action is repeated, and by unrolling once we can obtain one of the possibly infinite instances of the action. For this strategy to succeed, we need to obtain the output  $![U_1]$  in  $S'_2$  which is *guarded* under the input action  $?[U_2]$ . Then, the input action can be compared, and the remaining types checked, following the standard coinductive method.

To summarise, in asynchronous coinductive subtyping we need to formalise both the *unfolding* of a type and also the type *contexts* specifying the top-level actions that may guard an output (or selection).

We generalise the type unfolding function defined in [19] so that it can be applied to guarded types, yielding the following definition, based on [37]:

**Definition 4.1** (*n-Time unfolding*).

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

For any recursive type  $S$ ,  $\text{unfold}^n(S)$  is the result of inductively unfolding the top level recursion up to a fixed level of nesting. For example:

$$\text{unfold}^1(?[U].\mu\mathbf{t}.![U'].\mathbf{t}) = ?[U].![U'].\mu\mathbf{t}.![U'].\mathbf{t}$$

$$\text{unfold}^2(?[U].\mu\mathbf{t}.?[U].\mu\mathbf{t}'.![U'].\mathbf{t}') =$$

$$\text{unfold}^1(?[U].?[U].\mu\mathbf{t}'.![U'].\mathbf{t}') = ?[U].?[U].![U'].\mu\mathbf{t}'.![U'].\mathbf{t}'$$

From the definition we have that  $\text{unfold}^1(\text{unfold}^n(S)) = \text{unfold}^n(\text{unfold}^1(S))$ , even though normally we apply from the outside. Also, since recursive types are not unfolded until they become guarded, but only  $n$ -times,  $\text{unfold}^n(S)$  terminates. Moreover, because our recursive types are contractive, there is no need to apply unfolding indefinitely to obtain a guarded type.

Then, we proceed to define the contexts corresponding to a nested structure of top-level input actions (where branching is treated like input in the sense that a label is to be received). The rationale is that a supertype is less asynchronous than a subtype, hence may consist of input actions before any outputs that need to be checked first, based on the prefix of the subtype. Thus, the multi-hole *asynchronous contexts* are defined as follows:

**Definition 4.2** (*Asynchronous contexts*).

$$\mathcal{A} ::= \langle \cdot \rangle^{h \in H} \mid ?[U].\mathcal{A} \mid \&[l_i : \mathcal{A}_i]_{i \in I}$$

We write  $\mathcal{A}\langle S_h \rangle^{h \in H}$  for the context  $\mathcal{A}$  with holes indexed by  $h \in H$ , where each hole  $\langle \cdot \rangle^{h \in H}$  is substituted with  $S_h$ . For example, taking  $H = \{1, 2\}$  and

$$\mathcal{A} = \&[l_1 : ?[U].\langle \cdot \rangle^{1 \in H}, l_2 : \langle \cdot \rangle^{2 \in H}]$$

we obtain

$$\mathcal{A}(\![U']\!.S_h)^{h \in H} = \&[l_1 : ?[U].\![U']\!.S_1, l_2 : \![U']\!.S_2]$$

To formalise subtyping in the presence of recursive types a *simulation*-based (or *coinductive*) method is used, in which subtyping is determined by membership of the goal within a binary relation on types. We adapt the standard simulation approaches from [19,44,11], extending the method non-trivially to account for asynchrony.

**Definition 4.3** (*Asynchronous subtyping*). A relation  $\mathfrak{R} \in \mathcal{T} \times \mathcal{T}$  is an asynchronous type simulation if  $(T_1, T_2) \in \mathfrak{R}$  implies the following:

1. If  $T_1 = \diamond$ , then  $T_2 = \diamond$ .
2. If  $T_1 = \text{unit}$ , then  $T_2 = \text{unit}$ .
3. If  $T_1 = U \rightarrow T$ , then  $T_2 = U \rightarrow T$ .
4. If  $T_1 = U \multimap T$ , then  $T_2 = U \multimap T$ .
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  $T_1 = \text{end}$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = \text{end}$ .
7. If  $T_1 = \![U_1]\!.S_1$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = \mathcal{A}(\![U_2]\!.S_{2h})^{h \in H}$  with  $(U_2, U_1) \in \mathfrak{R}$  and  $(S_1, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{R}$ .
8. If  $T_1 = ?[U_1].S_1$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = ?[U_2].S_2$ ,  $(U_1, U_2) \in \mathfrak{R}$  and  $(S_1, S_2) \in \mathfrak{R}$ .
9. If  $T_1 = \oplus[l_i : S_{1i}]_{i \in I}$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = \mathcal{A}(\oplus[l_j : S_{2jh}]_{j \in J_h})^{h \in H}$  and  $\forall h \in H. I \subseteq J_h$  and  $\forall i \in I. (S_{1i}, \mathcal{A}(S_{2ih})^{h \in H}) \in \mathfrak{R}$ .
10. If  $T_1 = \&[l_i : S_{1i}]_{i \in I}$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = \&[l_j : S_{2j}]_{j \in J}$ ,  $J \subseteq I$  and  $\forall j \in J. (S_{1j}, S_{2j}) \in \mathfrak{R}$ .
11. If  $T_1 = \mu t.S$ , then  $(\text{unfold}^1(T_1), T_2) \in \mathfrak{R}$ .

The coinductive subtyping  $T_1 \leq_c T_2$  (read:  $T_1$  is an *asynchronous subtype* of  $T_2$ ) is defined when there exists a type simulation  $\mathfrak{R}$  with  $(T_1, T_2) \in \mathfrak{R}$ . Formally,  $\leq_c$  is the largest type simulation, defined as the union of all type simulations.

Most cases are similar to the ones in [37,11], but in order to facilitate the asynchronous rules the unfolding of the supertype is performed at each case for some level  $n$ . (1–4, 6) are the base cases, while (5) says that the shared channel type is invariant (as in the standard session types [19,37,24]).

Now we focus on the new rules: in (7), an output prefix of  $T_1$  can be simulated when  $T_2$  can be unfolded to obtain a type that has an output hidden under an asynchronous context  $\mathcal{A}$ , which by definition consists of only inputs and branchings. The type  $U_1$  is compared to  $U_2$ , the first available top-level output; this is contravariant which is standard in  $\pi$ -calculus [44]. Then, the continuation  $S_1$  of  $T_1$  is compared with the type  $\mathcal{A}(S_{2h})^{h \in H}$  consisting of the asynchronous context in which the output(s) have been removed, since they were matched with the output prefix of  $T_1$ . For the input in (8), we do not use any context, since the input must appear as the first action after unfolding. No action can appear before the desired input at the supertype: if there is a branching (which is a form of input, with labels as values) it is not comparable, and if there is an output or selection then  $T_2$  cannot be a supertype of the input-prefixed type  $T_1$ , since it would be intuitively more asynchronous.

In (9), selection is defined similarly to output and any label appearing in  $T_1$  must be included in the top level selections of the asynchronous context derived from  $T_2$ . Note that in the supertype, each hole in the context may use a different indexing set  $I_h$ , but the set  $I$  of the subtype is smaller than all these sets ( $\forall h \in H. I \subseteq I_h$ ). Dually to selection, in (10), branching is defined like input and any labelled branch of (the unfolding of)  $T_2$  must be supported in  $T_1$ . Finally (11) forces  $T_1$  to be unfolded until it becomes a guarded type.

*Remark* To include subtyping between base types, we would need to follow [32] where we employ a slightly more elaborate definition, in which for all types *except session types* output is covariant and input is contravariant. There, we define:

$$(S, S')^{\otimes} = (S', S) \quad (T, T')^{\otimes} = (T, T') \quad \text{if } T, T' \text{ are not session types}$$

The subtyping simulation in [32] has the following output-input clauses:

7. If  $T_1 = ![U_1].S_1$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = \mathcal{A}(![U_2].S_{2h})^{h \in H}$  with  $(U_1, U_2)^{\otimes} \in \mathfrak{N}$  and  $(S_1, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{N}$ .
8. If  $T_1 = ?[U_1].S_1$ , then for some  $n$ ,  $\text{unfold}^n(T_2) = ?[U_2].S_2$ ,  $(U_2, U_1)^{\otimes} \in \mathfrak{N}$  and  $(S_1, S_2) \in \mathfrak{N}$ .

In this definition output appears covariant, but because of the inversion applied *only to session types* it becomes, in this case, contravariant. This explains why our present definitions show contravariant output subtyping (unit and other invariant types are not affected). Now, if we consider the types `int` and `real` with  $\text{int} \leq_c \text{real}$ , then we have  $(\text{int}, \text{real})^{\otimes} = (\text{int}, \text{real})$ , i.e., *no inversion*, hence in (7) above we would obtain a covariant subtyping. For example, we would have  $![\text{int}].\text{end} \leq_c ![\text{real}].\text{end}$ , and not the opposite which would be non-sensical.

The usual subtyping for functional types can also be integrated into (3, 4) using the above definition from [32], but it is orthogonal to our purposes and therefore omitted for simplicity.

#### 4.2. Examples of asynchronous subtyping

We show four small but representative examples which highlight key points of our subtyping relation. The first example shows that permuting outputs in advance of inputs in an infinite type preserves subtyping. The second example demonstrates that in some subtypings, a finite number of extra outputs can appear in the subtype, and dually, a finite number of extra inputs can appear in the supertype; this is acceptable when the total outputs remain infinite without losing type compatibility, and similarly for inputs. The third example demonstrates a case where  $n$ -level unfolding is required. The fourth example which is atypical exposes a class of subtypings that induce infinite simulation relations, due to asynchronous subtyping.

*Three typical examples* Consider the types given previously:

$$S_1 = ![U_1].?[U_2].\mu t.![U_1].?[U_2].t \quad S_2 = \mu t.?[U_2].![U_1].t$$

It is easy to verify that  $S_1 \leq_c S_2$  by checking that the following relation is a type simulation:

$$\mathfrak{N} = \{ (S_1, S_2), (U_1, U_1), (?[U_2].\mu t.![U_1].?[U_2].t, ?[U_2].S_2), \\ (U_2, U_2), (\mu t.![U_1].?[U_2].t, S_2) \}$$

It is also straightforward to show that for the following types:

$$S_3 = ![U_1].\mu t.![U_1].?[U_2].t \quad S_4 = ?[U_2].\mu t.?[U_2].![U_1].t$$

it holds that  $S_3 \leq_c S_4$  using the following simulation:

$$\mathfrak{N} = \{ (S_3, S_4), (U_1, U_1), (\mu t.![U_1].?[U_2].t, ?[U_2].S_4), \\ (![U_1].?[U_2].\mu t.![U_1].?[U_2].t, ?[U_2].S_4), \\ (?[U_2].\mu t.![U_1].?[U_2].t, ?[U_2].?[U_2].S_4), (U_2, U_2) \}$$

We can demonstrate easily that for the following types:

$$S_5 = \mu t.![U].?[U].\&[l_1 : t, l_2 : t] \\ S_6 = \mu t_1.?[U].\mu t_2.\&[l_1 : ![U].t_1, l_2 : ![U].t_1]$$

we have that  $S_5 \leq_c S_6$  with the following simulation:

$$\mathfrak{N} = \{ (S_5, S_6), (U, U), (\text{unfold}^1(S_5), S_6), \\ (?[U].\&[l_1 : S_5, l_2 : S_5], ?[U].\&[l_1 : S_6, l_2 : S_6]), \\ (\&[l_1 : S_5, l_2 : S_5], \&[l_1 : S_6, l_2 : S_6]) \}$$

in which the fourth pair (which is added when matching the output) is obtained after unfolding  $S_6$  at level  $n = 2$ , i.e., using  $\text{unfold}^2(S_6)$ ; this is because there are two  $\mu$ -binders guarding the asynchronous context where the output is located. Moreover, since as we prove in the next subsection  $\leq_c$  is transitive, we can also find a simulation  $\mathfrak{N}'$  such that:

$$(\mu \mathbf{t} . ! [U_1] . ? [U] . \& [l_1 : \mathbf{t}, l_2 : \mathbf{t}], \mu \mathbf{t}_1 . ? [U] . \mu \mathbf{t}_2 . \& [l_1 : ! [U_2] . \mathbf{t}_1, l_2 : ! [U_3] . \mathbf{t}_1]) \in \mathfrak{R}'$$

whenever  $(U_2, U_1) \in \mathfrak{R}'$  and  $(U_3, U_1) \in \mathfrak{R}'$ . For this the simulation will support the intermediate results

$$(\mu \mathbf{t} . ! [U_1] . ? [U] . \& [l_1 : \mathbf{t}, l_2 : \mathbf{t}], \mu \mathbf{t}_1 . ? [U] . \mu \mathbf{t}_2 . \& [l_1 : ! [U_1] . \mathbf{t}_1, l_2 : ! [U_1] . \mathbf{t}_1]) \in \mathfrak{R}'$$

and

$$(\mu \mathbf{t}_1 . ? [U] . \mu \mathbf{t}_2 . \& [l_1 : ! [U_1] . \mathbf{t}_1, l_2 : ! [U_1] . \mathbf{t}_1], \mu \mathbf{t}_1 . ? [U] . \mu \mathbf{t}_2 . \& [l_1 : ! [U_2] . \mathbf{t}_1, l_2 : ! [U_3] . \mathbf{t}_1]) \in \mathfrak{R}'$$

*A more controversial example* Consider the types:

$$S_7 = \mu \mathbf{t} . ! [U_1] . \mathbf{t} \quad S_8 = \mu \mathbf{t} . ! [U_1] . ? [U_2] . \mathbf{t}$$

Perhaps surprisingly, it holds that  $S_7 \leq_c S_8$ , as evidenced by the following simulation:

$$\begin{aligned} \mathfrak{R} &= \{ (U_1, U_1), \\ &\quad (S_7, S_8), \\ &\quad (! [U_1] . S_7, S_8), \\ &\quad (S_7, ? [U_2] . S_8), \\ &\quad (! [U_1] . S_7, ? [U_2] . S_8), \\ &\quad (S_7, ? [U_2] . ? [U_2] . S_8), \\ &\quad (! [U_1] . S_7, ? [U_2] . ? [U_2] . S_8), \\ &\quad \vdots \\ &\quad (S_7, ? [U_2]^{\omega} . S_8), \\ &\quad (! [U_1] . S_7, ? [U_2]^{\omega} . S_8) \} \\ &= \{ (U_1, U_1) \} \cup \{ (S_7, ? [U_2]^n . S_8), (! [U_1] . S_7, ? [U_2]^n . S_8) \mid n \in \mathbb{N} \} \end{aligned}$$

where  $? [U_2]^n . S_8$  is the type  $S_8$  prefixed with a sequence of  $n$  input actions  $? [U_2]$ . Effectively, the subtype is sending all the infinite outputs in advance, and never receives any values, i.e., it is taking asynchronous optimisation to the extreme. There are many similar examples, where the common denominator in all is the presence, within a recursion at the subtype, of a greater *proportion* of output actions (including selection) compared to the supertype. For instance,  $\mu \mathbf{t} . ! [U_1] . ! [U_1] . ? [U_2] . \mathbf{t} \leq_c \mu \mathbf{t} . ! [U_1] . ? [U_2] . \mathbf{t}$  also holds and can be shown with an infinite simulation relation.

The above examples may seem slightly pathological, since values received in a buffer may never be used by the process that owns it. For instance, by  $S_7 \leq_c S_8$  a process can record the interface  $s : S_8$  when it actually implements the behaviour  $s : S_7$ , and by duality it can interact with  $\bar{s} : \bar{S}_8 = \mu \mathbf{t} . ? [U_1] . ! [U_2] . \mathbf{t}$ . Clearly, the values of type  $U_2$  are received in the buffer  $s$  (by the outputs on  $\bar{s}$ ) but not in the program that implements  $s : S_7$ . As a consequence, in a naive implementation the buffer can increase in size indefinitely, which is undesirable and in some cases unsafe (e.g., buffer overflow). However, dealing with unreachable data is typically the job of a garbage collector, as in most mainstream languages, so we do not think this is a real problem.

*Type soundness and progress in the presence of asynchronous subtyping* As shown in the last example, a surprising property of our notion of asynchronous subtyping is that it allows an implementation to not actually receive all the values sent to its buffer. It is then natural to ask how this may affect the properties one expects from a sessions system.

First, *type safety* is not violated since no value of unexpected type is ever received within a term, because two inputs (resp. two outputs) on the same endpoint cannot be permuted. However, one property that can be affected is *progress*. Specifically, if a session on  $s$  or a linear function containing  $s$  is never received from a buffer — due to a subtyped process not performing the input at all — then a process waiting to perform an action on the dual endpoint  $\bar{s}$  may become *stuck*.<sup>1</sup> This situation is not easy to address in the present framework, because asynchronous optimisation means that we can postpone inputs ad infinitum, which is not so different than not having those inputs at all. On the other hand, the “standard” sessions systems only guarantee progress on a per session basis, allowing the interleaving of sessions even if it may cause deadlocks, so in that sense not much is lost. We should note that, if one wishes to ensure that all messages are received, there are some solutions: we can restrict subtyping as in [32, p. 181], or following the recent work [9], motivated in part by our subtyping; we return to this later.

<sup>1</sup> Issues pertaining to such “orphan” messages are also discussed in [12,13].

#### 4.3. The relation $\leq_c$ is a preorder

We conclude this section with the main theorem, stating that  $\leq_c$  is a preorder. In inductively defined subtyping systems, commonly presented as a set of deduction rules, transitivity is a property by definition [18,43]. In a coinductive setting, transitivity cannot be assumed, and not every simulation is guaranteed to contain the necessary hypotheses; however, we can prove that  $\leq_c$  is transitive by careful construction of supporting simulations, containing the necessary components up to unfolding and context manipulation.

If  $\leq_c$  was not transitive, there would not be type safety. The typical explanation is that, if there exists  $U_1 \leq_c U_2$  and  $U_2 \leq_c U_3$  such that  $U_1 \not\leq_c U_3$ , then from two consecutive applications of subsumption we may provide a value of type  $U_1$  when  $U_3$  is expected, which is unsafe when  $U_1 \not\leq_c U_3$ . For a detailed exposition to the issues arising from the use of coinductive definitions in subtyping, see Chapter 21 of [43].

The standard method of relational composition [19] is not enough for proving the transitivity of  $\leq_c$ . The difficulty is that, given  $S_1 \leq_c S_2$  and  $S_2 \leq_c S_3$ , we need to find a subtyping relation that includes enough elements to justify  $S_1 \leq_c S_3$  directly. However, due to the use of nested  $n$ -times unfolding with manipulation of asynchronous contexts,  $S_1 \leq_c S_2$  provides insufficient information which cannot be straightforwardly combined with the hypotheses from  $S_2 \leq_c S_3$  to obtain the result.

Our objective is to discover how to obtain the “missing elements,” and to achieve it we gradually formalise a set of extensions on simulations, essentially monotonous functions from simulations to simulations, and then utilise them to prove the main result, [Theorem 4.4](#), stating that  $\leq_c$  is a preorder.

**Theorem 4.4** ( $\leq_c$  is a preorder). *The relation  $\leq_c$  is reflexive and transitive.*

**Overview of proof.** The proofs of [Theorem 4.4](#) are given in [Appendix B](#).

Specifically, we perform the following steps:

1. We prove as standard that unfolding  $S_1$  or  $S_2$  or both in  $S_1 \leq_c S_2$  preserves subtyping. We formalise the *unfolding extension* of a simulation to include such  $n$ -times unfoldings. ([Lemmas B.1 and B.2](#), [Definition B.3](#), [Proposition B.1](#).)
2. We define a class of *single-step permutation contexts* representing an input/branching guarded type. Then we formalise rules for moving an output/selection appearing within such a context (that is, immediately after the initial input/branching), to the position ahead of it. This represents the finest granularity of permutation since it is not defined to be transitive. ([Definition B.4](#).)
3. The *contextual extension* of a simulation is defined, which is a supporting construction. It is necessary in order to obtain the subtypings that arise when removing an output/selection from a single-step permutation context, thus changing its original structure. ([Definition B.5](#) and [Lemma B.6](#).)
4. The *asynchronous extension* of degree  $n$  is defined by applying  $n$  consecutive single-step permutations on the subtypes in a simulation relation, and up to contexts  $\mathcal{A}$  (that is, also deep within the structure of types). Both the contextual and the unfolding extensions are necessary to prove that this relation is also a simulation. ([Definition B.7](#) and [Lemma B.8](#).)
5. Multi-step permutations that can extract an output/selection from deep within a context  $\mathcal{A}$ , placing it ahead of all actions (that is, prefixing  $\mathcal{A}$ ), are shown to be included in the asynchronous extension of degree  $\omega$ . This is effectively a proof that the transitive application of nested single-step permutations is included in the asynchronous extension. ([Corollary B.9](#).)
6. The *transitivity connection* of two simulations is then defined, utilising a composition of asynchronous extensions for the given simulations. The proof that the transitivity connection is a simulation implies that  $\leq_c$  is transitive. ([Definition B.10](#) and [Lemma B.11](#).)
7. The relation  $\leq_c$  is shown to be a preorder: reflexivity is easy to obtain using straightforward techniques, and transitivity is proved directly by utilising the result for transitivity connections. ([Theorem 4.4](#).)

## 5. Higher-order linear session typing system

### 5.1. Typing system

We now present the typing system, which combines techniques from linear  $\lambda$ -calculus and session typing, integrating the asynchronous subtyping from the previous section. The system presented here is for *initial programs*, i.e., for terms without any queues or already activated sessions. We will augment the type system later, so as to also cover the *runtime* constructs.

**Environments** We first define three kinds of finite mappings for environments, needed when typing a term with free identifiers:

$$(\text{Shared}) \ \Gamma ::= \emptyset \mid \Gamma, u : \text{unit} \mid \Gamma, u : U \rightarrow T \mid \Gamma, u : \langle S \rangle$$

$$(\text{Linear}) \ \Lambda ::= \emptyset \mid \Lambda, x : U \multimap T \quad (\text{Session}) \ \Sigma ::= \emptyset \mid \Sigma, k : S$$



(Common)

$$\begin{array}{c}
\text{(UNIT)} \\
\hline
\Gamma; \emptyset; \emptyset \vdash () : \text{unit} \\
\\
\text{(LVAR)} \\
\hline
\Gamma; \{x : U \multimap T\}; \emptyset \vdash x : U \multimap T \\
\\
\text{(SHARED)} \\
\hline
\Gamma, u : U; \emptyset; \emptyset \vdash u : U \\
\\
\text{(SESSION)} \\
\hline
\Gamma; \emptyset; \{k : S\} \vdash k : S
\end{array}$$

(Structure, Subtyping)

$$\begin{array}{c}
\text{(PROMOTION)} \\
\hline
\Gamma; \emptyset; \emptyset \vdash P : U \multimap T \\
\Gamma; \emptyset; \emptyset \vdash P : U \rightarrow T \\
\\
\text{(DERELICTION)} \\
\hline
\Gamma; \Lambda, x : U \multimap T; \Sigma \vdash P : T' \\
\Gamma, x : U \rightarrow T; \Lambda; \Sigma \vdash P : T' \\
\\
\text{(SUB)} \\
\hline
\Gamma; \Lambda; \Sigma \vdash P : T \quad \Sigma \leq_c \Sigma' \quad T \leq_c T' \\
\Gamma; \Lambda; \Sigma' \vdash P : T'
\end{array}$$

(Functional)

$$\begin{array}{c}
\text{(ABS)} \\
\hline
\Gamma; \Lambda; \Sigma \vdash x : U \vdash P : T \\
\Gamma; \Lambda; \Sigma \vdash \lambda(x : U).P : U \multimap T \\
\\
\text{(REC)} \\
\hline
\Gamma, x : U \rightarrow T; \emptyset; \emptyset \vdash \lambda(y : U).P : U \rightarrow T \\
\Gamma; \emptyset; \emptyset \vdash \mu(x : U \rightarrow T).\lambda(y : U).P : U \rightarrow T \\
\\
\text{(APP)} \quad T' = U \rightarrow T \text{ or } T' = U \multimap T \\
\hline
\Gamma; \Lambda_1; \Sigma_1 \vdash P : T' \quad \Gamma; \Lambda_2; \Sigma_2 \vdash Q : U \\
\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \vdash P Q : T
\end{array}$$

Fig. 9. Linear session typing: common and functional rules.

(Process)

$$\begin{array}{c}
\text{(NIL)} \\
\hline
\Gamma; \emptyset; \emptyset \vdash 0 : \diamond \\
\\
\text{(NEW)} \\
\hline
\Gamma, a : \langle S \rangle; \Lambda; \Sigma \vdash P : \diamond \\
\Gamma; \Lambda; \Sigma \vdash (\nu a : \langle S \rangle).P : \diamond \\
\\
\text{(CONN)} \\
\hline
\Gamma; \emptyset; \emptyset \vdash u : \langle S \rangle \quad \Gamma; \Lambda; \Sigma, x : S \vdash P : \diamond \\
\Gamma; \Lambda; \Sigma \vdash u(x).P : \diamond \\
\\
\text{(CONNDUAL)} \\
\hline
\Gamma; \emptyset; \emptyset \vdash u : \langle \bar{S} \rangle \quad \Gamma; \Lambda; \Sigma, x : S \vdash P : \diamond \\
\Gamma; \Lambda; \Sigma \vdash \bar{u}(x).P : \diamond \\
\\
\text{(RECV)} \\
\hline
\Gamma; \Lambda; \Sigma, k : S \vdash x : U \vdash P : \diamond \\
\Gamma; \Lambda; \Sigma, k ? [U].S \vdash k?(x).P : \diamond \\
\\
\text{(SEND)} \\
\hline
\Gamma; \Lambda_1; \Sigma_1 \vdash P : \diamond \quad \Gamma; \Lambda_2; \Sigma_2 \vdash V : U \quad k : S \in \Sigma_i \quad i = 1 \text{ or } i = 2 \\
\Gamma; \Lambda_1, \Lambda_2; (\Sigma_1, \Sigma_2) \setminus \{k : S\}, k : ! [U].S \vdash k!(V).P : \diamond \\
\\
\text{(PAR)} \\
\hline
\Gamma; \Lambda_{1,2}; \Sigma_{1,2} \vdash P_{1,2} : \diamond \\
\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \vdash P_1 \mid P_2 : \diamond \\
\\
\text{(BRA)} \\
\hline
\Gamma; \Lambda; \Sigma, k : S_i \vdash P_i : \diamond \quad (\forall i \in I) \\
\Gamma; \Lambda; \Sigma, k : \&[l_i : S_i]_{i \in I} \vdash k \triangleright \{l_i : P_i\}_{i \in I} : \diamond \\
\\
\text{(CLOSE)} \\
\hline
\Gamma; \Lambda; \Sigma \vdash P : T \quad k \notin \text{dom}(\Gamma, \Lambda, \Sigma) \\
\Gamma; \Lambda; \Sigma, k : \text{end} \vdash P : T \\
\\
\text{(SEL)} \\
\hline
\Gamma; \Lambda; \Sigma, k : S_j \vdash P : \diamond \quad j \in I \\
\Gamma; \Lambda; \Sigma, k : \oplus[l_i : S_i]_{i \in I} \vdash k \triangleleft l_j.P : \diamond
\end{array}$$

Fig. 10. Linear session typing: processes.

$\Gamma$  is a finite mapping, associating *shared* value types to identifiers.  $\Lambda$  associates variables and *linear* function types.  $\Sigma$  is a finite mapping from variables/session channels to session types.  $\Sigma, \Sigma'$  and  $\Lambda, \Lambda'$  denote disjoint-domain unions.  $\Gamma, u : U$  means  $u \notin \text{dom}(\Gamma)$ , and similarly for the other environments.

*Typing judgement* The typing judgement takes the shape:

$$\Gamma; \Lambda; \Sigma \vdash P : T$$

which is read: under a (global) shared environment  $\Gamma$  and a linear function environment  $\Lambda$ , a term  $P$  has type  $T$  with session usages described by  $\Sigma$ . We say that a judgement is *well-formed* if the environments (pairwise) do not share elements in their domains, that is, when the disjoint union  $\text{dom}(\Gamma) \uplus \text{dom}(\Lambda) \uplus \text{dom}(\Sigma)$  is defined.

To reduce the number of type rules, we make use of the following abbreviation:

$$(\Gamma; \Lambda; \Sigma) \circledast u : T = \begin{cases} \Gamma; \Lambda, u : T; \Sigma & \text{if } T = U \multimap T', \\ \Gamma; \Lambda; \Sigma, u : T & \text{if } T = S, \\ \Gamma, u : T; \Lambda; \Sigma & \text{otherwise.} \end{cases}$$

*Typing rules* The typing rules for identifiers, subtyping, and functions are given in Fig. 9. The rules for processes and sessions are given in Fig. 10. In each rule, we assume that the environments in the consequence are defined.

Starting from Fig. 9, the first group is **(Common)**. First we have a rule for the unit value  $()$ , assigning the type  $\text{unit}$ . In the conclusion, notice that an arbitrary  $\Gamma$  is allowed, but no linear variables ( $\Lambda = \emptyset$ ), or sessions ( $\Sigma = \emptyset$ ). This restriction agrees with the use of weakening only for shared environments, a condition necessary for the preservation of linearity. (Shared) is an introduction rule for identifiers with shared types, i.e., not including  $U \multimap T$  or  $S$ . (LVar) is for linear variables and (Session) is for session endpoints, recording  $x : U \multimap T$  in  $\Lambda$  and  $k : S$  in  $\Sigma$ , respectively. The general strategy is that the environments  $\Lambda$  and  $\Sigma$  record precisely the desired usages of linear variables/sessions, and then within a derivation these usages are combined using disjoint union (to ensure that no copying takes place) and prefixing composition in the case of sessions (to ensure that certain separated usages are seen as one largest use). The use of disjoint union effectively forbids contraction. The absence of weakening guarantees that all linear hypotheses are actually used.

The group **(Structure)** consists of two rules from Linear Logic [21]. The rule (Promotion) ensures that shared functions do not contain linear terms, as unrestricted functions may be used more than once, breaking linearity, or may not be used at all, again violating linearity by making endpoints or linear functions disappear. The rule is a special case of linear promotion [21], since the type  $U \rightarrow T$  is basically  $!(U \multimap T)$ . Dually, (Dereliction) allows to use a shared function in a linear way, which is perfectly safe, and this is convenient when we wish to record, e.g.,  $![U \multimap T].S$  in an environment where the sent function has the unrestricted type  $U \rightarrow T$ . The group **(Subtyping)** consists of one subsumption rule, (Sub), introducing the coinductive subtyping  $\leq_c$  into typing derivations. We write  $\Sigma \leq_c \Sigma'$  when  $\text{dom}(\Sigma) = \text{dom}(\Sigma')$  and for all  $k:S \in \Sigma$ , we have  $k:S' \in \Sigma'$  with  $S \leq_c S'$ . Notice that subsumption can apply to the session environment, but not to other environments, and it can also apply to the given type  $T$  for the term  $P$ .

The second group, **(Function)**, comes from the simply typed linear  $\lambda$ -calculus. In the abstraction rule (Abs), the argument  $x:U$  is from the appropriate environment following the definition of  $\S$ , and it is removed in the conclusion, as expected. (App) is the rule for functional application, and allows the arrow type to be either linear or unrestricted, similarly to [53]; this is needed due to (Rec), since abstractions and variables can always be assigned a linear arrow type, by rules (Abs) and (Dereliction), respectively. The conclusion says that the session environments and linear variable sets of  $P$  and  $Q$  must be disjoint; otherwise, there is copying (more than one usage) of the respective linear terms, which is forbidden. Rule (Rec) is similar to (Abs), but with the addition of a hypothesis for  $x$  in the premise, representing the function itself, and used for typing instances of the function within its body. It is required that the linear function and session environments are empty, since a recursive function may rewrite itself repetitively copying all its contents.

In Fig. 10 we have the final group, **(Process)**, for processes integrated with linear functional and session typing. Rule (Nil) types the empty process. (New) hides a shared name. There is no typing rule for session channels  $(s, \bar{s})$  in initial programs, but in Section 6 we define a rule (New<sub>s</sub>) that verifies the communication patterns for the two endpoints  $s$  and  $\bar{s}$ , in order to ensure compatible dyadic interactions up to asynchronous permutations.

(Conn) and (ConnDual) are for initiating sessions. In the premises of (Conn), the usage  $S$  of the endpoint  $x$  in  $P$  has to agree with the type  $\langle S \rangle$  recorded for the shared identifier  $u$  in the typing environment  $\Gamma$ . Rule (ConnDual) is similar, however the type in the environment  $\Gamma$  is dual to the usage in the session body  $P$ . This is needed in order to indicate which side of the session is followed with respect to a shared channel type, since connecting processes must use their endpoints dually. (Recv) is for receiving values, and uses the notation with  $\S$  to cover the different cases for linear, session, and unrestricted types. The new session type is composed in the conclusion's session environment, in a way that agrees with the protocol, that is, the input is appended before any subsequent actions on  $k$  within  $P$ .

(Send) is the most complex rule, integrating session typing and linear typing. Either  $\Sigma_1$  or  $\Sigma_2$  contains the complete session  $k:S$ , which in practise means that after sending a value, the rest of the session on endpoint  $k$  must appear (and be completed) either in the continuation  $P$  of the sending process, or inside the value  $V$ . In the latter case, we can even have that  $V = k$ , which implements higher-order session passing of  $k$  over  $k$ , i.e., a self-delegation. The composition  $\Sigma_1, \Sigma_2$  is defined in the conclusion, which entails that no endpoint appears in both the remaining sender  $P$  and the sent value  $V$ , because, in that case, we would have a race condition between the receiver of  $V$  and  $P$ , in the usage of communications over these common sessions. The same applies to linear variables free in  $V$  and  $P$ . If  $V$  has a functional type, all session endpoints within it must be complete, that is, suffixed with *end*, because they should not compose further. This is achieved by the necessary use of a suitable instance of (Close). This rule uniformly generalises the corresponding rules in the session types literature [19,48,54,24]. In the conclusion, we delete  $k:S$  where it occurs, either in  $\Sigma_1$  or  $\Sigma_2$ , and the updated type for  $k$  is recorded in the conclusion's session environment, consisting of the continuation type  $S$  prefixed with the output  $![U]$ .

In (Par), we parallel-compose two processes, assuming disjointness of linear function and session environments, as in (App). (Bra) and (Sel) are the standard rules for branching and selection from [24]. In (Bra) all continuations  $P_i$  must have corresponding session usages on  $k$  that agree with the branch type. In (Sel) the continuation  $P$  must have a usage  $S_j$  on  $k$  that agrees with the type corresponding to the selected label  $l_j$  on the selection type of the conclusion.

**Closing sessions** In the above rules for session communication, the premises always contain a hypothesis for the subject of the session action, e.g.,  $k:S$  appears in  $\Sigma_i$  located in the premise of the typing for  $k!\langle V \rangle.P$ . This does not necessarily imply that  $k$  appears in  $P$ , as the usage  $\{k:\text{end}\}$  can be obtained using (Close). This rule is used to effectively close a session on  $k$  by introduction of a hypothesis  $k:\text{end}$ , in order for further composition (i.e., more session actions on  $k$ ) to be rejected.

## 5.2. Examples of typing

Here we state a few examples and counter-examples that demonstrate the purpose of the type system. We revisit some examples from Section 3.2 and from the Introduction.

First, session endpoints must not become “forgotten”:

$$(\lambda(x:S).\mathbf{0}) \cdot s$$

In the above term, after reduction by the (beta) rule, the endpoint  $s$  will not appear any more, and the session on  $\bar{s}$  might become stuck. This term is only typable if  $S = \text{end}$ , otherwise it is not typable because in the premises of rule (Abs) we require a session hypothesis  $x:S$  which cannot be introduced in the typing of  $\mathbf{0}$  except by use of (Closed). Second, session endpoints must not be copied:

$$(\lambda(x:S).(x!\langle V \rangle \mid x!\langle V' \rangle)) \cdot s$$

The above term reduces to:

$$s!\langle V \rangle \mid s!\langle V' \rangle$$

in which we have copied  $s$  breaking the condition of linearity, which is undesirable as the endpoint  $\bar{s}$  will nondeterministically interact with one of the outputs, leaving the other waiting forever. The first term is untypable because typing the function body  $x!\langle V \rangle \mid x!\langle V' \rangle$  with (Par) requires that the sessions in each parallel process are disjoint, which is not the case here due to the common presence of  $x$ . We also revisit the examples in Section 3.2.

1.  $(\lambda(x:U).x \cdot ()) \cdot \ulcorner s!\langle 5 \rangle.0 \urcorner$  is typed with  $U = \text{unit} \multimap \diamond$ , using (App) followed by (Abs) for the left and right subterms of the application, respectively.
2.  $(\lambda(x:U).0) \cdot \ulcorner s!\langle 5 \rangle.0 \urcorner$   
This term is unsafe as the thunk which contains  $s$  does not appear in the function that receives it, after reduction. This is an indirect way for an endpoint to become “forgotten” as before. The typing fails because  $U = \text{unit} \multimap \diamond$  (as above) and (Abs), used for the left subterm of the application, requires  $x:U$  to appear in the linear function environment of the typing of  $0$ , which is impossible.
3.  $(\lambda(x:U').0) \cdot \ulcorner \bar{a}(x).x!\langle 5 \rangle.0 \urcorner$  is typed with  $U' = \text{unit} \rightarrow \diamond$ , applying (App) followed by (Abs) for the two subterms, respectively.

We finally type the optimised higher-order mobility from the Introduction. In the connect process:

$$a(x).x!\langle 5 \rangle.x!\langle \text{true} \rangle.x?(y).(y() \mid R),$$

$a$  has a type  $\langle S_1 \rangle$  where  $S_1 = ![\text{nat}].![\text{bool}].?[U].\text{end}$  and  $U$  is the type of  $y$  (receiving the mobile code  $\ulcorner P \urcorner$ ). This is obtained by applying (Conn), (Send), and (Recv) appropriately. On the other hand, in the optimised session:

$$\bar{a}(x).x!\langle \ulcorner P \urcorner \rangle.x?(z_1).x?(z_2).Q,$$

$x$  is typed with  $S'_2 = ![\text{nat}].?[[\text{nat}].?[[\text{bool}].\text{end}]]$ , applying (ConnDual) with  $a:\langle \bar{S}_2 \rangle$ , then (Send) and (Recv). By an application of (Sub) in the body of the session,  $x$  can also be typed by  $S_2$  (the dual of  $S_1$ ), because  $S'_2 \leq_c S_2$  by Definition 4.3(7). So, the same term can also be assigned  $a:\langle \bar{S}_2 \rangle$  which is the same as  $a:\langle S_1 \rangle$ , and we are done.

## 6. Higher-order linear session typing for runtime processes

### 6.1. Typing system for runtime

The typing system extends the one for programs given previously, replacing a few rules with more general versions. New formulations are needed for the integration of typing at the level of session queues, and for ensuring that the asynchronous calculus is sound.

**Queue types** Due to the presence of labels in session queues, we need to extend the types to facilitate all buffer components, as follows:

$$\tau ::= U \mid l$$

Therefore, every label induces a singleton type identified with the label value.

**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 a 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*  $S - \bar{\tau} = S'$  which subtracts the vector  $\bar{\tau}$  of the queue types of the values stored in a queue from the complete session type  $S$  of the queue, obtaining a remaining session  $S'$ . When the remainder  $S'$  is end, then the session has been completed; otherwise it is not closed yet. The rules are formalised in Fig. 11.

(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  $\bar{\tau}$  of the queue type. (Put) disregards the output action type of the session and calculates the remainder  $S'$  of  $S - \bar{\tau}$ , which is returned prefixed with the original output giving  $![U].\bar{\tau}$ . This is because we are subtracting the input queue types, and therefore the output is not consumed. (Branch) is similar to (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 on the nested types, similarly to (Put), because selection is an output action. An example of the use of session remainders can be found in Section 6.3.

$$\begin{array}{c}
\text{(EMPTY)} \\
\frac{}{S - \epsilon = S} \\
\\
\text{(GET)} \\
\frac{S - \bar{\tau} = S'}{? [U]. S - U \bar{\tau} = S'} \\
\\
\text{(PUT)} \\
\frac{S - \bar{\tau} = S'}{! [U]. S - \bar{\tau} = ! [U]. S'} \\
\\
\text{(BRANCH)} \\
\frac{S_k - \bar{\tau} = S' \quad k \in I}{\& [l_i : S_i]_{i \in I} - l_k \bar{\tau} = S'} \\
\\
\text{(SELECT)} \\
\frac{S_i - \bar{\tau} = S'_i \quad \forall i \in I}{\oplus [l_i : S_i]_{i \in I} - \bar{\tau} = \oplus [l_i : S'_i]_{i \in I}}
\end{array}$$

Fig. 11. Session remainder.

$$\begin{array}{c}
\text{(LABEL)} \\
\frac{}{\Gamma; \emptyset; \emptyset \vdash l : l} \\
\\
\text{(QUEUE)} \\
\frac{\Gamma; \emptyset; \Sigma_i \vdash h_i : \tau_i \quad i \in 1..n \quad \Sigma_0 = \{\bar{s} : \text{end}\}}{\Gamma; \emptyset; (\Sigma_0, \dots, \Sigma_n) \odot s :: \tau_1.. \tau_n \vdash s : h_1..h_n : \diamond} \\
\\
\text{(NEW}_s\text{)} \\
\frac{\Gamma; \emptyset; \Delta, s :: (S_1, \bar{\tau}_1), \bar{s} :: (S_2, \bar{\tau}_2) \vdash P : \diamond \quad S_i - \bar{\tau}_i = S'_i \quad i \in \{1, 2\} \quad S'_1 \leq_c \bar{S}'_2}{\Gamma; \emptyset; \Delta \vdash (\nu s) P : \diamond} \\
\\
\text{(NEW)} \\
\frac{\Gamma, a : (S); \Lambda; \Delta \vdash P : \diamond}{\Gamma; \Lambda; \Delta \vdash (\nu a : (S)) P : \diamond} \\
\\
\text{(PAR)} \\
\frac{\Gamma; \Lambda_{1,2}; \Delta_{1,2} \vdash P_{1,2} : \diamond}{\Gamma; \Lambda_1, \Lambda_2; \Delta_1 \odot \Delta_2 \vdash P_1 \mid P_2 : \diamond}
\end{array}$$

Fig. 12. Runtime typing for asynchronous HO $\pi$ -calculus.

*Typing system for terms with session queues* We first extend the session environment as follows:

$$\Delta ::= \Sigma \mid \Delta, k :: \bar{\tau} \mid \Delta, k :: (S, \bar{\tau})$$

The typing judgement is now of the form:

$$\Gamma; \Lambda; \Delta \vdash l : l \quad \text{and} \quad \Gamma; \Lambda; \Delta \vdash P : T$$

The first judgement is used for typing any labels appearing in a session queue.  $\Delta$  contains usage information for queues in a term  $(s :: \bar{\tau})$ , so that the cumulative result can be compared with the expected session type; for this we use the pairing  $(s :: (S, \bar{\tau}))$  that combines the usage of a channel and the sequence of types already on its queue. Observe that the lighter notation  $(k : \bar{\tau})$  is ambiguous, since  $\bar{\tau}$  can be  $\tau' = S'$ . This is why we use  $(k :: (S, \bar{\tau}))$  and  $(k :: \bar{\tau})$ , respectively.

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

$$\begin{aligned}
\Delta_1 \odot \Delta_2 &= \{s :: (\Delta_i(s), \Delta_j(s)) \mid s : S \in \Delta_i, s :: \bar{\tau} \in \Delta_j\} \\
&\cup \Delta_1 \setminus \text{dom}(\Delta_2) \cup \Delta_2 \setminus \text{dom}(\Delta_1)
\end{aligned}$$

The typing rules for runtime are listed in Fig. 12. (Label) types a label in a queue, while (Queue) forms a sequence corresponding to the types of the values in a queue: we ensure the disjointness of session environments of values, and apply a weakening of ended session types ( $\Sigma_0$ ) for closure under the structure rules. (New<sub>s</sub>) 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  $S_1 - \bar{\tau}_1$  and  $S_2 - \bar{\tau}_2$  are dual session types *up to asynchronous subtyping*; more precisely,  $S'_1$  must be a subtype of the dual of  $S'_2$ , written  $S'_1 \leq_c \bar{S}'_2$ . This is equivalent to  $S'_2 \leq_c \bar{S}'_1$ . Since the session endpoints are compatible, we can restrict  $s$ . The combination of coinductive subtyping with a syntactic duality operator, which is practically the same as the compatibility relation in [20], has two advantages: first, it avoids the need for a separate coinductive duality as in [19]; secondly, as is detailed in [3], a simple syntactic duality does not work with equi-recursive types, and our solution avoids such problems. (Par) composes processes, including queues, and records the session usage by  $\odot$ ; this rule subsumes (Par) for programs. Note that, as this is a runtime typing system, there are no free variables at the top level. Moreover, queues can only appear at the top-level, in parallel to the terms that may appear in initial programs, and never inside functions. Finally, we had to redefine (New) to account for restriction over queues, i.e., with a  $\Delta$ -environment.

## 6.2. Typing the mobile business protocol

We can now type the hotel booking example in Section 2.3, guaranteeing its type safety. Agency has the following types at  $a$  and  $b$ .

$$\begin{aligned}
a : \langle ?[\text{string}].?[\text{date}]. \oplus [\text{move} : S_1 ; \text{cont} : S_1] \rangle, \quad b : \langle S_2 \rangle \\
\text{with } S_1 = ! [S_2].?[\text{double}].\text{end} \quad \text{and } S_2 = \oplus [\text{cont} : S_3 ; \text{move} : ! [\ulcorner \diamond^1 \urcorner].S_3] \\
\text{and } S_3 = ! [\text{string}].?[\text{double}].! [\text{ccard}].\text{end}
\end{aligned}$$

$S_1$  contains higher-order session passing of type  $S_2$ , and the thunk in  $S_2$  has a linear arrow type. Client and Hotel just have the dual of Agency's type at  $a$  and the dual of Agency's type at  $b$ , respectively. Note that in Client, the received session  $y$  appears subsequently in the sent code  $V$ , which is typed by (Send) with the side condition  $k : S_3 \in \Sigma_2$  explained in Section 6.

### 6.3. Typing the optimised mobile business protocol

Now, using also the runtime typing system, we can type the hotel booking example of Section 2.4, in the presence of asynchronous optimisation for higher-order mobility. Agent and standard Client can be typed, by using the rules in Figs. 9 and 10, as follows:

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

We then type MClient and obtain:

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

Applying Definition 4.3 we verify that  $S_{\text{MClient}} \leq_c \overline{S_{\text{Agent}}}$  (and  $S_{\text{MClient}} \leq_c S_{\text{client}}$ ). Then using typing rules (Conn, ConnDual) we can type both MClient and Agent with  $a : \langle S_{\text{Agent}} \rangle \in \Gamma$ , after applying (Sub) on the premises of (ConnDual) typing the body of MClient.

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

$$(\nu s)(\tilde{s} \triangleright \{ \text{move} : \tilde{s}?(code).(\text{run code} \mid \dots), \text{local} : \dots \} \\ \mid s:\text{rtt} \mid \tilde{s}:\text{move} \cdot \ulcorner s!(\text{ritz}) \dots \urcorner)$$

with  $\tilde{s}$  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) from Fig. 12, we type  $\tilde{s}:\text{move} \cdot \ulcorner s!(\text{ritz}) \dots \urcorner$  with session environment

$$\{s : S'_{\text{MClient}}, \tilde{s} :: \text{move} \cdot \text{unit} \multimap \diamond\}$$

where  $S'_{\text{MClient}}$  comes from typing the HO code containing  $s$ , and:

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

and similarly we type  $s:\text{rtt}$  with:  $\{s :: \text{int}\}$ .

The Agent  $\tilde{s} \triangleright \{ \text{move} : \dots, \text{local} : \dots \}$  is typed with (Bra) under session environment:

$$\{\tilde{s} : \&[\text{move} : ?[\text{unit} \multimap \diamond].S'_{\text{Agent}}, \text{local} : S'_{\text{Agent}}]\}$$

The above session environments can be synthesised using  $\odot$  to obtain:

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

Now we use the rules in Fig. 11 to calculate the session remainder of each queue:

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

and we have:

$$![\text{string}].![\text{string}].![\text{ccard}].?[\text{double}].\text{end} \leq_c \overline{S'_{\text{Agent}}}$$

Finally, we can apply (New<sub>s</sub>) and complete the derivation.

## 7. 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 for composing environments of runtime processes. Our definition extends the one in [19] to accommodate for the presence of buffers, using session remainders.

**Definition 7.1** (*Balanced  $\Delta$* ).  $\text{balanced}(\Delta)$  holds if for all

$$\{s :: (S_1, \bar{\tau}_1), \tilde{s} :: (S_2, \bar{\tau}_2)\} \subseteq \Delta$$

with  $S_1 - \bar{\tau}_1 = S'_1$  and  $S_2 - \bar{\tau}_2 = S'_2$ , we have  $S'_1 \leq_c \overline{S'_2}$ .

The definition is based on  $(\text{New}_s)$  in the runtime typing system (Fig. 12): intuitively, all subprocesses generated from an initial typable program should conform to the balanced condition.

Next, we define an ordering between session environments which abstractly represents an interaction at session channels.

**Definition 7.2** ( $\Delta$  ordering). Recall  $\odot$  defined in Section 6. We define  $\Delta \sqsubseteq_s \Delta'$  as follows:

$$\begin{aligned}
 & k : ?[U].S \odot k :: U \bar{\tau} \sqsubseteq_s k : S \odot k :: \bar{\tau} \\
 & k : ![U].S \odot \bar{k} :: \bar{\tau} \sqsubseteq_s k : S \odot \bar{k} :: \bar{\tau} U \\
 & k : \&[l_i : S_i]_{i \in I} \odot k :: l_j \bar{\tau} \sqsubseteq_s k : S_j \odot k :: \bar{\tau} \quad j \in I \\
 & k : \oplus[l_i : S_i]_{i \in I} \odot \bar{k} :: \bar{\tau} \sqsubseteq_s k : S_j \odot \bar{k} :: \bar{\tau} l_j \quad j \in I \\
 & \frac{k : \text{unfold}^n(S) \odot \Delta \sqsubseteq_s \Delta'}{k : S \odot \Delta \sqsubseteq_s \Delta'} \quad \frac{\Delta_1 \sqsubseteq_s \Delta_2 \text{ and } \Delta \odot \Delta_1 \text{ defined}}{\Delta \odot \Delta_1 \sqsubseteq_s \Delta \odot \Delta_2} \\
 & \frac{k : S_j \odot \bar{k} :: \bar{\tau} \sqsubseteq_s k : S'_j \odot \bar{k} :: \bar{\tau}' \quad \text{for all } j \in H}{k : \mathcal{A}(S_h)^{h \in H} \odot \bar{k} :: \bar{\tau} \sqsubseteq_s k : \mathcal{A}(S'_h)^{h \in H} \odot \bar{k} :: \bar{\tau}'}
 \end{aligned}$$

The first four axioms capture the transfer of types (corresponding to values) between programs and queues. For example the first axiom captures how an input session against a non-empty queue will evolve by removing the prefix and head element, respectively. The output axioms can be understood by duality. Then we have rules that introduce  $n$ -times unfolding (this is needed due to asynchrony) and arbitrary contexts ( $\Delta$ ) which simplify the other rules. In the last rule, which allows to deal with asynchronous subtypes, there are two notable points. First, we are only interested in output actions, and this is why we use the queue  $\bar{k}$ . Second, note that the queue type  $\bar{k} :: \bar{\tau}$  is the same for all premises ( $j \in H$ ), since we are performing a common asynchronous action. In fact,  $\bar{\tau}'$  will be equal to  $\bar{\tau}h$  where  $h$  is a label or value type; this is evident from the output axioms. 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  $\text{balanced}(\Delta)$  and  $\Delta \sqsubseteq_s \Delta'$  then  $\text{balanced}(\Delta')$ . Then we have:

**Theorem 7.3** (Type soundness).

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

The proofs can be found in Appendix C. We make use of a number of supporting lemmas; the actual proof of Type Soundness begins on page 34.

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

**Definition 7.4** (Error process). We say  $P$  is an error if  $P \equiv (\vec{v}\vec{a})(\vec{v}\vec{s})(Q \mid R)$  where  $Q$  is one of the following: (a) a  $\mid$ -composition of two  $k$ -processes or of a  $k$ -process and a  $\bar{k}$ -process, that does not form a  $k$ -redex or a  $k$ -input with an empty  $k$ -buffer; (b) a  $k$ -redex consisting a  $k$ -input and  $k$ -buffer such that  $Q = k?(x).Q' \mid k:l_k \bar{h}$ , or  $Q = k \triangleright \{l_i : P_i\}_{i \in I} \mid k:Vh$ , or  $Q = k \triangleright \{l_i : P_i\}_{i \in I} \mid k:l_k \bar{h}$  with  $k \notin I$ ; (c) a  $k$ -process with  $k$  or  $\bar{k}$  in  $\vec{s}$  but with  $\bar{k}$  not free in  $R$  or  $Q$ ; (d) a prefixed process or application containing a  $k$ -buffer.

The above says that a process is an error if (a) it breaks the linearity of  $k$  by having e.g. two  $k$ -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 7.3, we achieve the following general communication-safety theorem:

**Theorem 7.5** (Communication safety). If  $\Gamma; \emptyset; \Delta \vdash P : \diamond$  with  $\text{balanced}(\Delta)$ , then  $P$  never reduces into an error.

**Proof.** It is enough to consider a one step reduction from a well-typed term. From Theorem 7.3 we know that the result is well-typed. Therefore it suffices to prove that a well-typed term cannot be an error. We consider the given cases. For (a), we may have a composition of two  $k$ -processes such as, e.g.,  $k : \bar{h}_1 \mid k : \bar{h}_2$  or  $k?(x_1).P_1 \mid k?(x_2).P_2$ . It is clear that no such combination is typable: we cannot compose by  $\odot$  any environments  $\Delta_1$  and  $\Delta_2$  with  $k$  on both, unless if one is a queue typing  $k :: \bar{\tau}$  and the other is a session typing  $k : S$ . For (b), a communication mismatch is untypable since the session



remainder will be undefined, e.g.,  $?[U].S - l_k \bar{c}$  is not defined, and similarly for  $\&[l_i : S_i]_{i \in I} - l_k \bar{c}$  when  $k \notin I$ . When the remainder is undefined, the rule (New<sub>s</sub>) cannot apply, and therefore the term is untypable. For (c), a missing occurrence of the dual buffer is excluded by (New<sub>s</sub>). In particular, even if a session on  $\bar{k}$  is ended and so does not occur in communications, the buffer on  $\bar{k}$  will still exist under the scope of  $k/\bar{k}$ . For (d), a buffer cannot occur in the body of an abstraction or under an input prefix or a branching, as can be seen by the use of  $\Sigma$ -environments in the user-level typing rules in Figs. 9 and 10.  $\square$

**Corollary 7.6** (*Open communication safety*). *If  $\Gamma; \Delta; \Delta \vdash P : \diamond$  with  $\text{balanced}(\Delta)$ , then  $P$  never reduces into an error.*

**Proof.** This follows easily from Theorem 7.5, since we can close the linear interface with abstractions and then apply to linear function arguments, obtaining a term of the same type for which safety holds. In particular,  $P\sigma$ , with  $\sigma$  a closing substitution for  $\Delta$ , will never reduce to an error, so the same is easily shown to hold for  $P$ .  $\square$

## 8. Related work

There is a large literature on linear and session types for both the  $\lambda$ -calculus and the  $\pi$ -calculus. Below we give the most closely related work, dividing into three parts: one focuses on the linear typing system of the  $\lambda$ -calculus and the session types for functional programming languages, the next focuses on asynchronous subtyping systems, and finally the last explains the relationship between linearity and asynchrony from the aspect of proof theory, following recent developments. See also [16] for discussions on other type disciplines of the  $\pi$ -calculus as well as on applications of session types.

### 8.1. Linear and session typing systems for higher-order functions

Our typing system is substructural in the sense that for session environments  $\Sigma$  we do not allow weakening and contraction, ensuring that a session channel is recorded as having been used only when it actually occurs in session communication expressions. Similarly no structural transformations can apply to linear variable environments, ensuring that the occurrence of a variable manifests that it has indeed been used exactly once. The ways in which our typing system enforces linearity can be seen as an amalgamation of the two approaches in [53], retaining the simplicity of declarative systems, and the decidability of algorithmic ones. Walker's work [53] provides a good exposition to substructural typing (in which linear and affine usages can be seen as special cases). Note that in our system there is no need to enforce linear usage for other than functional types. Applying the inference techniques of [17,15] and [51], with the algorithmic subtyping of [19], it should be possible to construct a type inference system.

Session types in functional languages have been studied in various works. In the first study [52], the authors define a concurrent multi-threaded functional language with sessions. Their language supports sending of channels and higher-order values, branching and selection, recursive sessions and channel sharing. It has an explicit multi-threading primitive (`fork`) and explicit stores. The paper [20] extends the previous language to a variant of sessions where message sending is non-blocking. This is handled by explicitly storing an entry for the two endpoint channels in a buffer. Its functionality is the same as our use of two session channels for distinguishing the two endpoints (similarly to [19]). They simplify their previous type judgement which required input and output environments [52] by integrating linear typing with a `split` operator, which is more directly related to the original non-deterministic typing of [53]. While a precise typability comparison is difficult due to our additional primitives, their work also shows a use of linear types for functional languages with sessions.

One of the active areas in the functional setting is the integration of session types into the lazy functional language Haskell [39,45,27]. Incorporating primitives for session interaction into Haskell requires to define an appropriate IO-monad, which is also suitable for solving aliasing problems. Instead of extending the type system of an existing language and adding linear types like our work and [52,20,6], the work [39,45] encode sessions using the features of Haskell's type system. In general, the encoding approach in [39,45] generates more cumbersome types, but can take advantage of Haskell's type inference (them in most cases). The work [27] establishes a more advanced session type inference technique. An ML-style polymorphism based on [20] is also investigated in [6].

Also, the work [10] uses (the synchronous part of) our typing system (published in [35]) to encode session types in linear behavioural types in the  $\text{HO}\pi$ -calculus. This demonstrates that the substructural features of our typing system makes it easy to translate session types to other structures, mechanically.

Finally, the work [49] presents an alternative session system for higher-order processes, based on a logical interpretation integrated with a functional language. This system enjoys stronger properties in terms of progress, so that processes do not get deadlocks, but is slightly more complex due to the heavy use of monads. Moreover, it does not make use of any form of asynchronous subtyping.

### 8.2. Asynchronous session typing and subtyping

Asynchronous subtyping was first studied in [37] for multiparty session types [25]; however, 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 [26] and web service protocols [23], to which we can now apply our theory for flexible optimisation. The proof of the transitivity in this work requires a more complex construction of the transitive closure  $\text{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$  (Definition B.10) than the one in [37] due to the higher-order constructs. In spite of the richness of the type structures, we proposed a more compact runtime typing and proved communication safety in the presence of higher-order code, which is not presented in [37]. Moreover, our new typing system extends naturally the synchronous account of the linear typing system published in [35], demonstrating a smooth integration of two kinds of type-directed optimisation.

The coinductive subtyping of recursive session types was first studied in [19], adapting standard methods from IO-subtyping in the  $\pi$ -calculus [44]. The subtyping system of [19] does not provide any form of asynchronous permutation, thus does not need the nested  $n$ -times unfolding (Definition 4.1). Moreover, our transitivity proof is significantly more involved than in [19] due to the incorporation with  $n$ -time unfolding, permutation, 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 (e.g. [25,4]) 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, and also more flexible, as it naturally extends to asynchronous contexts. Our queue typing is more similar to that in [20], where smaller types are obtained after *matching* with buffer values. However, our method works with queue types rather than with values directly, which allowed it to be extended smoothly to handle asynchronous optimisation, which is not treated in [20]. For example, we allow a type consisting of an output followed by an input action to be reduced with a type corresponding to the input, leaving the output prefix intact. Moreover, using a more delicate composition between values and queue typing, our system enables linear mobile code to be stored in the queues.

An analysis of asynchronous session action permutations, encompassing an asynchronous “acceptance” relation which accommodates for output actions performed in advance, appears in an unpublished manuscript [40]. The authors suggest that their algorithm is terminating. However, if their system admits  $\mu t.[U_1].t$  as a subtype of  $\mu t.[U_1].?[U_2].t$ , which as we show on page 13 induces an infinite simulation, then it is unclear how it avoids divergence without any special provision.

Finally, a notion of asynchronous context and a definition of asynchronous duality that resembles our subtyping (combined with duality) appears in [5]. However, this notion is only developed in order to prove type soundness and it is not integrated with the typing system which was mentioned as an interesting future work. It is developed for finite sessions that, additionally, do not support delegation (name passing). Our work develops such a subtyping for a much more expressive calculus supporting name and code mobility, and also in the context of recursive session types. These features require co-inductive methods that really bring to the surface a number of challenges such as those arising from infinite simulations.

The recent work [9] studies a notion of *preciseness* in session subtyping, including an adaptation of our notion of asynchronous subtyping. As we mentioned in Section 4.2, the subtyping of [9] avoids so called “orphan” messages, i.e., those that are never received from a queue, by restricting the subtyping relation to contain a finite amount of branchings (in our case this would also include inputs) before an output can be fetched from inside an asynchronous context. In simple terms, they do not allow the accumulation of messages which follows from missing inputs. We believe a practical application of asynchronous subtyping will make use of both approaches, ours and that of [9]: for many kinds of values that do not require linear constraints, messages can safely be left on a queue and later garbage collected; for messages containing linear values, a restriction might be needed such as the one in [9] or the one in [32, p. 181] or a buffer bound as in [20].

As a general remark, note that our choice to use  $\mu$ -types instead of (infinite) regular trees serves better our aim of informing programming technology, and given the restrictions, notably that of contractiveness, the two notions are equivalent [18,2]. Actually, the coinductive treatment would not differ much, except in notational aspects, since we would have to fetch output actions from deep elements of the tree representation as we do with asynchronous contexts.

### 8.3. Linearity and asynchrony from the proof theoretical perspective

A typical use of linearity in processes is to simply require that linear channels are used exactly once, which differs than sessions-based linearity where channels are used once “at any moment” and can be reused in order to complete a protocol. In that sense, linearity in sessions is about avoiding race conditions on channels, but the two notions can be interchanged as seen by recent works [22].

There is, however, a deeper notion of linearity that arises from propositions-as-types interpretations, starting from [1]. Recently, the work [7] gave the first such correspondence for sessions types, matching session typed processes to Intuitionistic Linear Logic proofs. This kind of interpretation becomes more relevant for asynchrony once the constraints of sequentiality (arising from sequent proofs) are relaxed, as has been done in [14]<sup>2</sup> where logical sessions are obtained for asynchronous  $\pi$ -calculus, and even more in [33] where logical sessions based on Proof Nets are obtained for a Solos [29] calculus. Indeed, once we eliminate many of the prefixes, the need to perform asynchronous subtyping may seem redundant, however this is not the case: in distributed computing communications are implemented using sockets or channels of some form, so our buffered model is in fact more realistic. In the case of [14], our subtyping would allow output actions

<sup>2</sup> Note that, as explained in [33], this work does not enjoy Subject Reduction, but this can be fixed easily.

Term	fv	fn
$x$	$\{x\}$	$\emptyset$
$a$	$\emptyset$	$\{a\}$
$l / () / \mathbf{0}$	$\emptyset$	$\emptyset$
$s$	$\emptyset$	$\{s\}$
$\bar{s}$	$\emptyset$	$\{\bar{s}\}$
$\lambda x. P$	$\text{fv}(P) \setminus \{x\}$	$\text{fn}(P)$
$\mu x. \lambda y. P$	$\text{fv}(P) \setminus \{x, y\}$	$\text{fn}(P)$
$u(x). P / \bar{u}(x). P$	$\text{fv}(u) \cup (\text{fv}(P) \setminus \{x\})$	$\text{fn}(u) \cup \text{fn}(P)$
$k?(x). P$	$\text{fv}(k) \cup (\text{fv}(P) \setminus \{x\})$	$\text{fn}(k) \cup \text{fn}(P)$
$k!(V). P$	$\text{fv}(k) \cup \text{fv}(V) \cup \text{fv}(P)$	$\text{fn}(k) \cup \text{fn}(V) \cup \text{fn}(P)$
$k \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$	$\text{fv}(k) \cup \text{fv}(P_1) \cup \dots \cup \text{fv}(P_n)$	$\text{fn}(k) \cup \text{fn}(P_1) \cup \dots \cup \text{fn}(P_n)$
$k \triangleleft l. P$	$\text{fv}(k) \cup \text{fv}(P)$	$\text{fn}(k) \cup \text{fn}(P)$
$P \mid Q / P Q$	$\text{fv}(P) \cup \text{fv}(Q)$	$\text{fn}(P) \cup \text{fn}(Q)$
$(\nu a : \langle S \rangle) P$	$\text{fv}(P)$	$\text{fn}(P) \setminus \{a\}$
$(\nu s) P$	$\text{fv}(P)$	$\text{fn}(P) \setminus \{s, \bar{s}\}$
$k:\bar{h}$	$\emptyset$	$\text{fn}(k) \cup \text{fn}(\bar{h})$

Fig. A.13. Free variables and free names.

hidden under an input prefix to be extracted, which corresponds to valid transformations in Linear proofs. For example, we would allow  $B \otimes (A \multimap C)$  to be a subtype of  $A \multimap (B \otimes C)$ , under certain conditions.

## 9. Conclusion

We formalise for the first time session typing for a process language that allows not only data but also runnable code to be the subject of structured type-safe communications. The ability to exchange code is fundamental in concurrent and distributed systems where programs cannot be fully fixed ab initio and dynamicity is a prerequisite. We then relax the strict compatibility requirements that govern pairs of interacting processes to allow certain classes of message-passing actions to be permuted, offering not only greater flexibility in composing programs, but also guidance toward type-safe optimisations. Our session typing system for the  $\text{HO}\pi$ -calculus can serve as a theoretical foundation for process and functional languages, and our asynchronous subtyping has been already implemented in order to allow *message overlapping* in message passing parallel algorithms in a C language with sessions [41]. The first author's thesis also demonstrates that the theory developed in this article is smoothly extensible to object-orientation [32]. Our future work includes a type-preserving fully abstract encoding of  $\text{HOS}\pi$  into the session  $\pi$ -calculus based on a session-based asynchronous bisimulation [28] or behavioural equivalences [42]; a development of a decidable algorithm for the asynchronous subtyping relation along the lines of [19]; extensions to multiparty session types [4,25]; and an incorporation with actor-based languages for concurrency, following the Erlang-based development in [34]. In summary, an automatic optimisation that preserves the intended semantics and does not violate type-safety is interesting, both theoretically and practically, and in this work we have established a solid theory to support this development.

## Acknowledgments

We thank the reviewers for their comments. The work is partially supported by EP/K034413/1, EP/K011715/1 and EP/L00058X/1, EU project FP7-612985 UpScale, and ICT COST Action IC1201 BETTY.

## Appendix A. Syntax

In Fig. A.13, we list the sets of free names and variables of  $\text{HOS}\pi$ .

## Appendix B. Proofs on asynchronous subtyping

This appendix gives the proofs on the properties of  $\leq_c$  (Theorem 4.4). The outline is given in the last paragraph of Section 4.3.

**Lemma B.1.** *If  $S_1 \leq_c S_2$  then  $\text{unfold}^n(S_1) \leq_c S_2$ .*

**Proof.** Let  $\mathfrak{R}$  be a type simulation such that  $S_1 \mathfrak{R} S_2$ . Let

$$\mathcal{U}_1^n(\mathfrak{R}) = \bigcup_{i \in 1..n} \left\{ (\text{unfold}^i(S'_1), S'_2) \mid (S'_1, S'_2) \in \mathfrak{R} \right\} \cup \mathfrak{R}$$

Clearly  $(\text{unfold}^n(S_1), S_2) \in \mathcal{U}_1^n(\mathfrak{R})$ , but it has to be shown that  $\mathcal{U}_1^n(\mathfrak{R})$  is a type simulation. For this we need to demonstrate that for any  $(T_1, T_2) \in \mathcal{U}_1^n(\mathfrak{R})$  the rules of simulation (Definition 4.3) hold. Since  $\mathfrak{R} \subseteq \mathcal{U}_1^n(\mathfrak{R})$  and  $\mathfrak{R}$  is a simulation, we only

need to examine the cases for  $(\text{unfold}^i(S'_1), S'_2) \in \mathcal{U}_1^n(\mathfrak{N}) \setminus \mathfrak{N}$ , that is, for the new elements for which  $(S'_1, S'_2) \in \mathfrak{N}$  holds by our construction of  $\mathcal{U}_1^n(\mathfrak{N})$ .

In the following we write  $\leq_c^{(11)}$  to mean case (11) of Definition 4.3.

**Case**  $\text{unfold}^i(S'_1) = \text{end}$ . Then  $S'_1 = \mu \mathbf{t}_1 \dots \mu \mathbf{t}_z.\text{end}$  for  $0 \leq z \leq i$ . We have, by assumption, that  $(S'_1, S'_2) \in \mathfrak{N}$ , therefore after applying  $\leq_c^{(11)}$   $z$  times we get  $(\text{end}, S'_2) \in \mathfrak{N}$ , and by the rules of simulation  $\text{unfold}^m(S'_2) = \text{end}$ , for some  $m$ , as required.

**Case**  $\text{unfold}^i(S'_1) = ![U_1].S_{10}$ . W.l.o.g let  $S'_1 = \mu \mathbf{t}_1 \dots \mu \mathbf{t}_z.[U_1].S'_{10}$  with  $0 \leq z \leq i$ . We have  $(S'_1, S'_2) \in \mathfrak{N}$  and after  $z$  uses of  $\leq_c^{(11)}$  we obtain  $(\text{unfold}^z(S'_1), S'_2) \in \mathfrak{N}$  which can be written, based on the shape of  $S'_1$ , as  $(![U_1].S'_{10}, S'_2) \in \mathfrak{N}$ . The type  $S'_{10}$  is derived from  $S'_{10}$  after the type variable substitutions induced by the  $z$  unfoldings on  $S'_1$ . By the rules of simulation, from  $(![U_1].S'_{10}, S'_2) \in \mathfrak{N}$  we obtain  $\text{unfold}^m(S'_2) = \mathcal{A}([U_2].S_{2h})^{h \in H}$  and  $(U_2, U_1) \in \mathfrak{N}$  and  $(S'_{10}, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{N}$ . From the shape of  $S'_1$  given previously we have that  $\text{unfold}^i(S'_1) = ![U_1].\text{unfold}^{i-z}(S'_{10})$  and by our assumptions  $S_{10} = \text{unfold}^{i-z}(S'_{10})$ . By the construction of  $\mathcal{U}_1^n(\mathfrak{N})$  and  $(S'_{10}, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{N}$  we have that  $(\text{unfold}^{i-z}(S'_{10}), \mathcal{A}(S_{2h})^{h \in H}) \in \mathcal{U}_1^n(\mathfrak{N})$ . Therefore we have  $(S_{10}, \mathcal{A}(S_{2h})^{h \in H}) \in \mathcal{U}_1^n(\mathfrak{N})$ . From the definition of  $\mathcal{U}_1^n(\mathfrak{N})$  which includes  $\mathfrak{N}$  the above provide us with  $(U_2, U_1) \in \mathcal{U}_1^n(\mathfrak{N})$  and  $(S_{10}, \mathcal{A}(S_{2h})^{h \in H}) \in \mathcal{U}_1^n(\mathfrak{N})$ , as required.

**Case**  $\text{unfold}^i(S'_1) = \mu \mathbf{t}_1 \dots \mu \mathbf{t}_z.S_{10}$ . Then without loss of generality  $S'_1 = \mu \mathbf{t}'_1 \dots \mu \mathbf{t}'_i.\mu \mathbf{t}_1 \dots \mu \mathbf{t}_z.S'_{10}$ . The type  $S_{10}$  is derived from  $S'_{10}$  after the type variable substitutions induced by the  $i$  unfoldings on  $S'_1$ . Since  $(S'_1, S'_2) \in \mathfrak{N}$ , after  $i$  applications of  $\leq_c^{(11)}$  we obtain  $(\text{unfold}^i(S'_1), S'_2) \in \mathfrak{N}$  and hence  $(\text{unfold}^1(\text{unfold}^i(S'_1)), S'_2) \in \mathfrak{N}$  which is the required result since  $\mathfrak{N} \subseteq \mathcal{U}_1^n(\mathfrak{N})$ .

Other cases are similar.  $\square$

**Lemma B.2.** If  $S_1 \leq_c S_2$  then  $S_1 \leq_c \text{unfold}^n(S_2)$ .

**Proof.** Let  $\mathfrak{N}$  be a type simulation such that  $S_1 \mathfrak{N} S_2$ . Let

$$\mathcal{U}_1^n(\mathfrak{N}) = \bigcup_{i \in 1..n} \left\{ (S'_1, \text{unfold}^i(S'_2)) \mid (S'_1, S'_2) \in \mathfrak{N} \right\} \cup \mathfrak{N}$$

The proof follows a pattern similar to the previous lemma. Clearly we have  $(S_1, \text{unfold}^n(S_2)) \in \mathcal{U}_1^n(\mathfrak{N})$ , but it has to be shown that  $\mathcal{U}_1^n(\mathfrak{N})$  is a type simulation. For this we need to demonstrate that for any  $(T_1, T_2) \in \mathcal{U}_1^n(\mathfrak{N})$  the rules of simulation (Definition 4.3) hold. Since  $\mathfrak{N} \subseteq \mathcal{U}_1^n(\mathfrak{N})$  and  $\mathfrak{N}$  is a simulation, we only need to examine the cases for  $(S_1, \text{unfold}^m(S_2)) \in \mathcal{U}_1^n(\mathfrak{N}) \setminus \mathfrak{N}$  with  $m \leq n$ , that is, for the new elements for which  $(S_1, S_2) \in \mathfrak{N}$  holds by the construction of  $\mathcal{U}_1^n(\mathfrak{N})$ .

Interesting cases are:

**Case**  $S_1 = \text{end}$ . Then  $(S_1, S_{20}) \in \mathfrak{N}$  and  $S_2 = \text{unfold}^m(S_{20})$  and  $\text{unfold}^z(S_{20}) = \text{end}$ . If  $z \leq m$  then  $\text{unfold}^m(S_{20}) = \text{end}$  as required. If  $z > m$  then  $\text{unfold}^{z-m}(S_2) = \text{end}$  as required.

**Case**  $S_1 = ![U_1].S'_1$ . Then  $(S_1, S_{20}) \in \mathfrak{N}$  and  $S_2 = \text{unfold}^m(S_{20})$  and  $\text{unfold}^z(S_{20}) = \mathcal{A}([U_2].S_{2h})^{h \in H}$  and  $(U_2, U_1) \in \mathfrak{N}$  and  $(S'_1, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{N}$ .

If  $z \leq m$  then, using the definition of unfold

$$S_2 = \text{unfold}^{m-z}(\mathcal{A}([U_2].S_{2h})^{h \in H}) = \mathcal{A}([U_2].\text{unfold}^{m-z}(S_{2h}))^{h \in H}$$

We have  $(U_2, U_1) \in \mathfrak{N}$ , then we need  $(S'_1, \mathcal{A}(\text{unfold}^{m-z}(S_{2h}))^{h \in H}) \in \mathcal{U}_1^n(\mathfrak{N})$ . From  $(S'_1, \mathcal{A}(S_{2h})^{h \in H}) \in \mathfrak{N}$  we obtain  $(S'_1, \text{unfold}^{m-z}(\mathcal{A}(S_{2h})^{h \in H})) \in \mathcal{U}_1^n(\mathfrak{N})$ , and then from the definition of unfold we obtain  $\text{unfold}^{m-z}(\mathcal{A}(S_{2h})^{h \in H}) = \mathcal{A}(\text{unfold}^{m-z}(S_{2h}))^{h \in H}$ . If  $z > m$  then  $\text{unfold}^{z-m}(S_2) = \mathcal{A}([U_2].S_{2h})^{h \in H}$  and the supporting elements are in  $\mathfrak{N}$ , as required.

Other cases are similar.  $\square$

**Definition B.3** (Unfolding extension). Given a simulation  $\mathfrak{N}$ , the *unfolding extension* of  $\mathfrak{N}$  is defined as follows:

$$\mathcal{U}^n(\mathfrak{N}) = \mathcal{U}_1^n(\mathfrak{N}) \cup \mathcal{U}_2^n(\mathfrak{N})$$

**Proposition B.1.** If  $\mathfrak{N} \leq_c$  then  $\mathcal{U}^n(\mathfrak{N}) \leq_c$ . That is, for any simulation  $\mathfrak{N}$ , the unfolding extension  $\mathcal{U}^n(\mathfrak{N})$  is a type simulation.

**Proof.** Trivial as  $\mathcal{U}^n(\mathfrak{R})$  is defined as the union of two simulations.  $\square$

We now define the single-step permutation transformations for top-level actions, which enable us to obtain more asynchronous subtypes, as this is needed further on when, given a simulation, we obtain more asynchronous simulations utilising single and multi-step permutations. There are two components, permutation contexts  $\mathcal{C}$  and permutation rules  $\gg$ , defined as follows:

**Definition B.4** (Single-step permutation).

Permutation Contexts

$$\mathcal{C} ::= ?[U].\langle \cdot \rangle^{h \in H} \mid \&[l_i : \langle \cdot \rangle^{h \in H}]_{i \in I}$$

Permutation Rules

$$S \gg S$$

$$\mathcal{C}(![U].S_h)^{h \in H} \gg ![U].\mathcal{C}\langle S_h \rangle^{h \in H}$$

$$\mathcal{C}\langle \oplus[l_i : S_{ih}]_{i \in I_h} \rangle^{h \in H} \gg \oplus[l_i : \mathcal{C}\langle S_{ih} \rangle^{h \in H}]_{i \in I} \quad \forall h \in H . I \subseteq I_h$$

**Definition B.5** (Contextual extension). Given a simulation  $\mathfrak{R}$ , the contextual extension of  $\mathfrak{R}$  is defined as follows:

$$\begin{aligned} \mathcal{CE}(\mathfrak{R}) = & \{ (?[U_1].S_1, ?[U_2].S_2) \mid (U_1, U_2) \in \mathfrak{R} \wedge (S_1, S_2) \in \mathfrak{R} \} \\ & \cup \{ (\&[l_i : S_{1i}]_{i \in I}, \&[l_j : S_{2j}]_{j \in J}) \mid J \subseteq I \wedge \forall j \in J . (S_{1j}, S_{2j}) \in \mathfrak{R} \} \\ & \cup \mathfrak{R} \end{aligned}$$

**Lemma B.6.** If  $\mathfrak{R} \subseteq \leq_c$  then  $\mathcal{CE}(\mathfrak{R}) \subseteq \leq_c$ . That is, for any simulation  $\mathfrak{R}$ , the contextual extension  $\mathcal{CE}(\mathfrak{R})$  is a type simulation.

**Proof.** Trivial since the generated pairs in  $\mathcal{CE}(\mathfrak{R})$  are exactly those justified by the conditions in Definition 4.3, cases (8) and (10), with the required assumptions provided in  $\mathfrak{R}$ . We do not need to examine the  $\mathfrak{R}$  subcomponent as it is a simulation by assumption.  $\square$

Next we define the asynchronous extension of a simulation, with degree  $n$ . The degree represents the number of single-step permutations, applied successively to all the components of the given simulation, up to asynchronous contexts  $\mathcal{A}$ .

**Definition B.7** (Asynchronous extension). Given a simulation  $\mathfrak{R}$ , the asynchronous extension of  $\mathfrak{R}$  with degree  $n$  is defined as follows:

$$\begin{aligned} \alpha^0(\mathfrak{R}) &= \mathfrak{R} \\ \alpha^n(\mathfrak{R}) &= \mathcal{CE}(\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))) \\ &= \cup \{ (\mathcal{A}\langle S'_{1h} \rangle^{h \in H}, S_2) \mid (\mathcal{A}\langle S_{1h} \rangle^{h \in H}, S_2) \in \alpha^{n-1}(\mathfrak{R}) \\ &\quad \wedge \forall h \in H . S_{1h} \gg S'_{1h} \} \quad (n \geq 1) \end{aligned}$$

The notation  $\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$  stands for the union of all  $\mathcal{U}^m(\alpha^{n-1}(\mathfrak{R}))$  such that  $m \in \mathbb{N}$ .

**Lemma B.8.** If  $\mathfrak{R} \subseteq \leq_c$  then  $\alpha^n(\mathfrak{R}) \subseteq \leq_c$ . That is, for any simulation  $\mathfrak{R}$  and degree  $n \in \mathbb{N}$ , the asynchronous extension  $\alpha^n(\mathfrak{R})$  is a type simulation.

**Proof.** We proceed by induction on the degree  $n$ . The base case of  $n = 0$  holds because  $\mathfrak{R}$  is a simulation by assumption. We then prove the inductive case for any  $n \geq 1$ .

By the inductive hypothesis  $\alpha^{n-1}(\mathfrak{R}) \subseteq \leq_c$ , then by Proposition B.1 we have  $\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R})) \subseteq \leq_c$ , and by Lemma B.6 we obtain  $\mathcal{CE}(\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))) \subseteq \leq_c$ . Therefore, it is not necessary to examine pairs in this subset of  $\alpha^n(\mathfrak{R})$ . Then, it remains to examine an arbitrary pair  $(\mathcal{A}\langle S'_{1k} \rangle^{k \in K}, S_2) \in \alpha^n(\mathfrak{R})$  such that  $(\mathcal{A}\langle S_{1k} \rangle^{k \in K}, S_2) \in \alpha^{n-1}(\mathfrak{R})$  with  $\forall k \in K . S_{1k} \gg S'_{1k}$ . We proceed by taking cases on the shape of the context  $\mathcal{A}$ .

**Case**  $\mathcal{A} = \langle \cdot \rangle^{k \in K}$ . Then let  $S_1 = \mathcal{A}\langle S_{1k} \rangle^{k \in K}$ , and  $S'_1 = \mathcal{A}\langle S'_{1k} \rangle^{k \in K}$ . We have  $S_1 \gg S'_1$ , and proceed by examination of the permutation applied.

**Subcase**  $S_1 = S'_1$ . Trivial.

**Subcase**  $S_1 = \mathcal{C}(![U].S_{1k})^{k \in K}$  and  $S'_1 = ![U].\mathcal{C}(S_{1k})^{k \in K}$ . We proceed with cases on  $\mathcal{C}$ .

If  $\mathcal{C} = ?[U_1].\langle \cdot \rangle^{k \in K}$ , then

$$S_1 = \mathcal{C}(![U].S_{1k})^{k \in K} = ?[U_1].![U].S_{1k}$$

and

$$\begin{aligned} S'_1 &= ![U].\mathcal{C}(S_{1k})^{k \in K} = ![U].?[U_1].S_{1k} \\ (S_1, S_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^n(S_2) = ?[U_2].S'_2 \\ &\quad \wedge (U_1, U_2) \in \alpha^{n-1}(\mathfrak{R}) \\ &\quad \wedge (![U].S_{1k}, S'_2) \in \alpha^{n-1}(\mathfrak{R}) \\ ([U].S_{1k}, S'_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^m(S'_2) = \mathcal{A}_1(![U'].S_{2h})^{h \in H} \\ &\quad \wedge (U', U) \in \alpha^{n-1}(\mathfrak{R}) \\ &\quad \wedge (S_{1k}, \mathcal{A}_1(S_{2h})^{h \in H}) \in \alpha^{n-1}(\mathfrak{R}) \end{aligned}$$

From the definition of  $n$ -times unfolding we obtain

$$\text{unfold}^{n+m}(S_2) = ?[U_2].\mathcal{A}_1(![U'].S_{2h})^{h \in H} = \mathcal{A}_2(![U'].S_{2h})^{h \in H}$$

with

$$\mathcal{A}_2 = ?[U_2].\mathcal{A}_1$$

Now we proceed to justify the inclusion  $(S'_1, S_2) \in \alpha^n(\mathfrak{R})$ . Then we have  $(![U].?[U_1].S_{1k}, S_2) \in \alpha^n(\mathfrak{R})$ . Also  $\text{unfold}^{n+m}(S_2) = \mathcal{A}_2(![U'].S_{2h})^{h \in H}$  with  $(U', U) \in \alpha^{n-1}(\mathfrak{R}) (\subseteq \alpha^n(\mathfrak{R}))$ . We need to show that  $(?[U_1].S_{1k}, \mathcal{A}_2(S_{2h})^{h \in H}) \in \alpha^n(\mathfrak{R})$  which can be written  $(?[U_1].S_{1k}, ?[U_2].\mathcal{A}_1(S_{2h})^{h \in H}) \in \alpha^n(\mathfrak{R})$ . We have  $(U_1, U_2) \in \alpha^{n-1}(\mathfrak{R})$  and  $(S_{1k}, \mathcal{A}_1(S_{2h})^{h \in H}) \in \alpha^{n-1}(\mathfrak{R})$ , hence we have  $(?[U_1].S_{1k}, \mathcal{A}_2(S_{2h})^{h \in H}) \in \mathcal{CE}(\alpha^{n-1}(\mathfrak{R})) \subseteq \alpha^n(\mathfrak{R})$  as required.

If  $\mathcal{C} = \&[l_i : \langle \cdot \rangle^{k \in K}]_{i \in I}$ , then

$$S_1 = \mathcal{C}(![U].S_{1k})^{k \in K} = \&[l_i : ![U].S_{1i}]_{i \in I}$$

and

$$\begin{aligned} S'_1 &= ![U].\mathcal{C}(S_{1k})^{k \in K} = ![U].\&[l_i : S_{1i}]_{i \in I} \\ (S_1, S_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^n(S_2) = \&[l_j : S_{2j}]_{j \in J} \\ &\quad \wedge J \subseteq I \\ &\quad \wedge \forall j \in J. (![U].S_{1j}, S_{2j}) \in \alpha^{n-1}(\mathfrak{R}) \\ &\Rightarrow \forall j \in J. \text{unfold}^{m_j}(S_{2j}) = \mathcal{A}_j(![U'].S'_{2jh})^{h \in H_j} \\ &\quad \wedge (U', U) \in \alpha^{n-1}(\mathfrak{R}) \\ &\quad \wedge \forall j \in J. (S_{1j}, \mathcal{A}_j(S'_{2jh})^{h \in H_j}) \in \alpha^{n-1}(\mathfrak{R}) \end{aligned}$$

Let  $m_{\max} = \max_{j \in J}(m_j)$ . From the unfolding construction of  $\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$  we obtain

$$\forall j \in J. (S_{1j}, \text{unfold}^{m_{\max}-m_j}(\mathcal{A}_j(S'_{2jh})^{h \in H_j})) \in \mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$$

From the definition of  $n$ -times unfolding we obtain

$$\begin{aligned} \text{unfold}^{n+m_{\max}}(S_2) &= \&[l_j : \text{unfold}^{m_{\max}-m_j}(\mathcal{A}_j(![U'].S'_{2jh})^{h \in H_j})]_{j \in J} \\ &= \mathcal{A}'(![U'].S'_{2jh})^{h \in H} \end{aligned}$$

with

$$\mathcal{A}' = \&[l_j : \text{unfold}^{m_{\max}-m_j}(\mathcal{A}_j)]_{j \in J} \quad \text{and} \quad H = \uplus_{j \in J}(H_j)$$

Now we proceed to justify the inclusion  $(S'_1, S_2) \in \alpha^n(\mathfrak{R})$ . We have  $\text{unfold}^{n+m_{\max}}(S_2) = \mathcal{A}'(![U'].S'_{2jh})^{h \in H}$ , and  $(U', U) \in \alpha^{n-1}(\mathfrak{R})$ . We then need to show that  $(\&[l_i : S_{1i}]_{i \in I}, \mathcal{A}'(S'_{2jh})^{h \in H}) \in \alpha^n(\mathfrak{R})$ . Since  $J \subseteq I$  and  $\forall j \in J. (S_{1j}, \text{unfold}^{m_{\max}-m_j}(\mathcal{A}_j(S'_{2jh})^{h \in H_j})) \in \mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$ , we have that  $(\&[l_i : S_{1i}]_{i \in I}, \&[l_j : \text{unfold}^{m_{\max}-m_j}(\mathcal{A}_j(S'_{2jh})^{h \in H_j})]_{j \in J}) \in \mathcal{CE}(\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R})))$ , as required.



**Subcase**  $S_1 = \mathcal{C}(\oplus[l_i : S_{1ik}]_{i \in I_k})^{k \in K}$  and  $S'_1 = \oplus[l_i : \mathcal{C}(S_{1ik})]_{i \in I}^{k \in K}$  with  $\forall k \in K. I \subseteq I_k$ . We proceed with cases on  $\mathcal{C}$ .

If  $\mathcal{C} = ?[U_1].\langle \cdot \rangle^{k \in K}$ , then

$$S_1 = \mathcal{C}(\oplus[l_i : S_{1ik}]_{i \in I_k})^{k \in K} = ?[U_1].\oplus[l_i : S_{1i}]_{i \in I}$$

and

$$\begin{aligned} S'_1 &= \oplus[l_i : \mathcal{C}(S_{1ik})]_{i \in I}^{k \in K} = \oplus[l_i : ?[U_1].S_{1i}]_{i \in I} \\ (S_1, S_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^n(S_2) = ?[U_2].S'_2 \\ &\quad \wedge (U_1, U_2) \in \alpha^{n-1}(\mathfrak{R}) \\ &\quad \wedge (\oplus[l_i : S_{1i}]_{i \in I}, S'_2) \in \alpha^{n-1}(\mathfrak{R}) \\ (\oplus[l_i : S_{1i}]_{i \in I}, S'_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^m(S'_2) = \mathcal{A}_1(\oplus[l_j : S_{2jh}]_{j \in J_h})^{h \in H} \\ &\quad \wedge \forall h \in H. I \subseteq J_h \\ &\quad \wedge \forall i \in I. (S_{1i}, \mathcal{A}_1(S_{2ih})^{h \in H}) \in \alpha^{n-1}(\mathfrak{R}) \end{aligned}$$

From the definition of  $n$ -times unfolding we obtain

$$\text{unfold}^{n+m}(S_2) = ?[U_2].\mathcal{A}_1(\oplus[l_j : S_{2jh}]_{j \in J_h})^{h \in H} = \mathcal{A}_2(\oplus[l_j : S_{2jh}]_{j \in J_h})^{h \in H}$$

with

$$\mathcal{A}_2 = ?[U_2].\mathcal{A}_1$$

Now we proceed to justify the inclusion  $(S'_1, S_2) \in \alpha^n(\mathfrak{R})$ . We have  $(\oplus[l_i : ?[U_1].S_{1i}]_{i \in I}, S_2) \in \alpha^n(\mathfrak{R})$ . Also  $\text{unfold}^{n+m}(S_2) = \mathcal{A}_2(\oplus[l_j : S_{2jh}]_{j \in J_h})^{h \in H}$  with  $\forall h \in H. I \subseteq J_h$ . We then need to show that  $\forall i \in I. (?[U_1].S_{1i}, \mathcal{A}_2(S_{2ih})^{h \in H}) \in \alpha^n(\mathfrak{R})$  which can be written  $(?[U_1].S_{1i}, ?[U_2].\mathcal{A}_1(S_{2ih})^{h \in H}) \in \alpha^n(\mathfrak{R})$ . We have  $(U_1, U_2) \in \alpha^{n-1}(\mathfrak{R})$  and  $(S_{1i}, \mathcal{A}_1(S_{2ih})^{h \in H}) \in \alpha^{n-1}(\mathfrak{R})$ , hence we have  $(?[U_1].S_{1i}, \mathcal{A}_2(S_{2ih})^{h \in H}) \in \mathcal{CE}(\alpha^{n-1}(\mathfrak{R}))$  as required.

If  $\mathcal{C} = \&[l_i : \langle \cdot \rangle^{k \in K}]_{i \in I}$ , then

$$S_1 = \mathcal{C}(\oplus[l_i : S_{1ik}]_{i \in I_k})^{k \in K} = \&[l'_j : \oplus[l_i : S_{1ij}]_{i \in I_j}]_{j \in J}$$

and

$$\begin{aligned} S'_1 &= \oplus[l_i : \mathcal{C}(S_{1ik})]_{i \in I}^{k \in K} = \oplus[l_i : \&[l'_j : S_{1ij}]_{j \in J}]_{i \in I} \quad \forall j \in J. I \subseteq I_j \\ (S_1, S_2) \in \alpha^{n-1}(\mathfrak{R}) &\Rightarrow \text{unfold}^n(S_2) = \&[l'_z : S_{2z}]_{z \in Z} \\ &\quad \wedge Z \subseteq J \\ &\quad \wedge \forall z \in Z. (\oplus[l_i : S_{1iz}]_{i \in I_z}, S_{2z}) \in \alpha^{n-1}(\mathfrak{R}) \\ &\Rightarrow \forall z \in Z. \text{unfold}^{m_z}(S_{2z}) = \mathcal{A}_z(\oplus[l_j : S_{2zjh}]_{j \in J_h})^{h \in H_z} \\ &\quad \wedge \forall z \in Z. \forall h \in H_z. I_z \subseteq J_h \\ &\quad \wedge \forall z \in Z. \forall i \in I_z. (S_{1iz}, \mathcal{A}_z(S_{2zih})^{h \in H_z}) \in \alpha^{n-1}(\mathfrak{R}) \end{aligned}$$

Let  $m_{\max} = \max_{z \in Z} (m_z)$ . Then from the unfolding construction of  $\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$  we obtain

$$\forall z \in Z. \forall i \in I_z. (S_{1iz}, \text{unfold}^{m_{\max}-m_z}(\mathcal{A}_z(S_{2zih})^{h \in H_z})) \in \mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R}))$$

From the definition of  $n$ -times unfolding we obtain

$$\begin{aligned} \text{unfold}^{n+m_{\max}}(S_2) &= \&[l'_z : \mathcal{A}_z(\oplus[l_j : S_{2zjh}]_{j \in J_h})^{h \in H_z}]_{z \in Z} \\ &= \mathcal{A}'(\oplus[l_j : S_{2zjh}]_{j \in J_h})^{h \in H} \end{aligned}$$

with

$$\mathcal{A}' = \&[l'_z : \text{unfold}^{m_{\max}-m_z}(\mathcal{A}_z)]_{z \in Z} \quad \text{and} \quad H = \cup_{z \in Z} (H_z)$$

Now we proceed to justify the inclusion  $(\oplus[l_i : \&[l'_j : S_{1ij}]_{j \in J}]_{i \in I}, S_2) \in \alpha^n(\mathfrak{R})$ . We have  $\text{unfold}^{n+m_{\max}}(S_2) = \mathcal{A}'(\oplus[l_j : S_{2zjh}]_{j \in J_h})^{h \in H}$  and from  $I \subseteq I_z \subseteq J \subseteq J_h \in H_z$  we obtain  $\forall h \in H. I \subseteq J_h$ . We then need to show that  $\forall i \in I. (\&[l'_j : S_{1ij}]_{j \in J}, \mathcal{A}'(\oplus[l_j : S_{2zjh}]_{j \in J_h})^{h \in H_z}) \in \alpha^n(\mathfrak{R})$ . These pairs are in  $\mathcal{CE}(\mathcal{U}^\omega(\alpha^{n-1}(\mathfrak{R})))$  by construction, as required.

**Case**  $\mathcal{A} = ?[U].\mathcal{A}'$ . From the shape of  $\mathcal{A}$ , we have  $(?[U].\mathcal{A}'(S'_{1k})^{k \in K}, S_2) \in \alpha^{n-1}(\mathfrak{R})$ . By the rules of simulation,  $\text{unfold}^m(S_2) = ?[U'].S'_2$  and  $(U, U') \in \alpha^{n-1}(\mathfrak{R})$  and  $(\mathcal{A}'(S'_{1k})^{k \in K}, S'_2) \in \alpha^{n-1}(\mathfrak{R})$ . By the construction of  $\alpha^n(\mathfrak{R})$  we have  $(\mathcal{A}'(S'_{1k})^{k \in K}, S'_2) \in \alpha^n(\mathfrak{R})$ . It is now straightforward to show that  $(?[U].\mathcal{A}'(S'_{1k})^{k \in K}, S_2)$  is justified by the rules of simulation and the above hypotheses.

**Case**  $\mathcal{A} = \&[l_i : \mathcal{A}_i]_{i \in I}$ . From the shape of  $\mathcal{A}$ ,  $(\&[l_i : \mathcal{A}_i \langle S_{1k} \rangle^{k \in K}]_{i \in I}, S_2) \in \alpha^{n-1}(\mathfrak{N})$ . By the rules of simulation,  $\text{unfold}^m(S_2) = \&[l_j : S_{2j}]_{j \in J}$  and  $J \subseteq I$  and  $\forall j \in J. (\mathcal{A}_j \langle S_{1k} \rangle^{k \in K}, S_{2j}) \in \alpha^{n-1}(\mathfrak{N})$ . By the construction of  $\alpha^n(\mathfrak{N})$  we have  $\forall j \in J. (\mathcal{A}_j \langle S'_{1k} \rangle^{k \in K}, S_{2j}) \in \alpha^n(\mathfrak{N})$ . As before it is now trivial to justify  $(\&[l_i : \mathcal{A}_i \langle S'_{1k} \rangle^{k \in K}]_{i \in I}, S_2) \in \alpha^n(\mathfrak{N})$ .  $\square$

**Corollary B.9** (Multi-step permutation).

1. If  $(\mathcal{A} \langle ! [U_1]. S_{1k} \rangle^{k \in K}, S_2) \in \alpha^\omega(\mathfrak{N})$  then  $(! [U_1]. \mathcal{A} \langle S_{1k} \rangle^{k \in K}, S_2) \in \alpha^\omega(\mathfrak{N})$
2. If  $(\mathcal{A} \langle \oplus [l_i : S_{1ih}]_{i \in I_h} \rangle^{h \in H}, S_2) \in \alpha^\omega(\mathfrak{N})$  and  $\forall h \in H. I \subseteq I_h$ , then  $(\oplus [l_i : \mathcal{A} \langle S_{1ih} \rangle^{h \in H}]_{i \in I}, S_2) \in \alpha^\omega(\mathfrak{N})$ .

As before, the notation  $\alpha^\omega(\mathfrak{N})$  stands for the union of all  $\alpha^n(\mathfrak{N})$  with  $n \in \mathbb{N}$ .

**Proof.** Every context  $\mathcal{A}$  can be written as a (possibly empty) nested structure of  $\mathcal{C}$  contexts, such that  $\mathcal{A} = \mathcal{C} \langle \mathcal{C}_h \langle \mathcal{C}_{hk} \langle \dots \rangle^{k \in K} \rangle^{h \in H} \rangle^{h \in H}$ . Every level of asynchronous permutation in  $\alpha^\omega(\mathfrak{N})$  generates pairs by applying a transformation on the innermost  $\mathcal{C}$  contexts of all matching types; in this way it reduces the depth of the innermost contexts for the generated type pairs, which are matched at the next level. At every level, the penultimate contexts become last. By induction on the maximum depth of the nested  $\mathcal{C}$ -context representation of any  $\mathcal{A}$ , we can obtain the result formally.  $\square$

**Proposition B.2.** If  $(S_1, S_2) \in \alpha^\omega(\mathfrak{N})$  then  $(\text{unfold}^n(S_1), S_2) \in \alpha^\omega(\mathfrak{N})$ .

**Proof.** Easy to obtain: since  $\gg$  allows the identity permutation, then for all  $m$ ,  $\alpha^m(\mathfrak{N})$  will include  $\mathcal{CE}(\mathcal{U}^\omega(\alpha^{m-1}(\mathfrak{N}))) \supseteq \mathcal{U}^\omega(\alpha^{m-1}(\mathfrak{N}))$  even when there are no more effective permutations to apply on any type, and up to all contexts. Suppose  $(S_1, S_2) \in \alpha^z(\mathfrak{N})$ , take  $m = z + n + 1$ , and we obtain the result  $(\text{unfold}^n(S_1), S_2) \in \alpha^m(\mathfrak{N})$ .  $\square$

**Transitivity connection** Next is the main definition of this section, the *transitivity connection* of two relations. It is defined as the relational composition (taking the union of both directions, needed due to the presence of contravariant components) of the asynchronous extensions of the given simulations, respectively. We then prove that the transitivity connection (of simulations) is a simulation, which is, effectively, a proof of the transitivity of  $\leq_c$ .

**Definition B.10** (Transitivity connection). For type simulations  $\mathfrak{N}_1$  and  $\mathfrak{N}_2$ , the *transitivity connection*  $\text{trc}(\mathfrak{N}_1, \mathfrak{N}_2)$  is defined as follows:

$$\text{trc}(\mathfrak{N}_1, \mathfrak{N}_2) = \alpha^\omega(\mathfrak{N}_1) \cdot \alpha^\omega(\mathfrak{N}_2) \cup \alpha^\omega(\mathfrak{N}_2) \cdot \alpha^\omega(\mathfrak{N}_1)$$

**Lemma B.11.** If  $\mathfrak{N}_{i \in \{1,2\}} \subseteq \leq_c$  then  $\text{trc}(\mathfrak{N}_1, \mathfrak{N}_2) \subseteq \leq_c$ . That is, for any two simulations  $\mathfrak{N}_1$  and  $\mathfrak{N}_2$ , the transitivity connection  $\text{trc}(\mathfrak{N}_1, \mathfrak{N}_2)$  is a type simulation.

**Proof.** We examine an arbitrary  $(T_1, T_3) \in \alpha^\omega(\mathfrak{N}_1) \cdot \alpha^\omega(\mathfrak{N}_2) \subseteq \text{trc}(\mathfrak{N}_1, \mathfrak{N}_2)$ , taking cases on the shape of  $T_1$ . The remaining cases, for membership in  $\alpha^\omega(\mathfrak{N}_2) \cdot \alpha^\omega(\mathfrak{N}_1)$ , are symmetric.

**Case**  $T_1 = ! [U_1]. S_1$ . Then  $(T_1, T_2) \in \alpha^\omega(\mathfrak{N}_1)$  and  $(T_2, T_3) \in \alpha^\omega(\mathfrak{N}_2)$ .

$$\begin{aligned} (! [U_1]. S_1, T_2) \in \alpha^\omega(\mathfrak{N}_1) &\Rightarrow \text{unfold}^n(T_2) = \mathcal{A}_1 \langle ! [U_2]. S_{2h} \rangle^{h \in H} \\ &\quad \wedge (U_2, U_1) \in \alpha^\omega(\mathfrak{N}_1) \\ &\quad \wedge (S_1, \mathcal{A}_1 \langle S_{2h} \rangle^{h \in H}) \in \alpha^\omega(\mathfrak{N}_1) \end{aligned}$$

$$\begin{aligned} (T_2, T_3) \in \alpha^\omega(\mathfrak{N}_2) &\Rightarrow (\text{unfold}^n(T_2), T_3) \in \alpha^\omega(\mathfrak{N}_2) \\ &\Leftrightarrow (\mathcal{A}_1 \langle ! [U_2]. S_{2h} \rangle^{h \in H}, T_3) \in \alpha^\omega(\mathfrak{N}_2) \end{aligned}$$

$$\begin{aligned} \text{Corollary B.9} &\Rightarrow (! [U_2]. \mathcal{A}_1 \langle S_{2h} \rangle^{h \in H}, T_3) \in \alpha^\omega(\mathfrak{N}_2) \\ &\Rightarrow \text{unfold}^m(T_3) = \mathcal{A}_2 \langle ! [U_3]. S_{3k} \rangle^{k \in K} \\ &\quad \wedge (U_3, U_2) \in \alpha^\omega(\mathfrak{N}_2) \\ &\quad \wedge (\mathcal{A}_1 \langle S_{2h} \rangle^{h \in H}, \mathcal{A}_2 \langle S_{3k} \rangle^{k \in K}) \in \alpha^\omega(\mathfrak{N}_2) \end{aligned}$$

$$\begin{aligned} U_{i \in \{1,2\}} = S'_i &\Rightarrow (U_3, U_1) = (U_3, U_1) \in \alpha^\omega(\mathfrak{N}_2) \cdot \alpha^\omega(\mathfrak{N}_1) \\ &\quad \wedge (S_1, \mathcal{A}_2 \langle S_{3k} \rangle^{k \in K}) \in \alpha^\omega(\mathfrak{N}_1) \cdot \alpha^\omega(\mathfrak{N}_2) \end{aligned}$$

$$\begin{aligned} U_{i \in \{1,2\}} \neq S &\Rightarrow (U_3, U_1) = (U_1, U_3) \in \alpha^\omega(\mathfrak{N}_1) \cdot \alpha^\omega(\mathfrak{N}_2) \\ &\quad \wedge (S_1, \mathcal{A}_2 \langle S_{3k} \rangle^{k \in K}) \in \alpha^\omega(\mathfrak{N}_1) \cdot \alpha^\omega(\mathfrak{N}_2) \end{aligned}$$

Hence  $(T_1, T_3)$  is justified in  $\text{trc}(\mathfrak{N}_1, \mathfrak{N}_2)$ .

**Case**  $T_1 = ?[U_1].S_1$ . Then  $(T_1, T_2) \in \alpha^\omega(\mathfrak{R}_1)$  and  $(T_2, T_3) \in \alpha^\omega(\mathfrak{R}_2)$ .

$$\begin{aligned} (?[U_1].S_1, T_2) \in \alpha^\omega(\mathfrak{R}_1) &\Rightarrow \text{unfold}^n(T_2) = ?[U_2].S_2 \\ &\wedge (U_1, U_2) \in \alpha^\omega(\mathfrak{R}_1) \\ &\wedge (S_1, S_2) \in \alpha^\omega(\mathfrak{R}_1) \end{aligned}$$

$$\begin{aligned} (T_2, T_3) \in \alpha^\omega(\mathfrak{R}_2) &\Rightarrow (\text{unfold}^n(T_2), T_3) \in \alpha^\omega(\mathfrak{R}_2) \\ &\Leftrightarrow (?[U_2].S_2, T_3) \in \alpha^\omega(\mathfrak{R}_2) \\ &\Rightarrow \text{unfold}^m(T_3) = ?[U_3].S_3 \\ &\wedge (U_2, U_3) \in \alpha^\omega(\mathfrak{R}_2) \\ &\wedge (S_2, S_3) \in \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

$$\begin{aligned} U_{i \in \{1,2\}} = S'_i &\Rightarrow (U_1, U_3) = (U_1, U_3) \in \alpha^\omega(\mathfrak{R}_1) \cdot \alpha^\omega(\mathfrak{R}_2) \\ &\wedge (S_1, S_3) \in \alpha^\omega(\mathfrak{R}_1) \cdot \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

$$\begin{aligned} U_{i \in \{1,2\}} \neq S &\Rightarrow (U_1, U_3) = (U_3, U_1) \in \alpha^\omega(\mathfrak{R}_2) \cdot \alpha^\omega(\mathfrak{R}_1) \\ &\wedge (S_1, S_3) \in \alpha^\omega(\mathfrak{R}_1) \cdot \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

Hence, as before,  $(T_1, T_3)$  is justified in  $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ .

**Case**  $T_1 = \oplus[l_i : S_{1i}]_{i \in I}$ . Then  $(T_1, T_2) \in \alpha^\omega(\mathfrak{R}_1)$  and  $(T_2, T_3) \in \alpha^\omega(\mathfrak{R}_2)$ .

$$\begin{aligned} (\oplus[l_i : S_{1i}]_{i \in I}, T_2) \in \alpha^\omega(\mathfrak{R}_1) &\Rightarrow \text{unfold}^n(T_2) = \mathcal{A}_1 \langle \oplus[l_j : S_{2jh}]_{j \in J_h} \rangle^{h \in H} \\ &\wedge \forall h \in H. I \subseteq J_h \\ &\wedge \forall i \in I. (S_{1i}, \mathcal{A}_1 \langle S_{2ih} \rangle^{h \in H}) \in \alpha^\omega(\mathfrak{R}_1) \end{aligned}$$

$$\begin{aligned} (T_2, T_3) \in \alpha^\omega(\mathfrak{R}_2) &\Rightarrow (\text{unfold}^n(T_2), T_3) \in \alpha^\omega(\mathfrak{R}_2) \\ &\Leftrightarrow (\mathcal{A}_1 \langle \oplus[l_j : S_{2jh}]_{j \in J_h} \rangle^{h \in H}, T_3) \in \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

$$\begin{aligned} \text{Corollary B.9 with } I \subseteq J_{h \in H} &\Rightarrow (\oplus[l_i : \mathcal{A}_1 \langle S_{2ih} \rangle^{h \in H}]_{i \in I}, T_3) \in \alpha^\omega(\mathfrak{R}_2) \\ &\Rightarrow \text{unfold}^m(T_3) = \mathcal{A}_2 \langle \oplus[l_z : S_{2zk}]_{z \in Z_k} \rangle^{k \in K} \\ &\wedge \forall k \in K. I \subseteq Z_k \\ &\wedge \forall i \in I. (\mathcal{A}_1 \langle S_{2ih} \rangle^{h \in H}, \mathcal{A}_2 \langle S_{3ik} \rangle^{k \in K}) \in \alpha^\omega(\mathfrak{R}_2) \\ &\Rightarrow \forall i \in I. (S_{1i}, \mathcal{A}_2 \langle S_{3ik} \rangle^{k \in K}) \in \alpha^\omega(\mathfrak{R}_1) \cdot \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

Hence  $(T_1, T_3)$  is justified in  $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ .

**Case**  $T_1 = \mu \mathbf{t}.S_1$ . Then  $(T_1, T_2) \in \alpha^\omega(\mathfrak{R}_1)$  and  $(T_2, T_3) \in \alpha^\omega(\mathfrak{R}_2)$ .

$$\begin{aligned} (\mu \mathbf{t}.S_1, T_2) \in \alpha^\omega(\mathfrak{R}_1) &\Rightarrow (\text{unfold}^1(\mu \mathbf{t}.S_1), T_2) \in \alpha^\omega(\mathfrak{R}_1) \\ &\Rightarrow (\text{unfold}^1(\mu \mathbf{t}.S_1), T_3) \in \alpha^\omega(\mathfrak{R}_1) \cdot \alpha^\omega(\mathfrak{R}_2) \end{aligned}$$

Thus,  $(T_1, T_3)$  is justified in  $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ .

Other cases are similar, and in fact simpler, because they make no use of asynchronous contexts and permutations.  $\square$

**Proof of Theorem 4.4.** For reflexivity it is easy to prove  $\{(T, T) \mid T \in \mathcal{T}\} \subseteq \leq_c$ . For transitivity, we have that whenever  $(T_1, T_2) \in \mathfrak{R}_1$  and  $(T_2, T_3) \in \mathfrak{R}_2$ , then  $(T_1, T_3) \in \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ , and  $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2) \subseteq \leq_c$  by Lemma B.11.  $\square$

## Appendix C. Proofs of type soundness

We proceed to show that typed processes enjoy type soundness and type safety. We begin with a number of auxiliary properties, and then prove the Substitution Lemma (page 30).

**Lemma C.1** (*Closed judgement*). *If  $\Gamma; \Lambda; \Sigma \vdash P : T$  and  $x \in \text{fv}(P)$  then  $x \in \text{dom}(\Gamma) \cup \text{dom}(\Lambda) \cup \text{dom}(\Sigma)$ .*

**Proof.** By induction on the typing derivation for  $P$ . The interesting cases are the axioms which form the leaves of a derivation. If the last rule is (Shared), (LVar), or (Session), then  $P = x$  and  $x$  appears in one of typing environments, depending on which axiom was applied. The other cases are easy to obtain using the inductive hypothesis.  $\square$

We have the standard weakening and strengthening for  $\Gamma$ , but not for  $\Lambda$  and  $\Sigma$ .

**Lemma C.2** ( $\Gamma$ -weakening). If  $\Gamma; \Lambda; \Sigma \vdash P : T$  and  $x \notin \text{dom}(\Gamma, \Lambda, \Sigma)$  then  $\Gamma, x:U; \Lambda; \Sigma \vdash P : T$ .

**Lemma C.3** ( $\Gamma$ -strengthening). If  $\Gamma, x:U; \Lambda; \Sigma \vdash P : T$  and  $x \notin \text{fv}(P)$  then  $\Gamma; \Lambda; \Sigma \vdash P : T$ .

The typing rule (Close) can be used to introduce arbitrary, but ended, hypotheses to the session environment. This is a form of weakening, albeit restricted, and we introduce the following lemma so that we can strengthen the hypotheses by removing any one introduced by (Close). This lemma is used in the proof of Structural Congruence.

**Lemma C.4** ( $\Sigma$ -strengthening). If  $\Gamma; \Lambda; \Sigma, k:\text{end} \vdash P : T$  and  $k \notin \text{fn}(P)$  then  $\Gamma; \Lambda; \Sigma \vdash P : T$ .

**Proof.** By induction on the typing derivation for  $P$ .  $\square$

**Lemma C.5** (Linear variable occurrence). If  $\Gamma; \Lambda, x:U \multimap T; \Sigma \vdash P : T$  then  $x \in \text{fv}(P)$ .

**Proof.** By induction on the typing derivation for  $P$ . Most cases are straightforward, using the inductive hypothesis. The interesting case is for (LVar), where  $P = x$ , proving the occurrence of the linear variable.  $\square$

**Lemma C.6** (Endpoint occurrence). If  $\Gamma; \Lambda; \Sigma, x:S \vdash P : T$  and  $S \neq \text{end}$  then  $x \in \text{fv}(P)$ .

**Proof.** By induction on the typing derivation for  $P$ . Most cases are straightforward, using the inductive hypothesis. The interesting case is for (Session), where  $P = x$ , proving the occurrence of the endpoint. The sidecondition  $S \neq \text{end}$  serves to exclude the cases where  $x$  appears in the session environment by introduction through (Close).  $\square$

**Lemma C.7** (Ended session). If  $\Gamma; \Lambda; \Sigma, x:S \vdash P : T$  and  $x \notin \text{fv}(P)$  then  $S = \text{end}$ .

**Proof.** By induction on the typing derivation for  $P$ . Most cases are straightforward, using the inductive hypothesis. The interesting case is when the last rule applied was (Close), which does not require  $x$  to be free in the term, and also implies that  $S = \text{end}$ .  $\square$

**Lemma C.8** (Linear unique occurrence). If  $\Gamma; \Lambda, x:U \multimap T; \Sigma \vdash P : T$ , and  $P = Q_1 \cdot Q_2$  or  $P = Q_1 \mid Q_2$  or  $P = k!\langle Q_1 \rangle.Q_2$  (in the last case  $Q_1 = V$ ), then  $x \notin \text{fv}(Q_i)$  for  $i = 1$  or  $i = 2$ .

**Proof.** We proceed by induction on the typing derivation for  $P$ . Note that we have  $x \in \text{fv}(P)$  by Lemma C.5. Suppose  $x \in \text{fv}(Q_1)$ . Assume  $\Lambda, x:U \multimap T \equiv \Lambda_1, \Lambda_2$  and  $\Sigma \equiv \Sigma_1, \Sigma_2$ . Let  $\Gamma; \Lambda_1; \Sigma_1 \vdash Q_1 : T_1$  (1) and  $\Gamma; \Lambda_2; \Sigma_2 \vdash Q_2 : T_2$  from the I.H. on the premises of the last rule applied; this was either (App) or (Par) or (Send). From Lemma C.1 we know that since  $x$  is free in  $Q_1$  it appears in one of the typing environments of (1), and in particular  $\Lambda_1$  since by the well-formedness of the assumed judgement for  $P$  it cannot appear in  $\Gamma$  or  $\Sigma_1 \subseteq \Sigma$  when it appears in  $\Lambda, x:U \multimap T$ . Now assume additionally that  $x \in \text{fv}(Q_2)$ . Then by Lemma C.1 we have that  $x \in \text{dom}(\Gamma, \Lambda_2, \Sigma_2)$  which is a contradiction since by the well-formedness of the judgement for  $P$  we have that  $x$  cannot appear in  $\Gamma$  or  $\Lambda_2 \subseteq \Lambda$  or  $\Sigma_2 \subseteq \Sigma$ . Hence  $x \notin \text{fv}(Q_2)$ . The case for  $x \notin \text{fv}(Q_1)$  is symmetric.  $\square$

**Lemma C.9** (Endpoint unique occurrence). If  $\Gamma; \Lambda; \Sigma, x:S \vdash P : T$ , and  $P = Q_1 \cdot Q_2$  or  $P = Q_1 \mid Q_2$  or  $P = k!\langle Q_1 \rangle.Q_2$  (in the last case  $Q_1 = V$ ), then  $x \notin \text{fv}(Q_i)$  for  $i = 1$  or  $i = 2$ .

**Proof.** The proof is by induction on the typing derivation, and follows the same pattern as in Lemma C.8. When  $x \notin \text{fv}(P)$ , which is a possibility due to (Close), the result is immediate. When  $x \in \text{fv}(P)$  we proceed as in Lemma C.8.  $\square$

The Substitution Lemma which follows is mostly standard, noting that we only need to define substitution for terms that do not contain runtime elements, i.e., we do not need to consider queues.

**Lemma C.10** (Substitution lemma).

1. Suppose  $\Gamma, x:U; \Lambda; \Sigma \vdash P : T$  and  $\Gamma; \emptyset; \emptyset \vdash V : U$ . Then  $\Gamma; \Lambda, \Sigma \vdash P\{V/x\} : T$ .
2. Assume  $\Gamma; \Lambda_1, x:U \multimap T'; \Sigma_1 \vdash P : T$  and  $\Gamma; \Lambda_2; \Sigma_2 \vdash V : U \multimap T'$  with  $\Lambda_1, \Lambda_2$  and  $\Sigma_1, \Sigma_2$  defined. Then  $\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \vdash P\{V/x\} : T$ .
3. Suppose  $\Gamma; \Lambda; \Sigma, x:S \vdash P : T$  and  $k \notin \text{dom}(\Gamma, \Lambda, \Sigma)$ . Then  $\Gamma; \Lambda; \Sigma, k:S \vdash P\{k/x\} : T$ .

**Proof.** The proof is by induction on the last rule applied in the typing derivation for  $P$ . In Part (1) we do not state cases where substitution has no effect, as these can be shown trivially from the assumptions with strengthening on the hypothesis for  $x$  in  $\Gamma, x:U$ . In Part (2), we assume that substitution is only applied when  $x \in \text{fv}(P)$ , which is correct since in any judgement,  $x \in \text{dom}(\Lambda)$  implies that  $x$  occurs in the term (see Lemma C.5). For Part (3) we cannot assume that  $x \in \text{fv}(P)$ , since usages of the shape  $x:\text{end}$  can be obtained using (Close) even when  $x$  is not free in the term.

Part (1)

**Case (Shared)**  $P = x \quad T = U \quad \Lambda = \Sigma = \emptyset$

We have  $P\{V/x\} = V$  by the hypotheses, then  $T = U$  and  $\Lambda = \Sigma = \emptyset$ . Then we use  $\Gamma; \emptyset; \emptyset \vdash V : U$  to obtain the required judgement  $\Gamma; \Lambda, \Sigma \vdash P\{V/x\} : T$ .

**Case (LVar)**  $P = x$  is excluded because  $x \in \text{dom}(\Lambda)$  implies  $x \notin \text{dom}(\Gamma)$  by the well-formedness of the judgement for  $P$ . This case is proved in Part (2).

**Case (Session)**  $P = x$  is excluded because  $x \in \text{dom}(\Sigma)$  implies  $x \notin \text{dom}(\Gamma)$  by the well-formedness of the judgement for  $P$ . This case is proved in Part (3).

**Case (Sub)** Trivial to show using the I.H. on the premise followed by an application of (Sub).

**Case (Promotion, Dereliction)** Easy to show using the I.H. on the premise followed by an application of the same rule.

**Case (Abs)**  $P = \lambda(z : U_1).Q \quad z \neq x \quad T = U_1 \multimap T_1$

From the I.H. on the premises we have  $(\Gamma, x:U; \Lambda; \Sigma) \S z:U_1 \vdash Q\{V/x\} : T_1$  (1). With an application of (Abs) on (1), binding variable  $z$ , we obtain  $\Gamma, x:U; \Lambda; \Sigma \vdash \lambda(z : U_1).Q\{V/x\} : T$  (2). Now, since by the substitution we have that  $x \notin \text{fv}(\lambda(z : U_1).Q\{V/x\})$ , we use strengthening on (2) to remove the hypothesis for  $x$  and obtain the required judgement.

**Case (Rec)** is very similar to the case for (Abs).

**Case (App)**  $P = Q_1 \cdot Q_2 \quad \Lambda = \Lambda_1, \Lambda_2 \quad \Sigma = \Sigma_1, \Sigma_2$

From the premises we obtain  $\Gamma, x:U; \Lambda_i; \Sigma_i \vdash Q_i\{V/x\} : T_i$  with  $T_1 = U' \multimap T$  or  $T_1 = U' \rightarrow T$  and  $T_2 = U'$ . We then apply (App) with the above judgements in the premises, and obtain the result. Strengthening to remove the hypothesis for  $x$  (which is not free in the resulting term) is the last step.

**Case (Nil), (New), (New<sub>s</sub>)** are all straightforward to obtain from the premises using the I.H. followed by an application of the respective rule. Removing the hypothesis for  $x$  is used as before to obtain the desired shared environment for the final judgement.

**Case (Conn)**  $P = u(z).Q \quad z \neq x \quad T = \diamond$

We take the following cases:

1. Suppose  $u = x$ . Then we have  $\Gamma, x:U; \Lambda; \Sigma \vdash x(z).Q : \diamond$  (1). Also  $V = u'$  and from the assumptions  $\Gamma; \emptyset; \emptyset \vdash u' : U$  (2) with  $U = \langle S \rangle$ . We have  $P\{V/x\} = u'(z).Q\{V/x\}$ . From (1) we obtain the premise  $\Gamma, x:U; \Lambda; \Sigma, z:S \vdash Q : \diamond$  (3). Applying the I.H. on (3) we get  $\Gamma; \Lambda; \Sigma, z:S \vdash Q\{V/x\} : \diamond$  (4). We now apply (Conn) with (2) and (4) to obtain  $\Gamma; \Lambda; \Sigma \vdash u'(z).Q\{V/x\} : \diamond$  as required.
2. Suppose  $u \neq x$ . Then  $P\{V/x\} = u(z).Q\{V/x\}$ . From the assumption for  $P$  we obtain the premise  $\Gamma, x:U; \Lambda; \Sigma \vdash u : \langle S \rangle$  (5). Then since  $x \neq u$  we can strengthen the hypotheses and obtain  $\Gamma; \Lambda; \Sigma \vdash u : \langle S \rangle$  (6). We obtain (4) as before, and apply (Conn) using (4) and (6) to obtain the required judgement.

**Case (ConnDual)** is very similar to (Conn).

**Case (Recv)** is very similar to (Abs).

**Case (Send)** is similar to (App).

**Case (Par)** is straightforward to obtain using the I.H. on the premises followed by an application of (Par).

**Case (Close)** is straightforward to obtain using the I.H. on the premises followed by an application of (Close).

**Case (Bra), (Sel)** is easy to prove using the I.H. on the premises. Note that  $k \neq x$  by the assumptions since  $x$  is assigned a shared type.

Part (2)

**Case (Shared)**  $P = x$  is excluded because  $x$  appears in the linear function environment and therefore cannot also be in the shared environment as required by (Shared).

**Case (LVar)**  $P = x \quad T = U \multimap T' \quad \Lambda_1 = \emptyset \quad \Sigma_1 = \emptyset$

From the assumed judgement for  $P$  (note that  $\Lambda_1 = \emptyset$  and  $\Sigma_1 = \emptyset$ ) we have  $\Gamma; \{x:U \multimap T'\}; \emptyset \vdash x:T$ . Then from the assumed judgement for  $V$  we have  $\Gamma; \Lambda_2; \Sigma_2 \vdash V:U \multimap T'$  and since  $P\{V/x\} = V$  and  $\Lambda_1 = \Sigma_1 = \emptyset$ , this is the required typing judgement.

**Case (Session)**  $P = k = x$  is excluded because  $x$  appears in the linear function environment and therefore cannot also be in the session environment as required by (Session), by well-formedness.

**Case (Sub)** As in Part (1), trivial to show using the I.H. on the premise followed by an application of (Sub).

**Case (Abs)**  $P = \lambda(z:U_1).Q \quad z \neq x \quad T = U_1 \multimap T_1$

From the I.H. on the premises of the judgement for  $P$  we have (with the hypothesis for  $x$  now removed from the linear environment)  $(\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2) \S z:U_1 \vdash Q\{V/x\}:T_1$  (1). With an application of (Abs) on (1), binding variable  $z$ , we obtain  $\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \vdash \lambda(z:U_1).Q\{V/x\}:T$  as required.

**Case (Rec)** is similar to the case for (Abs).

**Case (App)**  $P = Q_1 \cdot Q_2 \quad \Sigma_1 = \Sigma_{11}, \Sigma_{12} \quad \Lambda_1, x:U \multimap T' = \Lambda_{11}, \Lambda_{12}$

From the assumption  $\Gamma; \Lambda_1, x:U \multimap T'; \Sigma_1 \vdash P:T$  (1) and [Lemma C.5](#) we have that  $x \in \text{fv}(P)$ . Using  $P = Q_1 \cdot Q_2$  with [Lemma C.8](#) on the assumption we also have that  $x \notin \text{fv}(Q_i)$  for some  $i \in \{1, 2\}$ . We can therefore take two cases:

1. Take  $x \in \text{fv}(Q_1)$  and  $x \notin \text{fv}(Q_2)$ . Then  $P\{V/x\} = Q_1\{V/x\} \cdot Q_2$ . The last rule applied (modulo (Sub)) is (App). By the I.H. on the premise  $\Gamma; \Lambda'_{11}, x:U \multimap T'; \Sigma_{11} \vdash Q_1:T_1$  of (1), where  $T_1$  is  $U_1 \multimap T$  or  $U_1 \rightarrow T$ , we obtain  $\Gamma; \Lambda'_{11}, \Lambda_2; \Sigma_{11}, \Sigma_2 \vdash Q_1\{V/x\}:T_1$  (2). The other premise of (1) is  $\Gamma; \Lambda_{12}; \Sigma_{12} \vdash Q_2:U_1$  (3). Now we can apply (App) with (2) and (3) as premises. We thus obtain the required judgement.
2. The case  $x \in \text{fv}(Q_2)$  and  $x \notin \text{fv}(Q_1)$  is symmetric. One note is that if  $U_1$  is an  $\rightarrow$ -type, then by (Promotion) we have that  $\Lambda_{12} = \Sigma_{12} = \emptyset$ , and  $x \notin \text{dom}(\Gamma)$  by WF of the assumption, hence in that case we have a contradiction since it must hold that  $x \notin \text{fv}(Q_2)$  by [Lemma C.1](#). This verifies our intuition that if a linear variable  $x$  appears in  $Q_2$ , then  $Q_2$  cannot be typed with a shared function type.

**Case (Nil), (New), (New<sub>s</sub>)** are straightforward using the I.H.

**Case (Conn), (ConnDual)** follow a similar pattern to the same cases in Part (1). The proof is slightly simpler since we have that if  $P = u(x).Q$  then since  $x$  is a linear function variable  $u \neq x$ .

**Case (Recv)** is very similar to (Abs).

**Case (Send)** is similar to (App).

**Case (Par)** is straightforward to obtain, as before, using the I.H. on the premises followed by an application of (Par).

**Case (Close)** is straightforward as in the previous part.

**Case (Bra), (Sel)** is easy to prove using the I.H. on the premises. Note that  $k \neq x$  by the assumptions since  $x$  is assigned a linear type.

*Part (3)*

Most cases are straightforward as before. For the case (App), (Par), the proof is similar to the other parts but makes use of [Lemma C.9](#) (instead of [Lemma C.8](#)).

**Case (Recv)**  $P = k?(z).Q \quad T = \diamond$

We have  $\Gamma; \Lambda; \Sigma, x:S \vdash k?(z).Q:\diamond$ . Then we take cases on  $k'$ .

1.  $k' = x$ . Then  $P\{k/x\} = k?(z).Q\{k/x\}$  and  $S = ?[U].S'$ . From the premises of (Recv) we obtain  $(\Gamma; \Lambda; \Sigma, x:S') \S z:U \vdash Q:\diamond$ . By the I.H. we have  $(\Gamma; \Lambda; \Sigma, k:S') \S z:U \vdash Q\{k/x\}:\diamond$ . Then with an application of (Recv) we obtain  $\Gamma; \Lambda; \Sigma, k:?[U].S' \vdash k?(z).Q\{k/x\}:\diamond$  as required.
2.  $k' \neq x$ . Then  $P\{k/x\} = k'? (z).Q\{k/x\}$  and  $\Sigma = \Sigma', k':?[U].S'$ . As before by the premises  $(\Gamma; \Lambda; \Sigma', k':S', x:S) \S z:U \vdash Q:\diamond$ . By the I.H.  $(\Gamma; \Lambda; \Sigma', k':S', k:S) \S z:U \vdash Q\{k/x\}:\diamond$ . Then with an application of (Recv) we obtain  $\Gamma; \Lambda; \Sigma, k:S \vdash k'? (z).Q\{k/x\}:\diamond$  as required.



**Case (Send)**  $P = k'!\langle V \rangle.Q \quad T = \diamond \quad \Sigma, x : S = (\Sigma_1, \Sigma_2) \setminus \{k' : S'\}, k' : !\langle U \rangle.S'$   
 $\Lambda = \Lambda_1, \Lambda_2$

From the premises of (Send) we have:

$$\Gamma; \Lambda_1; \Sigma_1 \vdash Q : \diamond \quad (1)$$

$$\Gamma; \Lambda_2; \Sigma_2 \vdash V : U \quad (2)$$

$$k' : S' \in \Sigma_i \quad i = 1 \text{ or } i = 2 \quad (3)$$

Then we perform case analysis on  $k'$ :

1. Suppose  $k' = x$ . Then  $S = !\langle U \rangle.S'$ . We now look at the occurrence of  $x$  in the session environments:
  - (a) Let  $x : S' \in \Sigma_1$ , then  $\Sigma_1 = \Sigma'_1, x : S'$  and  $P\{k/x\} = k'!\langle V \rangle.Q\{k/x\}$ . Using the I.H. on (1) we obtain  $\Gamma; \Lambda_1; \Sigma'_1, k : S' \vdash Q\{k/x\} : \diamond$  (3). Then we apply (Send) with (3) and (2), with the sideconditions clearly satisfied from the assumptions, to obtain  $\Gamma; \Lambda; \Sigma'_1, \Sigma_2, k : !\langle U \rangle.S' \vdash k'!\langle V \rangle.Q\{k/x\} : \diamond$  as required.
  - (b) Let  $x : S' \in \Sigma_2$ , then  $\Sigma_2 = \Sigma'_2, x : S'$  and  $P\{k/x\} = k'!\langle V \rangle.Q\{k/x\}$ . From the I.H. on (2) we have  $\Gamma; \Lambda_2; \Sigma'_2, k : S' \vdash V\{k/x\} : U$  (4). In this case we have  $U \neq U' \rightarrow T'$  otherwise  $\Sigma_2$  would be the empty set. We now apply (Send) with (1) and (4) to obtain  $\Gamma; \Lambda; \Sigma_1, \Sigma'_2, k : !\langle U \rangle.S' \vdash k'!\langle V \rangle.Q\{k/x\} : \diamond$  as required.
2. Suppose  $k' \neq x$ . As above we look at the occurrence of  $x$  in the session environments. Since  $k' \neq x$  we have two cases:
  - (a) Let  $x : S \in \Sigma_1$ , then  $\Sigma_1 = \Sigma'_1, x : S$  and  $P\{k/x\} = k'!\langle V \rangle.Q\{k/x\}$ . By the I.H. on (1),  $\Gamma; \Lambda_1; \Sigma'_1, k : S \vdash Q\{k/x\} : \diamond$  (5). We apply (Send) as before with (5) and (2), the sideconditions are satisfied (we do not check where  $k'$  occurs in the session environments  $\Sigma'_1$  and  $\Sigma_2$  as the sidecondition is met by the assumptions), and obtain  $\Gamma; \Lambda; (\Sigma'_1, \Sigma_2, k : S) \setminus \{k' : S'\}, k' : !\langle U \rangle.S' \vdash k'!\langle V \rangle.Q\{k/x\} : \diamond$  as required.
  - (b) Let  $x : S \in \Sigma_2$ , then  $\Sigma_2 = \Sigma'_2, x : S$  and  $P\{k/x\} = k'!\langle V \rangle.Q\{k/x\}$ . Using the same sequence of steps as before we obtain the result.

**Case (Close)**

Suppose (Close) was applied for some  $k'$ . Then the premise obtained is:

$$\Gamma; \Lambda; (\Sigma, x : S) \setminus \{k' : \text{end}\} \vdash P : T \quad (1)$$

We now distinguish two cases:

1.  $x = k'$ . Then  $S = \text{end}$ . Also,  $x \notin \text{dom}(\Gamma, \Lambda, \Sigma)$  by the well-formedness of (1). By Lemma C.1  $x \in \text{fv}(P)$  implies  $x \in \text{dom}(\Gamma, \Lambda, \Sigma)$  thus we have  $x \notin \text{fv}(P)$ . Then  $P\{k/x\} = P$ . From (1) we obtain:

$$\Gamma; \Lambda; \Sigma \vdash P : T \quad (2)$$

We have  $k \notin \text{dom}(\Gamma, \Lambda, \Sigma)$  by assumption, and we can apply (Close) to obtain:

$$\Gamma; \Lambda; \Sigma, k : \text{end} \vdash P : T$$

which is the desired result, since  $S = \text{end}$  and  $P\{k/x\} = P$ .

2.  $x \neq k'$ . Then from (1) we obtain:

$$\Gamma; \Lambda; (\Sigma \setminus \{k' : \text{end}\}), x : S \vdash P : T \quad (4)$$

By the I.H. on (4) we have:

$$\Gamma; \Lambda; (\Sigma \setminus \{k' : \text{end}\}), k : S \vdash P\{k/x\} : T \quad (5)$$

We now consider two cases:

- (a)  $k' : \text{end} \in \Sigma$ . Then with an application of (Close) on (5) we obtain:

$$\Gamma; \Lambda; \Sigma, k : S \vdash P\{k/x\} : T$$

as required.

- (b)  $k' : \text{end} \notin \Sigma$ . Then  $\Sigma \setminus \{k' : \text{end}\} = \Sigma$  and the result is immediate from (5).

The remaining session cases are straightforward.  $\square$

**Lemma C.11** (Shared value judgement). *If  $\Gamma; \Lambda; \Sigma \vdash V : U$  and  $U \in \{\text{unit}, \langle S \rangle\}$  then  $\Lambda = \Sigma = \emptyset$ .*

**Proof.** Straightforward to show by induction on the typing derivation. There are two cases to consider: if  $U = \text{unit}$  then by (Unit) the result is immediate; if  $U = \langle S \rangle$  then by (Shared) the result follows. No other typing rule need to be considered for this type of value.  $\square$

**Lemma C.12.** *If  $\Sigma_1, \Sigma_2$  defined and  $\Sigma'_1 \leq_c \Sigma_1$  and  $\Sigma'_2 \leq_c \Sigma_2$  then  $\Sigma'_1, \Sigma'_2$  defined and  $\Sigma'_1, \Sigma'_2 \leq_c \Sigma_1, \Sigma_2$ .*

**Proof.** Trivial by the definition of  $\leq_c$  on environments and the fact that it does not change the domain of an environment.  $\square$

**Lemma C.13.** *If  $\Gamma; \Lambda, x : U; \Sigma \vdash P : T$  then  $x$  is free in  $P$ .*

**Proof.** Since there is no weakening for the  $\Lambda$  environment, the only way to introduce  $x$  in  $\Lambda, x : U$  is by applying the axiom (LVar) with subject  $x$ . But whenever a linear variable is bound, it is removed from the linear set of the conclusion; see (Abs) and (Rec). Hence  $x$  appears in  $P$  and is not bound.  $\square$

**Lemma C.14.** *If  $\Gamma; \Lambda; \Sigma, k : S \vdash P : T$  and  $k$  is not free in  $P$  then  $S = \text{end}$ .*

**Proof.** Since there is no weakening for the  $\Sigma$  environment, and  $k$  is not free in  $P$ , the only way to introduce a mapping for  $k$  in  $\Sigma, k : S$  is by applying the axiom (Nil). But then  $S = \text{end}$ .  $\square$

**Lemma C.15** (Environment properties).

1. *If  $\Delta \odot k :: \vec{\tau}_1$  defined then  $\Delta \odot k :: \vec{\tau}_2$  defined for any  $\vec{\tau}_1$  and  $\vec{\tau}_2$ .*
2. *If  $\Delta \odot k : S_1$  defined then  $\Delta \odot k : S_2$  defined for any  $S_1$  and  $S_2$ .*
3. *If  $\Delta, \Delta'$  defined then  $\Delta \odot \Delta'$  defined and  $\Delta, \Delta' = \Delta \odot \Delta'$ .*
4.  *$\Delta \odot \Delta' = \Delta' \odot \Delta$  and  $(\Delta_1 \odot \Delta_2) \odot \Delta_3 = \Delta_1 \odot (\Delta_2 \odot \Delta_3)$ .*
5. *If  $\Delta_1 \odot \Delta_2$  defined and  $\Delta_2 \sqsubseteq_s \Delta_3$  then  $\Delta_1 \odot \Delta_3$  defined.*
6. *If  $\text{balanced}(\Delta)$  and  $\Delta \sqsubseteq_s \Delta'$  then  $\text{balanced}(\Delta')$ .*

**Proof.** Straightforward from the definitions of  $\text{balanced}(\Delta)$ ,  $\odot$  and  $\sqsubseteq_s$ .  $\square$

**Lemma C.16.** *If  $\Gamma; \Lambda; \Sigma \odot \Delta \vdash P : T$  and  $\Sigma \leq_c \Sigma'$  then  $\Gamma; \Lambda; \Sigma' \odot \Delta \vdash P : T$ .*

**Proof.** Outline: For each  $k : S \in \Sigma$  with  $k : S' \in \Sigma'$ , and with  $P \equiv (\nu \vec{a} : (\vec{S}))(\nu \vec{s})(P_1 \mid \dots \mid P_n)$ , we take cases on the free occurrence of  $k$  in some  $P_i$ . If  $P_i$  is not a queue process then by (Sub) we obtain  $k : S'$  in the session environment of the subderivation for  $P_i$ . If  $P_i$  is a queue process then it is typed using (Queue) and we can apply (Sub) as before on the premises. Then using (New), (New<sub>s</sub>) and (Par) we obtain the required judgement.  $\square$

**Lemma C.17** (Queue subsumption). *If  $\Gamma; \Lambda; \Delta \odot k :: \vec{\tau}_1 U_1 \vec{\tau}_2 \vdash P : T$  and  $U_1 \leq_c U_2$  then  $\Gamma; \Lambda; \Delta \odot k :: \vec{\tau}_1 U_2 \vec{\tau}_2 \vdash P : T$ .*

**Proof.** We can easily show by induction that there is an application of (Queue) in the derivation. In the premises of this instance of (Queue) we can apply (Sub) on the typing judgement of the value that corresponds to the  $U_1$  typing, then re-apply (Queue) using the new premise with  $U_2$ .  $\square$

**Proof of Theorem 7.3 (Type Soundness).**

**Part (1).** Subject congruence is standard, except for the case of garbage collection. The latter is easy: first use the restricted weakening environment  $\Sigma_0$  of rule (Queue) to obtain, after  $\odot$ -composition, the balanced usage pairs  $(\text{end}, \epsilon)$  for the dual ended queues; then by (New<sub>s</sub>) the ended session can be restricted.

**Part (2).** For this part we proceed as standard by taking cases on the last reduction rule applied. For all cases we assume:

$$\Gamma; \emptyset; \Delta \vdash P : T \quad (\star) \qquad \text{balanced}(\Delta) \qquad P \longrightarrow P'$$

**Case** (beta)  $P = (\lambda(x:U).Q) V$   $P' = Q\{V/x\}$

From  $(\star)$  we have that the judgement for  $P$  has as last rule(s) a (possibly empty) sequence of applications of (Sub), and then (App). We then have by the judgement  $(\star)$  before (Sub) and the premises of (App) that:

$$\begin{array}{lll} \Gamma; \emptyset; \Sigma_1, \Sigma_2 \vdash P : T' & (1) & \Gamma; \emptyset; \Sigma_1 \vdash \lambda(x:U).Q : U \multimap T' & (2) & \Gamma; \emptyset; \Sigma_2 \vdash V : U & (3) \\ \Sigma_1, \Sigma_2 \leq_c \Delta & (4) & T' \leq_c T & (5) & \text{By (Lift), if } U = U_0 \rightarrow T_0 \text{ then } \Sigma_2 = \emptyset & (6) \end{array}$$

To obtain (2) we have, after a possibly empty sequence of (Sub), an application of (Abs) with:

$$(\Gamma; \Lambda; \Sigma'_1) \S x:U \vdash Q : T'' \quad (7)$$

$$T'' \leq_c T' \quad (8)$$

$$\Sigma'_1 \leq_c \Sigma_1 \quad (9)$$

By Lemma C.12, (4) and (9) and since  $\Sigma_1, \Sigma_2$  is defined we have that  $\Sigma'_1, \Sigma_2$  is defined and:

$$\Sigma'_1, \Sigma_2 \leq_c \Sigma_1, \Sigma_2 \quad (10)$$

We now proceed with case analysis on the type  $U$ , separating the cases into *shared*, *linear function*, and *session* types. In each case we determine if the (environment) conditions are met for the Substitution Lemma to apply. Then the result follows easily.

- (a)  $U \in \{\text{unit}, \langle S \rangle, U_0 \rightarrow T_0\}$ . Then (7) is of the shape  $\Gamma, x:U; \emptyset; \Sigma'_1 \vdash Q : T''$ . By (7) and (3) and by Lemma C.11 and (6) we can assert that  $\Sigma_2 = \emptyset$ , and therefore  $\Sigma'_1 \leq_c \Sigma_1 \leq_c \Delta$ . Then, using Lemma C.10(1) with (7) and (3), we obtain:

$$\Gamma; \emptyset; \Sigma'_1 \vdash Q \{V/x\} : T'' \quad (11)$$

Finally using (Sub) with (10) and (4) for the session environment and (8) and (5) for the result type, we obtain:

$$\Gamma; \emptyset; \Delta \vdash Q \{V/x\} : T$$

- (b)  $U = U_0 \multimap T_0$ . Then (7) is of the shape  $\Gamma; \{x:U\}; \Sigma'_1 \vdash Q : T''$ .

Then, using Lemma C.10(1) with (7) and (3), we obtain:

$$\Gamma; \emptyset; \Sigma'_1, \Sigma_2 \vdash Q \{V/x\} : T' \quad (12)$$

The environment  $\Sigma'_1, \Sigma_2$  is defined; see (10). Using (Sub) as before, noting as in the previous case that  $\Sigma'_1, \Sigma_2 \leq_c \Sigma_1, \Sigma_2 \leq_c \Delta$ , we obtain:

$$\Gamma; \emptyset; \Delta \vdash Q \{V/x\} : T$$

- (c)  $U = S$ . Then  $V = s$  or  $V = \bar{s}$ . Without loss of generality we fix the proof to use  $s$ . From (3), and (2) with (Abs), following similar steps as before, we obtain:

$$\Gamma; \emptyset; \Sigma_2 \vdash s : S \quad (13)$$

$$\Gamma; \emptyset; \Sigma'_1, x : S \vdash Q : T'' \quad (14)$$

From Lemma C.10(2) with (13) and (14) we obtain:

$$\Gamma; \emptyset; \Sigma'_1, s : S \vdash Q \{V/x\} : T'' \quad (15)$$

We have that  $\{s : S\} \leq_c \Sigma_2$ , then using also (9) with Lemma C.12 as before we obtain  $\Sigma'_1, s : S \leq_c \Sigma_1, \Sigma_2$ . Then using (4) and (8) and (5) with (Sub) on (15) we obtain

$$\Gamma; \emptyset; \Delta \vdash Q \{V/x\} : T \quad \square$$

**Case (send)**  $P = s!(V).Q \mid \bar{s}:\vec{h} \quad P' = Q \mid \bar{s}:\vec{h} \cdot V$

As before we fix the output to use  $s$  and the input to use  $\bar{s}$ . The last rule applied was (Par) for runtime. From this we have:

$$\Gamma; \emptyset; \Sigma_1 \vdash s!(V).Q : \diamond \quad (1)$$

$$\Gamma; \emptyset; \Sigma_2 \odot \bar{s} :: \vec{\tau} \vdash \bar{s}:\vec{h} : \diamond \quad (2)$$

$$\Delta = \Sigma_1 \odot \Sigma_2 \odot \bar{s} :: \vec{\tau} \quad (3)$$

After a possible application of (Sub) on (1), by (Send) and its premises:

$$\Gamma; \emptyset; \Sigma'_1 \vdash s!(V).Q : \diamond \quad (4)$$

$$\Sigma'_1 = (\Sigma_{11}, \Sigma_{12}) \setminus \{s:S\}, s:[U].S \quad (5)$$

$$\Gamma; \emptyset; \Sigma_{11} \vdash Q : \diamond \quad (6)$$

$$\Gamma; \emptyset; \Sigma_{12} \vdash V : U \quad (7)$$

$$s:S \in \Sigma_{1i} \quad i \in 1, 2 \quad (8)$$

$$\text{if } U = U_1 \rightarrow T_1 \text{ then } \Sigma_{12} = \emptyset \quad (9)$$

$$\Sigma'_1 \leq_c \Sigma_1 \quad (10)$$

Using (8) with  $\Sigma_{11}, \Sigma_{12} = \Sigma'_{11}, \Sigma'_{12}, s:S$ , we get, using (3), (10) and Lemma C.16:

$$\Sigma'_1 \odot \Sigma_2 \odot \bar{s} :: \vec{\tau} \quad \text{is defined}$$

$$(\Sigma_{11}, \Sigma_{12}) \setminus \{s:S\}, s:[U].S \odot \Sigma_2 \odot \bar{s} :: \vec{\tau} \quad \text{is defined}$$

$$(\Sigma'_{11}, \Sigma'_{12}, s:S) \setminus \{s:S\}, s:[U].S \odot \Sigma_2 \odot \bar{s} :: \vec{\tau} \quad \text{is defined}$$

$$(\Sigma'_{11}, \Sigma'_{12}, s:[U].S) \odot \Sigma_2 \odot \bar{s} :: \vec{\tau} \quad \text{is defined}$$

Then using Lemma C.15(3) on the above, followed by Lemma C.15(1–2), we obtain:

$$\Sigma'_{11} \odot \Sigma'_{12} \odot s:S \odot \Sigma_2 \odot \bar{s}::\bar{\tau}U \quad \text{is defined} \quad (11)$$

In (2) the last rule applied was (Queue) and combining the premises and adding (7) we obtain, by a new application of (Queue) (noting also (9) which is needed):

$$\Gamma; \emptyset; \Sigma_2 \odot \bar{s}::\bar{\tau}U \vdash \bar{s}:\bar{h} \cdot V : \diamond \quad (12)$$

where the session environment is defined by (11). Then using (6), (12), and (Par), we obtain:

$$\Gamma; \emptyset; \Sigma_{11} \odot \Sigma_2 \odot \Sigma'_{12} \odot \bar{s}::\bar{\tau}U \vdash P' : \diamond \quad (13)$$

By Lemma C.15(3) we have:

$$\Sigma_{11} \odot \Sigma_{12} = \Sigma_{11}, \Sigma_{12} = \Sigma'_{11}, \Sigma'_{12}, s:S = \Sigma'_{11} \odot \Sigma'_{12} \odot s:S$$

and (13) becomes:

$$\Gamma; \emptyset; \Sigma'_{11} \odot \Sigma'_{12} \odot s:S \odot \Sigma_2 \odot \bar{s}::\bar{\tau}U \vdash P' : \diamond \quad (14)$$

By Lemma C.16 since  $\Sigma'_{11} \odot \Sigma'_{12} \leq_c \Sigma_1 \setminus s$ , we have from (14) that:

$$\Gamma; \emptyset; (\Sigma_1 \setminus s) \odot s:S \odot \Sigma_2 \odot \bar{s}::\bar{\tau}U \vdash P' : \diamond \quad (15)$$

From (10) we have  $s:![U].S \in \Sigma'_1$  and by (5) there is  $s:S' \in \Sigma_1$  with  $![U].S \leq_c S'$ . By the definition of simulation  $\text{unfold}^n(S') = \mathcal{A}(![U'].S'_h)^{h \in H}$  and  $U \leq_c U'$  and  $S \leq_c \mathcal{A}(S'_h)^{h \in H}$ . Finally using Lemma C.16 and Lemma C.17 on (15) we can obtain:

$$\Gamma; \emptyset; \Delta' \vdash P' : \diamond$$

where  $\Delta' = (\Sigma_1 \setminus s) \odot s:\mathcal{A}(S'_h)^{h \in H} \odot \Sigma_2 \odot \bar{s}::\bar{\tau}U'$  and it holds that  $\Delta \sqsubseteq_s \Delta'$ .

Case (recv) is very similar to (beta); the rest are easy to obtain.  $\square$

## References

- [1] S. Abramsky, Computational interpretations of linear logic, *Theor. Comput. Sci.* 111 (1993).
- [2] R.M. Amadio, L. Cardelli, Subtyping recursive types, *ACM Trans. Program. Lang. Syst.* 15 (1993) 575–631.
- [3] G. Bernardi, M. Hennessy, Using higher-order contracts to model session types, *CoRR* arXiv:1310.6176, 2013.
- [4] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: *CONCUR*, in: LNCS, vol. 5201, Springer-Verlag, 2008, pp. 418–433.
- [5] E. Bonelli, A. Compagnoni, Multipoint session types for a distributed calculus, in: *TGC'07*, in: LNCS, vol. 4912, Springer-Verlag, 2008, pp. 240–256.
- [6] V. Bono, L. Padovani, A. Tosatto, Polymorphic types for leak detection in a session-oriented functional language, in: *FMOODS*, in: LNCS, vol. 7892, Springer, 2013, pp. 83–98.
- [7] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: *CONCUR'10*, 2010, pp. 222–236.
- [8] M. Carbone, K. Honda, N. Yoshida, Structured communication-centred programming for web services, in: *ESOP'07*, in: LNCS, vol. 4421, Springer-Verlag, 2007, pp. 2–17.
- [9] T.C. Chen, M. Dezani-Ciancaglini, N. Yoshida, On the preciseness of subtyping in session types, in: *Proceedings of PPDP'14*, ACM Press, 2014, pp. 135–146.
- [10] O. Dardha, E. Giachino, D. Sangiorgi, Session types revisited, in: D.D. Schreye, G. Janssens, A. King (Eds.), *PPDP*, ACM, 2012, pp. 139–150.
- [11] R. Demangeon, K. Honda, Full abstraction in a subtyped pi-calculus with linear types, in: *CONCUR*, in: LNCS, vol. 6901, Springer-Verlag, 2011, pp. 280–296.
- [12] P.M. Denilou, N. Yoshida, Multiparty session types meet communicating automata, in: *ESOP 2012*, in: LNCS, vol. 7211, 2012, pp. 194–213.
- [13] P.M. Denilou, N. Yoshida, Multiparty compatibility in communicating automata: characterisation and synthesis of global session types, in: *Automata, Languages, and Programming*, in: LNCS, vol. 7966, 2013, pp. 174–186.
- [14] H. DeYoung, L. Caires, F. Pfenning, B. Toninho, Cut reduction in linear logic as asynchronous session-typed communication, in: P. Cégielski, A. Durand (Eds.), *CSL*, in: *LIPICs*, vol. 16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012, pp. 228–242.
- [15] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, N. Yoshida, Objects and session types, *Inf. Comput.* 207 (2009) 595–641.
- [16] M. Dezani-Ciancaglini, U. de'Liguoro, Sessions and session types: an overview, in: *WS-FM'09*, in: LNCS, vol. 6194, Springer, 2010, pp. 1–28.
- [17] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, S. Drossopoulou, Session types for object-oriented languages, in: *ECOOP'06*, in: LNCS, vol. 4067, Springer-Verlag, 2006, pp. 328–352.
- [18] V. Gapeyev, M. Levin, B. Pierce, Recursive subtyping revealed, *J. Funct. Program.* 12 (2003) 511–548.
- [19] S. Gay, M. Hole, Subtyping for session types in the pi-calculus, *Acta Inform.* 42 (2005) 191–225.
- [20] S. Gay, V.T. Vasconcelos, Linear type theory for asynchronous session types, *J. Funct. Program.* 20 (2010) 19–50.
- [21] J.Y. Girard, Linear logic, *Theor. Comput. Sci.* 50 (1987).
- [22] M. Giunti, V.T. Vasconcelos, Linearity, session types and the pi calculus, *Math. Struct. Comput. Sci.* (2014).
- [23] Web services choreography working group, Web services choreography description language, <http://www.w3.org/2002/ws/chor/>, 2002.
- [24] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: *ESOP'98*, in: LNCS, vol. 1381, 1998, pp. 22–138.
- [25] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: *POPL'08*, ACM, 2008, pp. 273–284.
- [26] R. Hu, N. Yoshida, K. Honda, Session-based distributed programming in Java, in: *ECOOP'08*, in: LNCS, vol. 5142, 2008, pp. 516–541.
- [27] K. Imai, S. Yuen, K. Agusa, Session type inference in Haskell, in: K. Honda, A. Mycroft (Eds.), *PLACES*, in: *EPTCS*, vol. 69, 2010, pp. 74–91.

- [28] D. Kouzapas, N. Yoshida, R. Hu, K. Honda, On asynchronous eventful session semantics, *Math. Struct. Comput. Sci.* 29 (5) (2014) 1–62.
- [29] C. Laneve, B. Victor, Solos in concert, *Math. Struct. Comput. Sci.* 13 (2003) 657–683.
- [30] R. Milner, Functions as processes, *Math. Struct. Comput. Sci.* 2 (1992) 119–141.
- [31] R. Milner, *Communicating and Mobile Systems: The  $\pi$ -Calculus*, CUP, 1999.
- [32] D. Mostrous, Session types in concurrent calculi: higher-order processes and objects, PhD thesis, Imperial College London, 2009.
- [33] D. Mostrous, Multiparty sessions based on proof nets, in: A.F. Donaldson, V.T. Vasconcelos (Eds.), *Proceedings PLACES*, 2014.
- [34] D. Mostrous, V.T. Vasconcelos, Session typing for a featherweight Erlang, in: W.D. Meuter, G.C. Roman (Eds.), *COORDINATION*, in: LNCS, vol. 6721, Springer, 2011, pp. 95–109.
- [35] D. Mostrous, N. Yoshida, Two session typing systems for higher-order mobile processes, in: *TLCA'07*, in: LNCS, vol. 4583, Springer-Verlag, 2007, pp. 321–335.
- [36] D. Mostrous, N. Yoshida, Session-based communication optimisation for higher-order mobile processes, in: *TLCA'09*, in: LNCS, vol. 5608, Springer-Verlag, 2009, pp. 203–218.
- [37] D. Mostrous, N. Yoshida, K. Honda, Global principal typing in partially commutative asynchronous sessions, in: *ESOP'09*, in: LNCS, vol. 5502, Springer-Verlag, 2009, pp. 316–332.
- [38] MPI, The Message Passing Interface (MPI) standard, <http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/intermediate/main.htm>, 2013.
- [39] M. Neubauer, P. Thiemann, An implementation of session types, in: *PADL*, in: LNCS, vol. 3057, Springer-Verlag, 2004, pp. 56–70.
- [40] M. Neubauer, P. Thiemann, Session types for asynchronous communication, 2004, unpublished manuscript.
- [41] N. Ng, N. Yoshida, K. Honda, Multiparty session C: safe parallel programming with message optimisation, in: *TOOLS*, in: LNCS, vol. 7304, Springer, 2012, pp. 202–218.
- [42] J.A. Pérez, L. Caires, F. Pfenning, B. Toninho, Linear logical relations for session-based concurrency, in: H. Seidl (Ed.), *ESOP*, in: LNCS, vol. 7211, Springer, 2012, pp. 539–558.
- [43] B.C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
- [44] B.C. Pierce, D. Sangiorgi, Typing and subtyping for mobile processes, *Math. Struct. Comput. Sci.* 6 (1996) 409–453.
- [45] R. Pucella, J.A. Tov, Haskell session types with (almost) no class, in: *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell'08, ACM, New York, NY, USA, 2008, pp. 25–36.
- [46] D. Sangiorgi, Expressing mobility in process algebras: first-order and higher order paradigms, Ph.D. thesis, University of Edinburgh, 1992.
- [47] SCRIBBLE, Scribble Project, [www.scribble.org](http://www.scribble.org), 2013.
- [48] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: *PARLE'94*, in: LNCS, vol. 817, Springer-Verlag, 1994, pp. 398–413.
- [49] B. Toninho, L. Caires, F. Pfenning, Higher-order processes, functions, and sessions: a monadic integration, in: M. Felleisen, P. Gardner (Eds.), *Programming Languages and Systems*, in: LNCS, vol. 7792, Springer, Berlin, Heidelberg, 2013, pp. 350–369.
- [50] UNIFI, International Organization for Standardization ISO 20022 UNiversal Financial Industry message scheme, <http://www.iso20022.org>, 2002.
- [51] V.T. Vasconcelos, A Note on a Typing System for the Higher-Order  $\pi$ -Calculus, Keio University, 1993.
- [52] V.T. Vasconcelos, S. Gay, A. Ravara, Typechecking a multithreaded functional language with session types, *Theor. Comput. Sci.* 368 (2006) 64–87.
- [53] D. Walker, in: Benjamin C. Pierce (Ed.), *Advanced Topics in Types and Programming Languages*, MIT, 2005.
- [54] N. Yoshida, V.T. Vasconcelos, Language primitives and type disciplines for structured communication-based programming revisited, *Electron. Notes Theor. Comput. Sci.* 171 (2007) 127–151.