Under consideration for publication in Math. Struct. in Comp. Science

Linearity, Session Types and the Pi Calculus

Marco Giunti[1] and Vasco Thudichum Vasconcelos[2]

[1] *CITI, Faculty of Sciences and Technology, New University of Lisbon*
[2] *LaSIGE, Faculty of Sciences, University of Lisbon*

Received 22 April 2013

We present a type system based on session types that works on a conventional pi calculus. Types are equipped with a constructor that describes the two ends of a single communication channel, this being the only type available for describing the behaviour of channels. Session types, in turn, describe the behaviour of each individual channel end, as usual. A novel notion of typing context split allows for typing processes not typable with extant type systems. We show that our system guarantees that typed processes do not engage in races for linear resources. We assess the expressiveness of type system by providing three distinct encodings—from the pi calculus with polarised variables, from the pi calculus with accept and request primitives, and from the linear pi calculus—into our system. For each language we present operational and typing correspondences, showing that our system effectively subsumes foregoing works on linear and session types. In the case of the linear pi calculus we also provide a completeness result.

1. Introduction

The pi calculus is a formalism for studying concurrency in programming languages. Introduced in the late eighties as a process algebra (Milner et al., 1992), its has seen multiple developments tho these days, including type systems that govern the behaviour of processes, the topic of this paper. One such class of type systems uses session types. Session types allow a concise description of protocols by detailing the patterns of the messages exchanged in each particular run of the protocol. They were firstly introduced for a dialect of the pi calculus (Honda et al., 1998; Takeuchi et al., 1994). Later the concept has been transferred to different realms, including functional (Gay and Vasconcelos, 2010; Vasconcelos et al., 2006) and object-oriented programming (Capecchi et al., 2009; Coppo et al., 2007; Dezani-Ciancaglini et al., 2005; Dezani-Ciancaglini et al., 2007; Dezani-Ciancaglini et al., 2006; Gay et al., 2010; Hu et al., 2008), service oriented computing (Cruz-Filipe et al., 2008; Bruni and Mezzina, 2008) and operating systems (Fähndrich et al., 2006), to name a few; the reader is referred to (Dezani-Ciancaglini and de'Liguoro, 2010) for a recent overview.

In this paper we concentrate on a subset of the pi calculus, as formulated in (Milner, 1992), and on *binary* session types. Binary session types describe the interaction of exactly two partners (or processes) exchanging messages on a bi-directional communication channel. For example, an interaction on which one partner sends a string, receives

a boolean value, replies with an integer and finally terminates the interaction, can be written as !string.?bool.!int.end. In order for the interaction to proceed as planned, the second partner must first receive a string, then emit a boolean value, and finally receive an integer before terminating the interaction. Such an interaction is captured by the type ?string.!bool.?int.end, a type that is dual to the one above.

The discussion leads to the conclusion that one single communication channel is endowed with two types, each governing the interaction on one of its two ends. The pi calculus does not distinguish the two ends of a single channel. In order to capture within a single type the capabilities of both ends of a channel, our types are pairs (S_1, S_2) where S_1 describes the behaviour of one end and S_2 the behaviour of the other.

We have seen that session types capture patterns of interactions happening on channels. For such interactions to occur free of interference, the channel must be shared by exactly two processes. This means that the type that governs the channel's behaviour must be linear, and in particular that it cannot be duplicated or discarded. In a world of linear types only there is no interference, hence no races. This also means that there cannot be competition for (shared) resources, thus greatly reducing the class of interesting programs one can write. In addition to linear channels our type system also accounts for shared (unrestricted, in the terminology of this paper) channels that govern interactions with shared resources. There are at least two ways one can understand the linear/unrestricted distinction on channels and on session types. One view classifies a channel as a whole, so that a channel that was created linear will continue as such during its lifetime. The other view classifies each interaction point within a protocol individually, so that a channel may, during its lifetime, behave as linear for a number of interactions and became unrestricted in its later life. We follow the latter approach for enhanced flexibility (Vasconcelos, 2012).

Processes are typed against contexts formed of typing assumptions on variables. Central to linear type systems is the notion of context splitting. In process algebras it is used to handle parallel composition and message exchange (input and output). For example, in order to type the parallel composition of two processes under a given context, we split the context in two parts, and use each part to type each process. The traditional definition of context splitting sends incoming shared types both ways, and linear types either way but not both, in order not to duplicate or discard linear information. This is also the case with our type system. But we go further in this respect.

We have said that channel types are pairs, simultaneously describing the capabilities of the two ends of the channel. When splitting a context containing a linear type it is usually necessary to send one capability to the left, and the other to the right. For example, imagine that S_1 in type (S_1, S_2) describes the input capability of a channel, whereas S_2 the output capability, and that both are linear. The question arises as to which (pair) type do we send to the left context. Once you have given away the output capability of a channel end what remains is the capability of performing no further input/output operations on that particular channel end, a capability which we denote by end. The context splitting relation we work with allows to split the above type into (S_1, end) and (end, S_2). It also allows to split the same type into (S_1, S_2) and $(\mathsf{end}, \mathsf{end})$, so that the process that uses that channel at type $(\mathsf{end}, \mathsf{end})$ cannot perform any interaction

on the channel. It may however use it for purposes other than interaction (e.g., testing for equality). The contribution of this paper is a novel type system. Such a system:

— works directly on the pi-calculus (Milner, 1992), as opposed to variants usually found in the literature of session types (Honda et al., 1998; Gay and Hole, 2005; Vasconcelos, 2012),
— types processes not accepted by existing systems (Gay and Hole, 2005; Giunti and Vasconcelos, 2010; Honda et al., 1998; Kobayashi et al., 1999; Vasconcelos, 2012),
— guarantees that typed processes do not get stuck at run-time and do not engage in races for linear resources (Theorem 3.1),
— subsumes previous type systems for the pi calculus, including the pi calculus with polarities and session types (Gay and Hole, 2005) (hence the conventional pi calculus), the original version of session types (Honda et al., 1998), and the linear pi calculus (Kobayashi et al., 1999).

The outline of the paper is as follows. The next section discusses the related work, and puts into perspective the choices we have made. Section 3 briefly recalls the syntax and operational semantics of the pi calculus, introduces our type system and its main result, whose proof is then outlined in Section 4. Section 5 tests the flexibility of our type system by embedding three systems. For each of these languages we prove an operational and a typing correspondence result. For the linear pi calculus we further provide a completeness result. Section 6 concludes the paper.

2. Related work

Since the proposal of session types (Honda, 1993; Honda et al., 1998; Takeuchi et al., 1994) (discussed in Section 5.2) a few variants were put forward, including a system for a pi calculus with polarities (Gay and Hole, 2005) (discussed in Section 5.1), a system for a pi calculus with a double binder for ν-processes and lin/un qualified session types (Vasconcelos, 2012), and a system for the conventional pi calculus, where the two input/output capabilities of a channel may be passed in a single message (Giunti and Vasconcelos, 2010). We also include in our comparison the linear pi-calculus (Kobayashi et al., 1999) (discussed in Section 5.3), since linear types can be seen as a degenerated form of session types, where a single message is exchanged. We try to systematise the relations between these type systems according to the four criteria below.

The two ends of a single channel:
- are syntactically indistinguishable (Giunti and Vasconcelos, 2010; Honda et al., 1998; Kobayashi et al., 1999); (This work);
- have distinct syntax (Gay and Hole, 2005; Vasconcelos, 2012).

Types describe:
- channel end-points (Gay and Hole, 2005; Giunti and Vasconcelos, 2010; Honda et al., 1998; Kobayashi et al., 1999; Vasconcelos, 2012);
- channels, both ends simultaneously (Kobayashi et al., 1999; Giunti and Vasconcelos, 2010); (This work).

The shared/linear distinction pertains to:	types as a whole (Gay and Hole, 2005; Honda et al., 1998; Kobayashi et al., 1999);
	communication points within a type (Giunti and Vasconcelos, 2010; Vasconcelos, 2012); (This work).
When a context is split:	unrestricted (end-point or channel) types go both ways and linear (end-point or channel) types go either way (Gay and Hole, 2005; Giunti and Vasconcelos, 2010; Honda et al., 1998; Vasconcelos, 2012); (This work);
	linear channel types go both ways, split as linear end-point types (Kobayashi et al., 1999; Giunti and Vasconcelos, 2010) or placed in a pair type with an end component (This work);
	unrestricted channel types go both ways, split as unrestricted end-point types (Kobayashi et al., 1999) or placed in a pair type type with an end component (This work).

We now present a few examples that discriminate among these type systems. We write $\overline{c}\,\mathsf{true}.P$ for the process that writes true on channel c and continues as P; $c(x).P$ for the process that reads a value from c, binds it to x, and continues as P; $\mathbf{0}$ for the terminated process (often omitted in the continuations); and $P \mid Q$ for the parallel composition of processes P and Q. All processes below reduce in two or three steps to $\mathbf{0}$. Process

$$\overline{c}\,\mathsf{true}.(\overline{c}\,3 \mid \overline{c}\,5) \mid c(x).c(y).c(z)$$

requires c to be typed with a linear session type (in order to type $\overline{c}\,\mathsf{true}$ followed by $\overline{c}\,3$) that later becomes unrestricted (in order to type the two messages $\overline{c}\,3$ and $\overline{c}\,5$ in parallel). It is typable in (Vasconcelos, 2012) (with the necessary syntactical adjustments), but not in (Honda et al., 1998), (Gay and Hole, 2005), or (Kobayashi et al., 1999). Process

$$\overline{c}\,d \mid c(x).(\overline{x}\,\mathsf{true} \mid x(y))$$

is typable in (Giunti and Vasconcelos, 2010) or any other system mentioned above, but not in (Vasconcelos, 2012) since the two input/output capabilities of a channel (d) are passed at the same time (on channel c). Notice that the second thread needs both the input and the output capability of d in order to exchange the true message. Process

$$\overline{c}\,\mathsf{true}.\overline{c}\,3 \mid c(x).c(y) \mid \overline{d}\,c \mid d(z)$$

is typable in the system proposed in this paper but not in (Giunti and Vasconcelos, 2010), since channel c occurs in three parallel threads. Notice that the channel occurs in subject position[†] in two threads only, and that the last occurrence of c entails no communication (channel c is inactive, it can only be passed around). Finally, process

$$\overline{c}\,\mathsf{true} \mid c(x).\mathsf{if}\,x\,\mathsf{then}\,(\overline{c}\,5 \mid c(y))\,\mathsf{else}\,\mathbf{0}$$

is not typable in the system proposed in this paper. To see why notice that the session type of channel c in the second thread must be linear, for it receives a boolean value

[†] A channel c occurs in subject position in processes of the form $\overline{c}\,v.P$ and $c(x).P$.

(forced by the conditional) and then exchanges an integer message. Notice also that the second thread needs both the input and the output capability of channel c in order to exchange the integer message. But the first parallel composition forces the input/output capabilities of channel c to be split between the two threads.

The ability to pass part of the functionality of a linear channel and retain the rest goes beyond what is usually available to session typing systems. In the realm of binary session types, it was first proposed by the authors (Giunti et al., 2009). The method is however to be found in several works in the literature; we mention two. Multiparty session types use a projection operator to extract session types from the global type of a system (Honda et al., 2008). The conversation calculus includes a type splitting relation allowing to decompose the type of processes (Baltazar et al., 2013; Caires and Vieira, 2010). In comparison with these systems, the splitting capabilities of our types are quite limited: from a linear (end point) type all we can do is extract an end type while keeping the original type. Nevertheless, our splitting relation allows typing processes hitherto not typable.

(Padovani, 2012) describes the behaviour of processes according to the channels they use. Session types appear as the restriction of such descriptions with respect to a single channel. To describe the simultaneous access to a channel by more than one process, types include a form of parallel composition. Parallel composition of (session) types is somewhat related to our pair types. The type system guarantees that the two ends of a session type are always owned by independent processes, contributing towards a progress result which is not among the aims of the present work.

3. The type system

This section reviews the syntax and the semantics of the pi calculus, introduces our type system and an extended example, and concludes with a discussion of main result.

3.1. *Syntax and operational semantics*

The syntax of the pi calculus extended with boolean constants and conditional processes is in Figure 1. We rely on a *countable* set of variables, ranged over by x, y, z. Values include variables and the booleans true and false. For processes we have (synchronous, unary) output and input, in the forms $\overline{x}\,v.P$ and $x(y).P$, as well as parallel composition, conditional, scope restriction, replication and the terminated process. Except for the boolean values and the conditional, our language is exactly that of (Milner, 1992). The extensions are not strictly necessary for the development of the theory; they merely show how non-channel values can be incorporated in the language.

The binders for the language appear in parenthesis: x is bound in both $y(x).P$ and $(\nu x)P$. Free and bound variables in processes are defined accordingly, and so is alpha conversion, and substitution of a variable x by a value v in a process P, denoted $P[v/x]$. Notice that substitution is not a total function; it is not defined, e.g., for $(\overline{y}\,\mathsf{false})[\mathsf{true}/y]$. When writing $P[v/x]$ we assume that the substitution operation involved is defined. The set of free variables in a process P is denoted by $\mathrm{fv}(P)$. We follow Barendregt's

Syntax

v ::=		Values:	$x(x).P$	input
	true	true	$P \mid P$	composition
	false	false	$\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, P$	conditional
	x	variable	$(\nu x)P$	restriction
P ::=		Processes:	$!P$	replication
	$\overline{x}\, v.P$	output	$\mathbf{0}$	inaction

$\boxed{P \equiv P}$ *Rules for structural congruence*

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad !P \equiv P \mid !P$$
$$(\nu x)P \mid Q \equiv (\nu x)(P \mid Q) \qquad (\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$$

$\boxed{P \to P}$ *Rules for reduction*

$$\overline{x}\, v.P \mid x(y).Q \to P \mid Q[v/y] \qquad \text{[R-Com]}$$

$$\mathsf{if\,true\,then}\, P\, \mathsf{else}\, Q \to P \qquad \mathsf{if\,false\,then}\, P\, \mathsf{else}\, Q \to Q \qquad \text{[R-IfT] [R-IfF]}$$

$$\frac{P \to Q}{(\nu x)P \to (\nu x)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

[R-Res] [R-Par] [R-Struct]

Fig. 1. Pi calculus: Syntax and operational semantics

variable convention, requiring bound variables to be distinct from free variables in any mathematical context.

Structural congruence is the smallest relation on processes including the rules in Figure 1. The first three rules say that parallel composition is commutative, associative and has $\mathbf{0}$ for neutral element. The last rule on the first line captures the essence of replication as an unbounded number of identical processes. The rules in the second line deal with scope restriction. The first, scope extrusion, allows the scope of x to encompass Q; due to variable convention, x bound in $(\nu x)P$, cannot be free in Q. The other two rules state that restricting a terminated process has no effect and that the order of restrictions is irrelevant.

The reduction is the smallest binary relation on processes that includes the rules in Figure 1. The [R-Com] rule communicates value v from an output prefixed process $\overline{x}\, v.P$ to an input prefix $x(y).Q$; the result is the parallel composition of the continuation processes, where, in the input process, the bound variable y is replaced by value v. The rules for the conditional are straightforward. The rules in the last line allow reduction to happen underneath scope restriction and parallel composition, and incorporate structural congruence into reduction.

3.2. *Type checking*

The syntax of types is described in Figure 2. Types include the boolean type and channel types. The novelty with respect linear and session-based systems for the pi calculus is

q	$::=$	Qualifiers:		a	type variable
	lin	linear		$\mu a.S$	recursive end point
	un	unrestricted	T	$::=$	Types:
p	$::=$	Pre end point types:		bool	boolean
	$?T.S$	receive		(S, S)	channel
	$!T.S$	send	Γ	$::=$	Contexts:
S	$::=$	End point types:		$\emptyset$	empty context
	$q\ p$	qualified end point		$\Gamma, x\colon T$	variable binding
	end	used end point			

Fig. 2. Pi calculus: Types and typing contexts

the introduction of a new type constructor to describe the two ends of a same channel, (S_1, S_2), where S_1 details the behaviour of one end, whereas S_2 details that of the other end. An end point type S can be a pre type qualified with lin or un, the end type, a recursive type or a type variable. Each qualifier in a type controls the number of times the channel can be used at a given point: exactly once for lin; zero or more times for un. A pre type of the form $!T.S$ describes a channel end able to send a value of type T and to proceed as prescribed by S. Similarly, pre type $?T.S$ describes a channel end able to receive a value of type T and continue as S. End point type end describes a channel end on which no further interaction is possible. For recursive (end point) types we rely on a set of type variables, ranged over by a. Recursive types are required to be contractive, that is, containing no sub-expression of the form $\mu a_1 \ldots \mu a_n.a_1$. We will use the notation $*?T$ and $*!T$ to indicate respectively the types $\mu a.\mathsf{un}?T.a$ and $\mu b.\mathsf{un}!T.b$, where we assume a, b not occurring in T.

End point type equality is not syntactic. Instead, we define it as the equality of regular infinite trees obtained by the infinite unfolding of recursive types. The formal definition, which we omit, is co-inductive. In this way we use end point types $\mu a.\mathsf{lin}!\mathsf{bool}.\mathsf{lin}?\mathsf{bool}.a$ and $\mathsf{lin}!\mathsf{bool}.\mu b.\mathsf{lin}?\mathsf{bool}.\mathsf{lin}!\mathsf{bool}.b$ interchangeably, in any mathematical context. This allows us never to consider a type $\mu a.S$ explicitly (or a for that matter). Instead, we pick another type in the same equivalence class, namely $S[\mu a.S/a]$. If the result of the process turns out to start with a μ, we repeat the procedure. The procedure is bound to terminate due to contractiveness. In other words, we take an equi-recursive view of types, see (Pierce, 2002).

Type duality plays a central role in the theory of session types, ensuring that communication between the two ends of a channel proceeds smoothly. Intuitively, the dual of output is input and the dual of input is output. In particular if S_2 is dual of S_1, then $q?T.S_1$ is dual of $q!T.S_2$. End point type end is dual of itself. Rather than providing a co-inductive definition of duality, we start by defining a function $\overline{S}$ from end-point channels into end-point channels as follows.

$$\overline{q?T.S} = q!T.\overline{S} \qquad \overline{q!T.S} = q?T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{\mu a.S} = \mu a.\overline{S} \qquad \overline{a} = a$$

Then, to check whether a given end point type S_1 is dual of another type S_2, we first build the dual of S_1 and then check that the thus obtained type is equivalent to S_2. For example, to show that end point type $\mu a.\mathsf{lin}?\mathsf{bool}.\mathsf{lin}!\mathsf{bool}.a$ is a dual of type $\mathsf{lin}!\mathsf{bool}.\mu b.\mathsf{lin}?\mathsf{bool}.\mathsf{lin}!\mathsf{bool}.b$, we build $\overline{\mu a.\mathsf{lin}?\mathsf{bool}.\mathsf{lin}!\mathsf{bool}.a} = \mu a.\mathsf{lin}!\mathsf{bool}.\mathsf{lin}?\mathsf{bool}.a$, and then show that $\mu a.\mathsf{lin}!\mathsf{bool}.\mathsf{lin}?\mathsf{bool}.a = \mathsf{lin}!\mathsf{bool}.\mu b.\mathsf{lin}?\mathsf{bool}.!\mathsf{bool}.b$. Qualifiers are important: S and $\overline{S}$ must be equally qualified so that a linear output process may find a linear input process to embark in reduction.

Contexts, or typing environments, are defined in Figure 2. In a context $\Gamma, x\colon T$ we assume that x does not occur in Γ; we also assume that the various variable bindings in Γ are unordered.

We define the un predicate over end point types, types and contexts. For end point types we have that $\mathsf{un}(S)$ holds in the following cases,

$$\mathsf{un}(\mathsf{end}) \qquad \mathsf{un}(\mathsf{un}\, p) \qquad \mathsf{un}(\mu a.S) \text{ if } \mathsf{un}(S)$$

and for types we set $\mathsf{un}(T)$ as follows,

$$\mathsf{un}(\mathsf{bool}) \qquad \mathsf{un}((S_1, S_2)) \text{ if } \mathsf{un}(S_1), \mathsf{un}(S_2).$$

The predicate is then extended point-wise to contexts, so that $\mathsf{un}(\Gamma)$ if and only if $\mathsf{un}(T)$ for all $x\colon T$ in Γ. We say that a type T is *unrestricted* (or shared) when $\mathsf{un}(T)$, and similarly for an end point S. We say that an end point S is *linear* if S is not unrestricted, that is, if S is equal to a type of the form $\mathsf{lin}\, p$.

Typing relies on the splitting relation described in Figure 3. Any end point type S can be split in two: S itself and end, in either order. Unrestricted end point types S, on the other hand, may be split into two copies of S. In this way we make sure that linear end point types are never duplicated or eliminated, but we may loose information about unrestricted end points. For types, bool, as an unrestricted type according to the definition above, is split into two identical copies. Channel types are split in two by splitting the two end point types they are formed of. Finally, context splitting is defined inductively, based on type splitting. The two rules for non-empty contexts allow to "send" the first component of a linear end point type to the left context, or to the right context. We often write $T_1 \circ T_2$ for some type T such that $T = T_1 \circ T_2$, if there is such a triple (T, T_1, T_2) in the splitting relation, and similarly for end point types S and contexts Γ.

Below we show some examples of type splitting, where L, L_1, L_2 represent linear types and U, U_1, U_2 unrestricted types.

$$\begin{aligned}
(U_1, U_2) &= (U_1, U_2) \circ (U_1, U_2) & \qquad (U_1, U_2) &= (\mathsf{end}, U_2) \circ (U_1, \mathsf{end}) \\
(L_1, L_2) &= (\mathsf{end}, \mathsf{end}) \circ (L_1, L_2) & \qquad (L_1, L_2) &= (L_1, \mathsf{end}) \circ (\mathsf{end}, L_2) \\
(L, U) &= (L, U) \circ (\mathsf{end}, U)
\end{aligned}$$

What we do *not* have is duplication or elimination of linear types,

$$\begin{aligned}
(L_1, L_2) &\neq (L_1, L_2) \circ (L_1, L_2) \\
(L_1, L_2) &\neq (L_1, \mathsf{end}) \circ (\mathsf{end}, \mathsf{end})
\end{aligned}$$

$\boxed{S = S \circ S}$ *End point type splitting rules*

$$S = S \circ \mathsf{end} \qquad S = \mathsf{end} \circ S \qquad \mathsf{un}\, p = \mathsf{un}\, p \circ \mathsf{un}\, p$$

$\boxed{T = T \circ T}$ *Type splitting rules*

$$\mathsf{bool} = \mathsf{bool} \circ \mathsf{bool} \qquad \frac{R = R_1 \circ R_2 \qquad S = S_1 \circ S_2}{(R, S) = (R_1, S_1) \circ (R_2, S_2)}$$

$\boxed{\Gamma = \Gamma \circ \Gamma}$ *Context splitting rules*

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = T_1 \circ T_2}{\Gamma, x\colon T = (\Gamma_1, x\colon T_1) \circ (\Gamma_2, x\colon T_2)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = T_2 \circ T_1}{\Gamma, x\colon T = (\Gamma_1, x\colon T_1) \circ (\Gamma_2, x\colon T_2)}$$

$\boxed{\Gamma + (x\colon T) = \Gamma}$ *Context update rule*

$$(\Gamma, x\colon T_1) + (x\colon T_2) = \Gamma, x\colon T_1 \circ T_2$$

$\boxed{\Gamma \vdash v\colon T}$ *Typing rules for values*

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{true}\colon \mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{false}\colon \mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad \text{[T-True] [T-False] [T-Var]}$$

$\boxed{\Gamma \vdash P}$ *Typing rules for processes*

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash !P} \qquad \text{[T-Inact] [T-Par] [T-Repl]}$$

$$\frac{\Gamma_1 \vdash v\colon \mathsf{bool} \qquad \Gamma_2 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2} \qquad \frac{\Gamma, x\colon (S, \overline{S}) \vdash P}{\Gamma \vdash (\nu x)P} \qquad \text{[T-If] [T-Res]}$$

$$\frac{\Gamma_1 \vdash x\colon (q?T.S, S') \qquad (\Gamma_2 + x\colon (S, S')), y\colon T \vdash P \qquad q = \mathsf{un} \Rightarrow q?T.S = S}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P} \qquad \text{[T-InL]}$$

$$\frac{\Gamma_1 \vdash x\colon (S', q?T.S) \qquad (\Gamma_2 + x\colon (S', S)), y\colon T \vdash P \qquad q = \mathsf{un} \Rightarrow q?T.S = S}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P} \qquad \text{[T-InR]}$$

$$\frac{\Gamma_1 \vdash x\colon (q\,!T.S, S') \quad \Gamma_2 \vdash v\colon T \quad \Gamma_3 + x\colon (S, S') \vdash P \quad q = \mathsf{un} \Rightarrow q!T.S = S}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}\, v.P} \qquad \text{[T-OutL]}$$

$$\frac{\Gamma_1 \vdash x\colon (S', q\,!T.S) \quad \Gamma_2 \vdash v\colon T \quad \Gamma_3 + x\colon (S', S) \vdash P \quad q = \mathsf{un} \Rightarrow q!T.S = S}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}\, v.P} \qquad \text{[T-OutR]}$$

Fig. 3. Pi calculus: Type checking

or the complete elimination of types,

$$T \neq (\mathsf{end}, \mathsf{end}) \circ (\mathsf{end}, \mathsf{end}) \quad \text{when } T \neq (\mathsf{end}, \mathsf{end}).$$

We also need an operation to update a given context $\Gamma, x\colon T_1$ with a context entry $x\colon T_2$, yielding a context where x is associated with the type T such that $T = T_1 \circ T_2$. The operation is partial, given the nature of type splitting. Some examples: $\Gamma, x\colon (\mathsf{end}, U) + (L, U) = \Gamma, x\colon (L, U)$, and $\Gamma, x\colon (L_1, \mathsf{end}) + (\mathsf{end}, L_2) = \Gamma, x\colon (L_1, L_2)$, and $\Gamma, x\colon (\mathsf{end}, \mathsf{end}) + (\mathsf{end}, \mathsf{end}) = \Gamma, x\colon (\mathsf{end}, \mathsf{end})$. The operation is used to type the continuation process in the rules for input and output prefixes.

Equipped with the notions of type duality, unrestricted contexts, and context splitting and updating we are ready to introduce the typing rules in Figure 3. Those for values are

straightforward: the type of a boolean value is bool (rules [T-True] and [T-False]) and that of a variable is read from the context (rule [T-Var]). In both cases, the "unused" part of the context Γ must be unrestricted, so that linear values may be consumed.

For processes, rule [T-Inact] says that the terminated process can only be typed in an unrestricted context, thus ensuring that linear channels are not discarded. Rule [T-Par] uses context splitting to partition the context between the two processes: the incoming context is split into Γ_1 and Γ_2, and we use the former to type check process P_1 and the latter to type check process P_2. Rule [T-Repl] for replication requires the typing context not to contain linear values, for P may be used an unbounded number of times. Rule [T-If] for the conditional process splits the incoming context in two parts: one used to check the condition, the other to check both branches. The same context for the two branches is justified by the fact that only one of P_1 or P_2 will be executed. The rules for values require Γ_1 to be unrestricted. Linear types, if present in the conclusion $\Gamma_1 \circ \Gamma_2$ come from Γ_2, the context for the two branches. Rule [T-Res] allows restricting channels whose end points are dual, making sure that communication on the channel happens according to the plan.

There are two rules for input processes, [T-InL] and [T-InR], depending on whether the input end of the type for x presents itself on the left or on the right. Rule [T-InL], splits the incoming context in two: one to type x, the subject of communication, the other to type P, the continuation process. The type for x must be of the form $(q?T.S, S')$; the continuation process is typed at a context containing an entry $y\colon T$ for the bound variable. The new type for x is obtained by adding to the type of x in Γ_2 a type composed of the continuation S and the unused end point type S'. The table below summarises the various possibilities that match rule [T-InL]. Notice that the new type for x is constant in all four cases, regardless of how the context was split.

$(\Gamma_1 \circ \Gamma_2)(x)$	$\Gamma_1(x)$	$\Gamma_2(x)$	$\Gamma_2(x) \circ (S, S')$
$(\mathsf{lin}?T.S, L)$	$(\mathsf{lin}?T.S, L)$	$(\mathsf{end}, \mathsf{end})$	(S, L)
$(\mathsf{lin}?T.S, L)$	$(\mathsf{lin}?T.S, \mathsf{end})$	(end, L)	(S, L)
$(\mathsf{lin}?T.S, U)$	$(\mathsf{lin}?T.S, U)$	$(\mathsf{end}, \mathsf{end})$	(S, U)
$(\mathsf{lin}?T.S, U)$	$(\mathsf{lin}?T.S, \mathsf{end})$	(end, U)	(S, U)
$(\mathsf{lin}?T.S, U)$	$(\mathsf{lin}?T.S, U)$	(end, U)	(S, U)
$(\mathsf{un}?T.S, L)$	$(\mathsf{un}?T.S, L)$	$(\mathsf{end}, \mathsf{end})$	(S, L)
$(\mathsf{un}?T.S, L)$	$(\mathsf{un}?T.S, \mathsf{end})$	(end, L)	(S, L)
$(\mathsf{un}?T.S, L)$	$(\mathsf{un}?T.S, L)$	$(\mathsf{un}?T.S, \mathsf{end})$	(S, L)
$(\mathsf{un}?T.S, L)$	$(\mathsf{un}?T.S, \mathsf{end})$	$(\mathsf{un}?T.S, L)$	(S, L)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, U)$	$(\mathsf{end}, \mathsf{end})$	(S, U)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, \mathsf{end})$	(end, U)	(S, U)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, \mathsf{end})$	(S, U)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, \mathsf{end})$	$(\mathsf{un}?T.S, U)$	(S, U)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, U)$	(S, U)
$(\mathsf{un}?T.S, U)$	$(\mathsf{un}?T.S, U)$	(end, U)	(S, U)

The proviso in the typing rules for prefixes, $q = \mathsf{un} \Rightarrow q?T.S = S$, is necessary when we compose, e.g., (end, U) in Γ_2 with (S, U) in the last line of the table above. Would $\mathsf{un}?T.S$ be an arbitrary unrestricted type, we would be able to type the continuation at an arbitrary type $S = \mathsf{end} \circ S$. Take for example $S = \mathsf{lin}?\mathsf{bool}.\mathsf{end}$, so that the type under consideration becomes $\mathsf{un}?T.\mathsf{lin}?\mathsf{bool}.\mathsf{end}$. A channel that starts with an unrestricted type to become linear is always unsound. To see why, further take $\Gamma = x\colon (\mathsf{un}?T.S, \mathsf{un}!T.\overline{S})$ and $P = x(y).x(z)$. It should be easy to see that $\Gamma \vdash P$, and also $\Gamma \vdash !P \mid \overline{x}\, v$. But $!P \mid \overline{x}\, v$ reduces to process $!P \mid x(z)$ which is not typable under $x\colon (S, \overline{S})$. The converse, a linear-to-unrestricted evolution, is sound since no interference with the session is possible.

Rules [T-OutL] and [T-OutR] are similar, only that we split the context in three parts, the first to type x, the subject of communication, the second to type v, the object of communication, and the last to type P, the continuation process. If v is a linear channel end, splitting makes sure the continuation process keeps the channel at an end type. Splitting is quite flexible: given a (L_1, L_2) type, we may pass (L_1, L_2) and keep $(\mathsf{end}, \mathsf{end})$ (or vice versa), or pass (L_1, end) and keep (end, L_2) (or vice versa).

3.3. *An extended example*

Consider a service allowing to create online petitions, cf. (Vasconcelos, 2011). Petition creators receive from the petition service provider a channel on which they supply the title and the description of the petition, as well as the due date. After the initial setup, the exact same channel is ready to be distributed among the client's acquaintances to collect thousands of signatures, but not without the creator signing the petition first. The code for the creator can be written in the pi calculus as follows,

$$\begin{aligned} P &\stackrel{\text{def}}{=} \mathit{onlinePetition}(p).\overline{p}\,\langle \mathit{title}\rangle.\overline{p}\,\langle \mathit{description}\rangle.\overline{p}\,\langle \mathit{dueDate}\rangle.P' \\ P' &\stackrel{\text{def}}{=} \overline{p}\,\langle \mathit{signature}\rangle.(\overline{a_1}\,\langle p\rangle \mid \ldots \mid \overline{a_n}\,\langle p\rangle) \end{aligned}$$

where we added brackets to output prefixes in order to increase readability. Each of the acquaintances (not shown in the example), after reading p on channel a_i, can sign the petition by writing on channel p, and further distribute the channel at will. The code for the service provider is below:

$$\begin{aligned} Q &\stackrel{\text{def}}{=} !(\nu p)\overline{\mathit{onlinePetition}}\,\langle p\rangle.Q' \\ Q' &\stackrel{\text{def}}{=} p(\mathit{title}).p(\mathit{description}).p(\mathit{dueDate}).!p(\mathit{signature}).\mathit{ProcessSignatures} \end{aligned}$$

Replication allows an unbounded number of copies to be available, since the service must be accessible to multiple clients. For each copy, the service provider starts by creating a new petition channel p. One of its end points is then sent to the client via a message $\overline{\mathit{onlinePetition}}\,\langle p\rangle$, while the other is kept for further interaction in the continuation Q'.

The challenge is to let the client P and the service Q establish a (private) session on channel p and later allow the exact same channel to become shared by all potential petition signatories. Before seeing how this can be handled by our system, two considerations are in order. First note that the linear-to-unrestricted evolution of end point types is sound whereas the opposite is not (cf. discussion at end of Section 3.2). Second, we

stress two invariants of our linear discipline: types of the form (S_1, S_2) with S_1, S_2 linear enforce the read/write use of a channel in *at most* two threads, while types of the form (S, end) and (end, S) with S linear enforce read/write operations in *exactly* one thread.

The type T_1 below describes the behaviour of the petition channel p as used by process P. The entry on the left says that exactly two string messages followed by a date message date must be sent, after which an unbounded number of string messages may be exchanged. Conversely, the entry on the right denies any read/write operation on the channel.

$$T_1 \stackrel{\text{def}}{=} (\mathsf{lin!string.lin!string.lin!date.}*\mathsf{!string}, \mathsf{end})$$

We observe that T_1 does not guarantee that acquaintances (including the petition creator) sign the petition: this may be desirable, but not absolutely necessary. However, the linearity of the initial part enforces that any signed petition has a title, a description and a due date.

To guarantee that the communication proceeds as planned, the service should exhibit a compatible behaviour. This is attested by the type T_2 of channel p as used in the service's continuation Q'.

$$T_2 \stackrel{\text{def}}{=} (\mathsf{end}, \mathsf{lin?string.lin?string.lin?date.}*\mathsf{?string})$$

The fact that the right end point type of T_2 is dual to the left end point type of T_1 ensures a form of *safety*—that reduction does not get stuck (in rules [R-IfT], [R-IfF] and [R-Com]) and that processes do not engage in races for linear resources—as we show at the end of the section. Finally, the petition channel p, created by scope restriction in process Q, has type T, defined below.

$$T \stackrel{\text{def}}{=} (\mathsf{lin!string.lin!string.lin!date.}*\mathsf{!string}, \mathsf{lin?string.lin?string.lin?date.}*\mathsf{?string})$$

The reader will see that T is built by reconciling the left entry of T_1 with the right entry of T_2. Conversely, by means of the splitting rule $T = T_1 \circ T_2$ the left entry of T is delegated while the right entry of T is kept. By assigning the type $(*!T_1, *?T_1)$ to channel *onlinePetition* we can type check the parallel composition of the petition creator and the service provider. The details are below.

We start with a derivation for the client process P, where context Γ_1 contains the entries $a_i \colon (*!(*\mathsf{!string}, \mathsf{end}), \mathsf{end})$, *onlinePetition*$\colon (*!T_1, *?T_1)$, as well as *title*, *description*, *signature* of type string, and *dueDate* of type date. Notice that Γ_1 is unrestricted.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash \mathbf{0}}\,[\text{T-Inact}]}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash \overline{a_1}\,\langle p\rangle.\mathbf{0}}\,[\text{T-OutL}] \quad \dfrac{\dfrac{}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash \mathbf{0}}\,[\text{T-Inact}]}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash \overline{a_2}\,\langle p\rangle.\mathbf{0}}\,[\text{T-OutL}]}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash \overline{a_1}\,\langle p\rangle \mid \overline{a_2}\,\langle p\rangle}\,[\text{T-Par}]}{\Gamma_1, p \colon (*\mathsf{!string}, \mathsf{end}) \vdash P'}\,[\text{T-OutL}]}{\Gamma_1, p \colon (\mathsf{lin!date.}*\mathsf{!string}, \mathsf{end}) \vdash \overline{p}\,\langle \mathit{dueDate}\rangle.P'}\,[\text{T-OutL}]}{\Gamma_1, p \colon (\mathsf{lin!string.lin!date.}*\mathsf{!string}, \mathsf{end}) \vdash \overline{p}\,\langle \mathit{description}\rangle.\overline{p}\,\langle \mathit{dueDate}\rangle.P'}\,[\text{T-OutL}]}{\Gamma_1, p \colon T_1 \vdash \overline{p}\,\langle \mathit{title}\rangle.\overline{p}\,\langle \mathit{description}\rangle.\overline{p}\,\langle \mathit{dueDate}\rangle.P'}\,[\text{T-OutL}]}{\Gamma_1 \vdash P}\,[\text{T-InR}]$$

We now consider a derivation for service Q, where we take for Γ_2 an unrestricted

context compatible with Γ_1, i.e., a context such that $\Gamma_1 \circ \Gamma_2$ is defined. We further abbreviate the identifiers for the bound input variables, take Ps as a shorthand for process *ProcessSignatures*, and assume $\Gamma_2, p\colon (\mathsf{end}, *?\mathsf{string}), t\colon \mathsf{string}, r\colon \mathsf{string}, d\colon \mathsf{date}, s\colon \mathsf{string} \vdash Ps$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_2, p\colon (\mathsf{end}, *?\mathsf{string}), t\colon \mathsf{string}, r\colon \mathsf{string}, d\colon \mathsf{date}, s\colon \mathsf{string} \vdash Ps}{\Gamma_2, p\colon (\mathsf{end}, *?\mathsf{string}), t\colon \mathsf{string}, r\colon \mathsf{string}, d\colon \mathsf{date} \vdash p(s).Ps}\,[\text{T-InR}]}{\Gamma_2, p\colon (\mathsf{end}, *?\mathsf{string}), t\colon \mathsf{string}, r\colon \mathsf{string}, d\colon \mathsf{date} \vdash !p(s).Ps}\,[\text{T-Repl}]}{\Gamma_2, p\colon (\mathsf{end}, \mathsf{lin}?\mathsf{date}.*?\mathsf{string}), t\colon \mathsf{string}, r\colon \mathsf{string} \vdash p(d).!p(s).Ps}\,[\text{T-InR}]}{\Gamma_2, p\colon (\mathsf{end}, \mathsf{lin}?\mathsf{string}.\mathsf{lin}?\mathsf{date}.*?\mathsf{string}), t\colon \mathsf{string} \vdash p(r).p(d).!p(s).Ps}\,[\text{T-InR}]}{\Gamma_2, p\colon T_2 \vdash Q'}\,[\text{T-InR}]}{\Gamma_2, p\colon T \vdash \overline{onlinePetition}\,\langle p\rangle.Q'}\,[\text{T-OutL}]}{\Gamma_2 \vdash (\nu p)\overline{onlinePetition}\,\langle p\rangle.Q'}\,[\text{T-Res}]}{\Gamma_2 \vdash Q}\,[\text{T-Repl}]$$

We finally glue the results and conclude, by applying [T-Par], that $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q$.

We mentioned before that type end describes a channel end point on which no further interaction (read or write) is possible. This does not mean that a process that knows a given channel end point at type end cannot use it at all; it only means that read/write operations are barred. What other operations are then available on such an end point? In the simple form of pi calculus studied in this paper the only remaining operation is sending the channel around. We can nevertheless think of other operations, such as testing the channel for equality. Suppose that the petition server keeps track of the titles of all petitions so far created. Further suppose that channel *dbSet* is used to write a pair channel-title in the database. Upon the reception of the title, the server may update the database as follows.

$$Q'' \stackrel{\text{def}}{=} p(\mathit{title}).\overline{\mathit{dbSet}}\,\langle p, \mathit{title}\rangle.p(\mathit{description}).p(\mathit{dueDate})\ldots$$

Once again, the splitting relation allows to type this variant of the server. When typing the sub-process starting at $\overline{\mathit{dbSet}}\,\langle p, \mathit{title}\rangle$, the type $T_2' = (\mathsf{end}, \mathsf{lin}?\mathsf{string}.\mathsf{lin}?\mathsf{date}.*?\mathsf{string})$ of channel p is split in two: T_2' itself and $(\mathsf{end}, \mathsf{end})$. Type T_2' is used to type the continuation as before, whereas type $(\mathsf{end}, \mathsf{end})$ is used to pass p to the database. Taking D for $(*!(\mathsf{end}, \mathsf{end}), *?(\mathsf{end}, \mathsf{end}))$, Γ for $\mathit{dbSet}\colon D, p\colon T_2'$, and using rule [T-OutL] we have the following derivation.

$$\dfrac{\mathit{dbSet}\colon D, p\colon (\mathsf{end}, \mathsf{end}) \vdash \mathit{dbSet}\colon D \quad \mathit{dbSet}\colon D, p\colon (\mathsf{end}, \mathsf{end}) \vdash p\colon (\mathsf{end}, \mathsf{end}) \quad \Gamma \vdash p(d)\ldots}{\Gamma \vdash \overline{\mathit{dbSet}}\,\langle p\rangle.p(d)\ldots}$$

The database cannot use p to read or to write, but may store p in some data structure or compare it for equality, for example. The ability to pass part of the functionality of a linear channel and retain the rest goes beyond what is usually available to session typing systems, e.g. (Gay and Hole, 2005; Honda et al., 1998; Vasconcelos, 2012), (see Section 2).

3.4. *Well-typed programs do not go wrong*

Reduction preserves typability, but not for arbitrary contexts. To understand the situation, take for P the process $x(z).\mathsf{if}\ z\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0} \mid (\nu y)\overline{x}\,y$. We can easily see that P

is typable under the context $x\colon (\mathsf{lin!}(\mathsf{end},\mathsf{end}).\mathsf{end}, \mathsf{lin?bool}.\mathsf{end})$. But P reduces to process $(\nu y)\mathsf{if}\, y\, \mathsf{then}\, \mathbf{0}\, \mathsf{else}\, \mathbf{0}$ which is not typable (at all). The reader may have noticed the peculiarity of the type for channel x: one end point sends $(\mathsf{end},\mathsf{end})$, the other expects a bool. This context is not *balanced*: the type of one end point is not dual to the type of the other. The whole purpose of balancing is to make sure that the type of y in the output matches that of z in the input. A similar phenomenon happens with structural congruence. Take for T the type $(\mathsf{lin!bool}.\mathsf{end}, *\mathsf{!bool})$, then process $\overline{x}\,\mathsf{true} \mid !\overline{x}\,\mathsf{true}$ is typable under context $x\colon T$, but the structural congruent process $!\overline{x}\,\mathsf{true}$ is not. Once again, requiring balanced contexts solves the problem.

Requiring duality of the two end point types only makes sense when both end point types are "active", i.e., may engage in communications. An end point type end allows no further interaction at the end point, hence cannot be responsible for situations such as those depicted above. There is no reason to judge as unsafe process $x(y)$ when typed under context $x\colon (*!T, \mathsf{end})$. The $\mathsf{balanced}$ predicate for types is defined as below. The predicate is then extended point-wise to typing contexts. This is in essence the notion of (Gay and Hole, 2005) transposed to our setting.

$$\mathsf{balanced}(S, \overline{S}) \qquad \mathsf{balanced}(S, \mathsf{end}) \qquad \mathsf{balanced}(\mathsf{end}, S)$$

Notice that bool is not balanced, for it would allow to type, under balanced contexts, processes that we will judge unsafe, namely $x\colon \mathsf{bool} \vdash \mathsf{if}\, x\, \mathsf{then}\, P\, \mathsf{else}\, Q$. In short, processes typed under balanced contexts may have free channel variables but not free boolean variables. The types for the free channel variables are composed of two dual end points, or else one of them is end.

Given that type preservation is only certain under balanced contexts and that unbalanced types cannot possibly be restricted by rule [T-Res], the reader may wonder why we consider them at all. It turns out that unbalanced contexts are needed in certain derivations, in particular in situations where a thread holds the two ends of a same channel (Yoshida and Vasconcelos, 2007). For instance, process $\overline{z}\,x \mid z(w).w(y).\overline{x}\,\mathsf{true}$, typable under context $z\colon (*?S, *!S), x\colon (S, \overline{S})$ with $S = \mathsf{lin?bool}$, reduces to process $P = x(y).\overline{x}\,\mathsf{true}$, which we want to type under the same context. By applying rule [T-InL] to P we obtain a judgement with a $\mathsf{un\text{-}lin}$ type, namely $z\colon (*?S, *!S), x\colon (\mathsf{end}, \mathsf{lin!bool}) \vdash \overline{x}\,\mathsf{true}$. A further application of rule [T-OutR] gets rid of the $\mathsf{un\text{-}lin}$ type, yielding $z\colon (*?S, *!S), x\colon (\mathsf{end},\mathsf{end}) \vdash \mathbf{0}$ typable under rule [T-Inact]. We thus observe the presence of unbalanced contexts in a derivation tree conducting to a sequent with a balanced context.

We now discuss the sort of (malformed) processes our type system filters. The first obvious case happens when a conditional process tries to test a channel, rather than a boolean value. Examples include processes of the form $\mathsf{if}\, x\, \mathsf{then}\, P\, \mathsf{else}\, Q$ that do not occur under an input prefix on x. The second case happens when two processes try to communicate via rule [R-Com] but the substitution is not defined, as in $\overline{x}\,\mathsf{true} \mid x(y).\overline{y}\,\mathsf{false}$. Other than that there are no explicit run-time errors, since we work with a

monadic pi calculus[‡]. Errors related to the intended (linear) usage of channels are to be found on how a typing context classifies channels. For example, we classify as malformed a process that writes twice on x in parallel (as in $\overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{false}$), *if* it is typed under a context that associates x to $(\mathsf{lin!bool}.S_1, S_2)$ or to $(S_1, \mathsf{lin!bool}.S_2)$, but are willing to accept the process if the type of x is $(*\mathsf{!bool}, S)$ or $(S, *\mathsf{!bool})$.

It can be easily shown, by induction on the structure of processes, that for each process P there are $k \geq 0$ and $x_1, \ldots x_k, Q, R$ such that $P \equiv (\nu x_1) \ldots (\nu x_k)(Q \mid R)$. For our main result we need the notion of multi-step reduction. We say that P reduces in zero steps to Q if $P \equiv Q$, and that P reduces in $n+1$ steps to Q if $P \to R$ and R reduces in n steps to Q. By using typing preservation (Theorem 4.10) and rule [T-Res], we can show that, if $\Gamma \vdash P$ and P reduces in zero or more steps to $(\nu x_1) \ldots (\nu x_k)(Q \mid R)$ then there are $T_1, \ldots, T_n$ such that $x_1 \colon T_1, \ldots x_n \colon T_n \vdash Q \mid R$, where $0 \leq k \leq n$. Notice that variables $x_{k+1}, \ldots x_n$ are either free or do not occur at all in the initial process P. This justifies the formulation of our type safety theorem.

Theorem 3.1 (Main result). If $\Gamma \vdash P$ with $\mathsf{balanced}(\Gamma)$ and P reduces in zero or more steps to $(\nu x_1) \ldots (\nu x_k)(Q \mid R)$ then none of the following cases happen:

— $Q = \mathsf{if}\, x_i\, \mathsf{then}\, Q'\, \mathsf{else}\, Q''$,
— $Q = \overline{x_i}\, v.Q' \mid x_i(y).Q''$ and $Q''[v/y]$ is not defined,
— $Q = \overline{x_i}\, v.Q' \mid \overline{x_i}\, v'.Q''$ and $T_i = (\mathsf{lin!}T'.S_1, S_2)$,
— $Q = \overline{x_i}\, v.Q' \mid \overline{x_i}\, v'.Q''$ and $T_i = (S_1, \mathsf{lin!}T'.S_2)$,
— $Q = x_i(y).Q' \mid x_i(y).Q''$ and $T_i = (\mathsf{lin?}T'.S_1, S_2)$,
— $Q = x_i(y).Q' \mid x_i(y).Q''$ and $T_i = (S_1, \mathsf{lin?}T'.S_2)$,

where $x_1 \colon T_1, \ldots x_n \colon T_n \vdash Q \mid R$, for all $0 \leq i, k \leq n$, and all $T_1, \ldots, T_n$.

4. Proof of the main result

This section presents an outline of the proof of Theorem 3.1. After a few general lemmas about context splitting, we build towards the result that structural congruence preserves typability, by proving standard results such as weakening and strengthening. Before establishing soundness, we still need a substitution lemma.

In the following, we use the notation $\mathrm{dom}(\Gamma)$ and $\mathrm{range}(\Gamma)$ to denote respectively the set of variables and the set of types in Γ. The *unrestricted closure* of an end point is defined as $\mathcal{U}(\mathsf{lin}\, p) = \mathsf{end}$ and $\mathcal{U}(S) = S$ if $\mathsf{un}(S)$. The unrestricted closure of a type is defined as $\mathcal{U}(S_1, S_2) = (\mathcal{U}(S_1), \mathcal{U}(S_2))$, and $\mathcal{U}(\mathsf{bool}) = \mathsf{bool}$. The notion is then extended pointwise to typing contexts. Unrestricted closure is idempotent when applied to unrestricted contexts.

Lemma 4.1. If $\mathsf{un}(\Gamma)$ then $\mathcal{U}(\Gamma) = \Gamma$.

Proof. The result follows from the fact that $\mathcal{U}(T)$ differs from T only when T contains a linear end point type. □

[‡] This exact observation motivates the introduction of the polyadic pi calculus (Milner, 1993).

Lemma 4.2 (Properties of context splitting). Let $\Gamma = \Gamma_1 \circ \Gamma_2$.

1 $\Gamma_3 = \Gamma_2 \circ \Gamma_1$, for some Γ_3.
2 If $\Gamma_1 = \Gamma_3 \circ \Gamma_4$, then $\Gamma_5 = \Gamma_4 \circ \Gamma_2$ and $\Gamma = \Gamma_3 \circ \Gamma_5$, for some Γ_5.
3 $\Gamma = \mathcal{U}(\Gamma) \circ \Gamma$ and $\Gamma = \Gamma \circ \mathcal{U}(\Gamma)$.
4 $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$.
5 If $\mathsf{balanced}(\Gamma)$ then $\mathsf{balanced}(\Gamma_1)$ and $\mathsf{balanced}(\Gamma_2)$.

Proof. A straightforward induction on the length of Γ. □

Lemma 4.3 (Unrestricted type weakening). Let T be an unrestricted type and U an unrestricted end point type.

1 If $\Gamma \vdash v\colon T'$ then $\Gamma, x\colon T \vdash v\colon T'$.
2 If $\Gamma \vdash P$ then $\Gamma, x\colon T \vdash P$.
3 If $\Gamma, x\colon (S, \mathsf{end}) \vdash P$ then $\Gamma, x\colon (S, U) \vdash P$.
4 If $\Gamma, x\colon (\mathsf{end}, S) \vdash P$ then $\Gamma, x\colon (U, S) \vdash P$.

Proof. A simple case analysis in the first case, and induction on the structure of P in remaining. □

Corollary 4.4 (Unrestricted context weakening). If $\Gamma_1 \vdash P$ and $\mathsf{un}(\Gamma_2)$ and $\Gamma = \Gamma_1 \circ \Gamma_2$ then $\Gamma \vdash P$.

Proof. We note that Γ_1 differs from Γ in a number of end point types that are equal to end in $\mathrm{range}(\Gamma_1)$ and equal to U in $\mathrm{range}(\Gamma)$. By repeated applications of Lemma 4.3 we obtain the desired result, $\Gamma \vdash P$. □

Lemma 4.5 (Strengthening).

1 If $\Gamma, x\colon T \vdash v\colon T'$ and $x \neq v$ then $\Gamma \vdash v\colon T'$ and $\mathsf{un}(T)$.
2 If $\Gamma, x\colon T \vdash P$ and $x \notin \mathrm{fv}(P)$ then $\Gamma \vdash P$ and $\mathsf{un}(T)$.

Proof. A simple case analysis in the first case, and induction on the structure of P in the second. □

To show that structural congruence preserves typability, we need a preliminary lemma that we will use to close the case of replication.

Lemma 4.6. Let L be a linear end point type and $R = \mathsf{end}$ or $R = \overline{L}$.

1 If $\Gamma_1, x\colon (L, S) \vdash P$ then $\Gamma_2, x\colon (\mathsf{end}, R) \nvdash P$;
2 If $\Gamma_1, x\colon (S, L) \vdash P$ then $\Gamma_2, x\colon (R, \mathsf{end}) \nvdash P$.

Proof. By induction on the structure of the derivation. We draw an example for the first item; the remaining cases are similar. Assume that $\Gamma, z\colon (L, S) \vdash x(y).P$. When the derivation ends with [T-InL], we know that $\Gamma, z\colon (L, S) = \Gamma_1 \circ \Gamma_2$, $\Gamma_1 \vdash x\colon (q?T.S', S'')$ and $\Gamma_2 + x\colon (S', S'')), y\colon T \vdash P$. When $z \neq x$ the case follows directly from the induction hypothesis. Otherwise, when $z = x$, we know that $q = \mathsf{lin}$ and $L = \mathsf{lin}\,?T.S'$. For $R = \mathsf{end}$ or $R = \mathsf{lin}!T.S'$, a case analysis shows that there does not exist a rule to terminate a derivation of $\Gamma', x\colon (\mathsf{end}, R) \vdash x(y).P$. □

Lemma 4.7 (Structural congruence preserves typability). If $\Gamma \vdash P$ and $\mathsf{balanced}(\Gamma)$ and $P \equiv Q$ then $\Gamma \vdash Q$.

Proof. The proof follows by an analysis of the axioms in the definition of the relation $\equiv$. The axioms for associativity, commutativity, and the neutral of parallel composition follow from the basic results about context splitting, Lemma 4.2 items 1–3. The case of $(\nu x)\mathbf{0} \equiv \mathbf{0}$ is straightforward to establish. The case of $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$ follows by the fact that contexts are considered up to permutations of its entries.

The case of axiom $!P \equiv P \mid !P$, when read from left to right, relies on Lemma 4.1 and Lemma 4.2 item 3. When read right-to-left, we know from rules [T-Par],[T-Repl] that $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash P$, $\Gamma_2 \vdash !P$, where the latter implies $\Gamma_2 \vdash P$ and $\mathsf{un}(\Gamma_2)$. Next we show that $\mathsf{un}(\Gamma_1)$. The case $\Gamma_1 = \emptyset$ is clear. Otherwise let $\Gamma_1(x) = T$. From Lemma 4.2.5 and the definition of context splitting, we know that T is balanced. Distinguish two cases: when $T = \mathsf{bool}$ we have $\mathsf{un}(\mathsf{bool})$; otherwise $T = (S, R)$ and the rules for context splitting, the hypothesis $\Gamma_2 \vdash P$ and Lemma 4.6.1 tell us that $\mathsf{un}(S)$, while the splitting rules, the hypothesis $\Gamma_2 \vdash P$, and Lemma 4.6.2 tell us that $\mathsf{un}(R)$, and in turn $\mathsf{un}(S, R)$. We apply Corollary 4.4 to $\Gamma_2 \vdash !P$ and infer the desired result, $\Gamma \vdash !P$.

In the case of axiom $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$, the variable convention tells us that $x \notin \mathrm{fv}(Q)$ and also that $x \notin \Gamma$. When read left-to-right, we know from rules [T-Par],[T-Res] that $\Gamma_1, x\colon (S, \overline{S}) \vdash P$ and $\Gamma_2 \vdash Q$ with $\Gamma = \Gamma_1 \circ \Gamma_2$. We start by weakening the sequent $\Gamma_2 \vdash Q$ by introducing $x\colon (S', \overline{S'})$ where S' is S if $\mathsf{un}(S)$, else end. It is then easy to see that $(\Gamma_1, x\colon (S, \overline{S})) \circ (\Gamma_2, x\colon (S', \overline{S'}))$ is defined and equal to $\Gamma, x\colon (S, \overline{S})$. Conclude the case with rules [T-Par],[T-Res]. For the right-to-left case, the same two rules give us that $\Gamma, x\colon (S, \overline{S}) = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. We now analyse Γ_1 and Γ_2. Lemma 4.2 item 4 tells us that Γ_1, Γ_2 are of the form $\Gamma_1', x\colon T_1$, $\Gamma_2', x\colon T_2$, and Lemma 4.5 tells that T_2 is unrestricted and that $\Gamma_2' \vdash Q$. When $S = \mathsf{lin}\, p$ we apply [T-Res] and infer $\Gamma_1' \vdash (\nu x)P$. Otherwise, we have $\mathsf{un}(S)$, and four cases arise, corresponding to $T = (S, \overline{S})$ or $T = (\mathsf{end}, \mathsf{end})$ or $T = (S, \mathsf{end})$ or $T = (\mathsf{end}, \overline{S})$. In the first two cases we apply [T-Res] and infer $\Gamma_1' \vdash (\nu x)P$. In the last two we use weakening (Lemma 4.3) to obtain $\Gamma_1', x\colon (S, \overline{S}) \vdash P$. Conclude with rule [T-Par]. □

Lemma 4.8 (Inversion of the splitting relation). Let T_1 be a type of the form $(q_1!T_1'.S_1, q_2?T_2'.S_2)$. If $\Gamma = (\Gamma_1, x\colon T_1) \circ \Gamma_2$, then $x\colon T_1 \in \Gamma$ and $x\colon T_2 \in \Gamma_2$, where T_2 is of one of the four forms below.

1 $(\mathsf{end}, \mathsf{end})$
2 T_1 when $q_1 = q_2 = \mathsf{un}$
3 $(q_1!T_1'.S_1, \mathsf{end})$ when $q_1 = \mathsf{un}$
4 $(\mathsf{end}, q_2?T_2'.S_2)$ when $q_2 = \mathsf{un}$

Proof. Directly from the definition of context splitting. □

Note that there is an obvious dual lemma where $x\colon (q_2?T_2'.S_2, q_1!T_1'.S_1) \in \Gamma$ and T_2 is $(\mathsf{end}, q_1!T_1'.S_1)$ in case 3, and is $(q_2?T_2'.S_2, \mathsf{end})$ in case 4.

Lemma 4.9 (Substitution). If $\Gamma_1 \vdash v\colon T$ and $\Gamma_2, x\colon T \vdash P$ and $\Gamma_3 = \Gamma_1 \circ \Gamma_2$, then $P[v/x]$ is defined and $\Gamma_3 \vdash P[v/x]$.

Proof. The proof is by induction on the typing derivation and uses weakening and strengthening (Lemmas 4.3, 4.5). The proof is elaborate but straightforward. The result $P[v/x]$ follows by rules [T-InL],[T-InR],[T-OutL],[T-OutC], where it is required to type the subject x of an input or an output at channel type (S_1, S_2) by using [T-Var]; thus $T = \mathsf{bool}$ implies x not occurring as subject, as required. □

Theorem 4.10 (Type preservation). If $\Gamma_1 \vdash P_1$ with Γ_1 balanced and $P_1 \to P_2$, then $\Gamma_2 \vdash P_2$ and Γ_2 balanced and

1 $\Gamma_1 = \Gamma_2$, or
2 $\Gamma_1 = \Gamma_3, x\colon (q!T.S, q?T.\overline{S})$ and $\Gamma_2 = \Gamma_3, x\colon (S, \overline{S})$, or
3 $\Gamma_1 = \Gamma_3, x\colon (q?T.S, q!T.\overline{S})$ and $\Gamma_2 = \Gamma_3, x\colon (S, \overline{S})$.

Furthermore, in the last two cases, if $q = \mathsf{un}$ then $q!T.S = S$.

Proof. The proof is by induction on the derivation for the reduction. Notice that Γ_2 is balanced in any case.

When the derivation ends with rule [R-IfT] or [R-IfF] we use Corollary 4.4. When reduction ends with rule [R-Struct] we use Lemma 4.7.

When reduction ends with rule [R-Par] we have $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. We apply Lemma 4.2 item 5 and infer $\mathsf{balanced}(\Gamma_1)$. Given the induction hypothesis we know that $\Gamma_1' \vdash P'$. Distinguish the three situations as in the statement of the theorem. For the first, $\Gamma_1 = \Gamma_1'$, complete the proof with the [T-Par] rule. The two remaining cases are similar; we concentrate on the second (of the three). We know by induction that $\Gamma_1 = \Gamma_3, x\colon (q!T.S, q?T.\overline{S})$ and $\Gamma_1' = \Gamma_3, x\colon (S, \overline{S})$. Further distinguish two cases: $q = \mathsf{lin}$ and $q = \mathsf{un}$. When $q = \mathsf{lin}$ we know that $\Gamma_1 \circ \Gamma_2 = (\Gamma_3, x\colon (\mathsf{lin}!T.S, \mathsf{lin}?T.\overline{S})) \circ (\Gamma_2', x\colon (\mathsf{end}, \mathsf{end})) = \Gamma_3 \circ \Gamma_2', x\colon (\mathsf{lin}!T.S, \mathsf{lin}?T.\overline{S})$, hence $\Gamma_1' \circ \Gamma_2 = \Gamma_3 \circ \Gamma_2', x\colon (S, \overline{S})$ as required. When $q = \mathsf{un}$, we know that $\Gamma_1 \circ \Gamma_2 = (\Gamma_3, x\colon (\mathsf{un}!T.S, \mathsf{un}?T.\overline{S})) \circ (\Gamma_2', x\colon T)$. There are four cases for T, namely $(\mathsf{un}!T.S, \mathsf{un}?T.\overline{S})$, $(\mathsf{un}!T.S, \mathsf{end})$, $(\mathsf{end}, \mathsf{un}?T.\overline{S})$ and $(\mathsf{end}, \mathsf{end})$. In all cases, $\Gamma_1 \circ \Gamma_2 = \Gamma_3 \circ \Gamma_2', x\colon (\mathsf{un}!T.S, \mathsf{un}?T.\overline{S})$ hence $\Gamma_1' \circ \Gamma_2 = \Gamma_3 \circ \Gamma_2', x\colon (\mathsf{un}!T.S, \mathsf{un}?T.\overline{S})$. But in this case, the induction hypothesis also tells us that $\mathsf{un}!T.S = S$ as required.

When the derivation ends with rule [R-Res] we know that $\Gamma, y\colon (S, \overline{S}) \vdash P$ from $\Gamma \vdash (\nu y)P$. The induction hypothesis tells us that $\Gamma_2 \vdash P'$ for Γ_2 in the conditions of the theorem. We distinguish two cases. When $y = x$, we know that $\Gamma = \Gamma_3$ and apply rule [T-Res] to complete the case. Otherwise, we know that $\Gamma_3 = \Gamma_3', y\colon T$ and we apply rule [T-Res] to obtain $\Gamma_3', x\colon T_2 \vdash (\nu y)P$; it should be easy too see that $\Gamma_3', x\colon T_2$ is in the form required by the theorem.

When the derivation ends with rule [R-Com], there are two derivations to consider: [T-Par] preceded by [T-OutL] and [T-InR], or [T-Par] preceded by [T-OutR] and [T-InL]. We concentrate on the former, the latter is similar. From the derivation we learn that

1 $\Gamma = \Gamma_1' \circ \Gamma_1'' \circ \Gamma_1''' \circ \Gamma_2' \circ \Gamma_2''$,
2 $\Gamma_1' \vdash x\colon (q_1!T_1.S_1, S_1')$,
3 $\Gamma_1'' \vdash v\colon T_1$,
4 $\Gamma_1''' \circ x\colon (S_1, S_1') \vdash P$,

5 $\Gamma_2' \vdash x\colon (S_2', q_2?T_2.S_2)$,
6 $(\Gamma_2'' \circ x\colon (S_2', S_2)), y\colon T_2 \vdash Q$,
7 $q_1 = \mathsf{un}$ implies $q_1!T_1.S_1 = S_1$, and $q_2 = \mathsf{un}$ implies $q_2?T_2.S_2 = S_2$.

We claim that $T_1 = T_2$ and that there is a context Δ_1 such that $\Delta_1 = \Gamma_1'' \circ (\Gamma_2'' \circ x\colon (S_2', S_2))$ holds. By (3), (6) and the substitution lemma we get $\Delta_1 \vdash Q[v/y]$. We further claim that $\Delta_2 = (\Gamma_1''' \circ x\colon (S_1, S_1')) \circ \Delta_1$ holds for some Δ_2. From (4), $\Delta_1 \vdash Q[v/y]$, and the [T-Par] rule we obtain $\Delta_2 \vdash P \mid Q[v/y]$. It remains to show that $q_1 = q_2$ and $S_1 = \overline{S_2}$ and $\Gamma = \Gamma_3, x\colon (q_1!T_1.S_1, q_1?T_1.\overline{S_1})$ and $\Delta_2 = \Gamma_3, x\colon (S_1, \overline{S_1})$, in addition to the three claims above. We address the various pending results in turn.

From the properties of context split, Lemma 4.2, we know that $\Gamma_1' \circ \Gamma_2'$ and $\Gamma_1'' \circ \Gamma_1''' \circ \Gamma_2''$ are defined. Let T be the type $(q_1!T_1.S_1, q_2?T_2.S_2)$. From (2) and (5) and the definition of context splitting we know that $x\colon T \in \Gamma_1' \circ \Gamma_2'$. Now we apply Lemma 4.8 to contexts $\Gamma_1' \circ \Gamma_2'$ and $\Gamma_1'' \circ \Gamma_1''' \circ \Gamma_2''$, to obtain $x\colon T \in \Gamma$. But Γ is balanced by hypothesis, hence $T_1 = T_2$ and $S_1 = \overline{S_2}$ and $q_1 = q_2$. It remains to show that $x\colon (S_1, S_2)$ is in Δ_2.

Let $U = (U_1, U_2)$ be the type for x in $\Gamma_1'' \circ \Gamma_1''' \circ \Gamma_2''$. Lemma 4.8 also allows to figure out the four possible cases for U. Taking for A an arbitrary end point type, and abbreviating types $\mathsf{un}!T_1.S_1$ and $\mathsf{un}?T_1.S_2$ to $\mathsf{un}!$ and $\mathsf{un}?$, respectively, we fill the below table to complete the proof.

U_1	U_2	S_1	S_1'	S_2'	S_2	$U_1 \circ S_2'$	$U_2 \circ S_2$	$S_1 \circ (U_1 \circ S_2')$	$S_1' \circ (U_2 \circ S_2)$
end	end	A	end	end	$\overline{A}$	end	$\overline{A}$	A	$\overline{A}$
un!	un?	un!	un?	un!	un?	un!	un?	un!	un?
un!	end	un!	end	un!	un?	un!	un?	un!	un?
end	un?	un!	un?	end	un?	end	un?	un!	un?

It should be easy to see that the type for x in each context Γ_1'', Γ_1''' and Γ_2'' is exactly of one of the four possible forms for U. The columns for S_1 and S_2 are filled together based on the following facts: contexts $\Gamma_1''' \circ x\colon (S_1, S_1')$ and $\Gamma_2'' \circ x\colon (S_2', S_2)$ are defined, $S_1 = \overline{S_2}$, and item (7) above. The columns for S_1 and S_2' are again filled together based on the following facts: $(q_1!T_1.S_1, q_2?T_2.S_2) = (q_1!T_1.S_1, S_1') \circ (S_2', q_2?T_2.S_2)$, contexts $\Gamma_1''' \circ x\colon (S_1, S_1')$ and $\Gamma_2'' \circ x\colon (S_2', S_2)$ are defined. From the type splitting we get that S_2' is end or $\mathsf{un}!$, and S_1' is end or $\mathsf{un}?$. □

We are finally in a position to prove our main result.

Proof of the main result, Theorem 3.1. We first show that there is a balanced context Δ such that $\Delta \vdash (\nu x_1) \ldots (\nu x_k)(Q \mid R)$. If P reduces in zero steps, then use Lemma 4.7; otherwise use induction and type preservation, Theorem 4.10.

Let $\Delta = x_{k+1}\colon T_{k+1}, \ldots x_n\colon T_n$. Applying k times rule [T-Res] to the above sequent we obtain $\Delta' \vdash Q \mid R$, where $\Delta' = x_1\colon T_1, \ldots x_n\colon T_n$. Notice that Δ' is still balanced. Now, the first item does not hold because $\mathsf{bool} \notin \mathrm{range}(\Delta')$, and rule [T-If] therefore does not apply.

For the second item, we consider the structure of the derivation of $\Delta' \vdash Q \mid R$. There are four cases to consider depending on the choice of the left/right rules for input and output. All four cases are similar. One case uses rule [T-Par] twice, and rules

[T-InL] and [T-OutL], to obtain $\Delta_1'' \vdash v\colon T$ and $(\Delta_2'' + x\colon (S, S')), y\colon T \vdash Q''$. The properties of context splitting, Lemma 4.2, guarantee that there is a context Γ such that $\Gamma = \Delta_1'' \circ (\Delta_2'' + x\colon (S, S'))$. We are then in the conditions of substitution lemma, Lemma 4.9, which guarantees that $Q''[v/y]$ is defined.

For the third item, we show that the hypothesis $\mathsf{balanced}(T_i)$ leads to a contradiction. From $\mathsf{balanced}(\Delta')$ and $T_i = (\mathsf{lin!}T.S', S'')$ we obtain $S'' = \mathsf{end}$ or $S'' = \overline{\mathsf{lin!}T.S'}$. There are four cases to consider depending on the choice of the left/right rules for the two output processes. All four cases are similar. One case uses rule [T-Par] twice, and rules [T-OutL] twice again, to obtain $\Delta_1' \vdash x\colon (q_1!T_1.S_1, S_1')$ and $\Delta_2' \vdash x\colon (q_2!T_2.S_2, S_2')$. We have reached a contradiction since T_i can never be split into the two types for x above. The proofs of the remaining items are similar. □

5. Embeddings

In this section we assess the expressiveness of our typing system by embedding three systems of reference for session and linear types for the pi calculus. We encode the pi calculus with polarities and session types (Gay and Hole, 2005) (hence the conventional pi calculus (Milner et al., 1992)), the original version of pi with session types or the pi calculus with accept/request primitives (Honda et al., 1998), and the linear pi calculus (Kobayashi et al., 1999). For each of these languages we prove an operational and a typing correspondence result. In the case of the linear pi calculus we also provide a completeness result, thus proving that linear pi is a sub-language of ours. The embeddings highlight the fact that our type system is an extension of advanced type systems for the pi calculus.

5.1. *Embedding the pi calculus with polarities*

This section shows that the polarity system introduced by (Gay and Hole, 2005) can be embedded in our system. Since Gay and Hole show that the simply typed pi calculus can be embedded in the pi calculus with polarities; by transitivity the simply typed pi calculus can be embedded in our system as well.

In Figure 4 we present the branch/select-free fragment of the pi calculus with polarities. Variables may be optionally *polarised*, occurring in processes as well as in typing contexts as x^+ or x^- or simply as x. We write x^p for a general polarised variable, where p represents an optional polarity. Duality on polarities, written $\overline{p}$ exchanges $+$ and $-$. The new constructors of the language, input and output, are in Figure 4; the remaining are taken from Figure 1 (except for the conditional process which we do not consider); the syntactic category for values in Figure 1 does not contribute to the language.

The reduction relation, denoted by $\rightarrow_{\mathsf{p}}$, is defined inductively by the rules in Figure 1 with rule [R-Com] replaced by that in Figure 4 (rules [R-IfT], [R-IfF] excluded). It is easy to see that the two languages differ in the (optional) polarity annotation on (non-bound occurrences of) variables. We define an erase function that removes from a polarised process all occurrences of $+$ and $-$, to yield a process generated by the grammar

New syntactic forms (extends Figure 1)

$P ::= \ldots$		Processes:		$\mu a.S$	recursive type
	$\overline{x^p}\, x^p.P$	output	$S ::=$		Session types:
	$x^p(x).P$	input		end	termination
$T ::=$		Types:		$?T.S$	receive
	$\hat{}T$	shared type		$!T.S$	send
	S	session type		a	type variable
	a	type variable		$\mu a.S$	recursive type

$\boxed{P \to_{\mathsf{p}} P}$ *New reduction rules (extends Figure 1)*

$$\overline{x^p}\, z^q.P \mid x^{\overline{p}}(y).Q \to_{\mathsf{p}} P \mid Q[z^q/y] \qquad \text{[R-Com]}$$

$\boxed{\Gamma + x\colon T = \Gamma}$ *Context update rules*

$$\Gamma + x^p\colon S = \Gamma, x^p\colon S \qquad \text{if } x^p, x \notin \Gamma$$
$$\Gamma + x\colon T = \Gamma, x\colon T \qquad \text{if } x, x^+, x^- \notin \Gamma$$
$$(\Gamma, x\colon \hat{}T) + x\colon \hat{}T = \Gamma, x\colon \hat{}T$$

$\boxed{\Gamma \vdash_{\mathsf{p}} P}$ *Typing rules*

$$\frac{\Gamma \text{ completed}}{\Gamma \vdash_{\mathsf{p}} \mathbf{0}} \qquad \frac{\Gamma \vdash_{\mathsf{p}} P \qquad \Gamma \text{ unlimited}}{\Gamma \vdash_{\mathsf{p}} *P} \qquad \frac{\Gamma, x\colon \hat{}T \vdash_{\mathsf{p}} P}{\Gamma \vdash_{\mathsf{p}} (\nu x)P} \qquad \frac{\Gamma, x^+\colon S, x^-\colon \overline{S} \vdash_{\mathsf{p}} P}{\Gamma \vdash_{\mathsf{p}} (\nu x)P}$$

[T-Inact] [T-Repl] [T-New] [T-NewS]

$$\frac{\Gamma_1 \vdash_{\mathsf{p}} P \qquad \Gamma_2 \vdash_{\mathsf{p}} Q}{\Gamma_1 + \Gamma_2 \vdash_{\mathsf{p}} P \mid Q} \qquad \frac{\Gamma, x\colon \hat{}T, y\colon T \vdash_{\mathsf{p}} P}{\Gamma, x\colon \hat{}T \vdash_{\mathsf{p}} x(y).P} \qquad \frac{\Gamma, x^p\colon S, y\colon T \vdash_{\mathsf{p}} P}{\Gamma, x^p\colon ?T.S \vdash_{\mathsf{p}} x^p(y).P}$$

[T-Par] [T-In] [T-InS]

$$\frac{\Gamma, x\colon \hat{}T \vdash_{\mathsf{p}} P}{(\Gamma, x\colon \hat{}T) + y^q\colon T \vdash_{\mathsf{p}} \overline{x}\, y^q.P} \qquad \frac{\Gamma, x^p\colon S \vdash_{\mathsf{p}} P}{(\Gamma, x^p\colon !T.S) + y^q\colon T \vdash_{\mathsf{p}} \overline{x^p}\, y^q.P} \qquad \text{[T-Out] [T-OutS]}$$

Fig. 4. Pi calculus with polarities

in Figure 1. There is an obvious operational correspondence between the two languages, stated in Theorem 5.1.

The language of types includes a distinct category S for (linear) session types; in contrast to (Gay and Hole, 2005) we ignore subtyping as well as recursive types but take recursive session types into account. Duality is defined as in Section 3, with the appropriate syntactic changes which amount to erasing the qualifiers. Typing contexts now gather assumptions on polarised variables, in addition to simple variables as before. There is however one restriction on the variables occurring in a context: x and x^+ (or x^-) cannot occur simultaneously in a given context Γ, even though x^+ and x^- may. New assumptions are added to contexts by means of an update operation $+$, defined in Figure 4. This operation is stricter than context splitting in our system, for it does not take into account the "neutral" nature of type end. For example, we have that $x\colon (\mathsf{end}, \mathsf{end}) + x\colon (q!T.S, S')$ is defined in our system, whereas $x\colon \mathsf{end} + x\colon !T.S$ is not defined in the polarity system.

We say that a context is *unlimited* if it contains no session types, and is *completed* if all session types it contains are end.

The typing relation is inductively defined by the rules in Figure 4. The first thing to notice is that the rule for restriction in our system corresponds to a pair of rules in the system with polarities, one that takes care of shared types (unrestricted), whereas the other deals with session types (linear). Rules [T-In] and [T-InS] in Figure 4 have their counterpart in a single (left or right) rule in Figure 3, namely [T-InL] and [T-InR], thanks to type qualifiers, the context splitting and the context update operations. The same can be said of the relation between rules [T-Out],[T-OutS] and [T-OutL],[T-OutR]. Rule [T-NewS] introduces in the context two channel end points of dual types; contrast with rule [T-Res] in Figure 3: our system gathers the two end points in a single variable whose type contains the two dual end point types.

From the above description it should be obvious that the two systems are quite close to each other. In order to define the typing correspondence we need to translate session types, types and contexts for the polarised language (as in Figure 4) into end point types, types and contexts in our language (Figure 1).

Function $\{\!|S|\!\}$ translates session types, $[\![T]\!]$ translates types. The translation of session types is straightforward. For types, we translate $\hat{}T$ into an unrestricted type capable of continuously inputting and outputting values of type $[\![T]\!]$; recall from Section 3 that we use $*?T$ as an abbreviation for $\mu a.\mathsf{un}?T.a$, for some a not in T. A session type S, when interpreted as a type, is translated into a pair containing $\{\!|S|\!\}$ and end for the two end points. To translate contexts we assume that if both x^+ and x^- are in Γ then they occur in contiguous positions (and in this order). If this is not the case, then the context can be rearranged since we work with contexts up to entry permutation. The translation of contexts is given by the rules below, which must be tried in the given order; the first rule for mapping non-empty contexts is for polarised pairs while the second rule is for single entries.

$$
\begin{array}{lll}
\{\!|!T.S|\!\} = \mathsf{lin}![\![T]\!].\{\!|S|\!\} & \{\!|a|\!\} = a & [\![\emptyset]\!] = \emptyset \\
\{\!|?T.S|\!\} = \mathsf{lin}?[\![T]\!].\{\!|S|\!\} & [\![\hat{}T]\!] = (*?[\![T]\!], *![\![T]\!]) & [\![\Gamma, x^+ \colon S, x^- \colon S']\!] = [\![\Gamma]\!], x \colon (\{\!|S|\!\}, \{\!|S'|\!\}) \\
\{\!|\mathsf{end}|\!\} = \mathsf{end} & [\![S]\!] = (\{\!|S|\!\}, \mathsf{end}) & [\![\Gamma, x^p \colon S]\!] = [\![\Gamma]\!], x \colon [\![S]\!] \\
\{\!|\mu a.S|\!\} = \mu a.\{\!|S|\!\} & & [\![\Gamma, x \colon \hat{}T]\!] = [\![\Gamma]\!], x \colon [\![\hat{}T]\!]
\end{array}
$$

We are now in a position to state the main result of this section.

Theorem 5.1 (Polarity-pi to pi correspondence).

1 If $\Gamma \vdash_{\mathsf{p}} P$ then $[\![\Gamma]\!] \vdash \mathrm{erase}(P)$.
2 If $P \to_{\mathsf{p}} Q$, then $\mathrm{erase}(P) \to \mathrm{erase}(Q)$.

Proof. (1). The proof is by induction on the derivation of $\Gamma \vdash_{\mathsf{p}} P$, and uses unrestricted weakening (Lemma 4.3) to add polarised variables at type $(\mathsf{end}, \mathsf{end})$ in order to make context splitting defined. As an example, we draw two interesting cases.

For [T-InS] let $\Gamma, x^- \colon ?T.S \vdash_{\mathsf{p}} x(y).P$ be inferred from $\Gamma, x^- \colon S, y \colon T \vdash_{\mathsf{p}} P$ (the case for x^+ is similar). Assume the induction hypothesis to be $[\![\Gamma, x^- \colon S, y \colon T]\!] \vdash \mathrm{erase}(P)$. There are two cases corresponding to (i) $\Gamma = \Gamma', x^+ \colon S'$ or (ii) $x^+ \notin \Gamma$. In case (i) we

rewrite the induction hypothesis as $[\![\Gamma']\!], x\colon (\{[\![S']\!]\}, \{[\![S]\!]\}), y\colon [\![T]\!] \vdash \mathrm{erase}(P)$. We apply [T-InR] and infer $[\![\Gamma']\!], x\colon (\{[\![S']\!]\}, \mathsf{lin}\,?[\![T]\!].\{[\![S]\!]\}), y\colon [\![T]\!] \vdash \mathrm{erase}(P)$ as required. In case (ii), noticing that $(\mathsf{end}, \mathsf{end}) \circ (\{[\![S]\!]\}, \mathsf{end}) = [\![S]\!]$, we rewrite the induction hypothesis as $[\![\Gamma]\!], x\colon [\![S]\!], y\colon [\![T]\!] \vdash \mathrm{erase}(P)$. We apply [T-InL] and infer $[\![\Gamma]\!], x\colon (\mathsf{lin}\,?[\![T]\!].\{[\![S]\!]\}, \mathsf{end}) \vdash x(y).\,\mathrm{erase}(P)$.

For [T-Out], let $(\Gamma, x\colon \hat{}T) + y^q\colon T \vdash_{\mathsf{p}} \overline{x}\, y^q.P$ be inferred from $\Gamma, x\colon \hat{}T \vdash_{\mathsf{p}} P$. The induction hypothesis is $[\![\Gamma, x\colon \hat{}T]\!] \vdash \mathrm{erase}(P)$. If $y^{\overline{q}} \notin \mathrm{dom}(\Gamma)$, we use weakening and infer from the induction hypothesis the judgement $[\![\Gamma, x\colon \hat{}T]\!], y\colon (\mathsf{end}, \mathsf{end}) \vdash \mathrm{erase}(P)$. In either case we complete the proof by applying [T-OutR].

(2). By a straightforward induction on the derivation of a reduction step $\rightarrow_{\mathsf{p}}$. □

The converse of the above result is clearly not true. Take for P the polarised process $\overline{x^+}\, y \mid x^+(z)$. Then $\mathrm{erase}(P) = \overline{x}\, y \mid x(z)$ reduces while P does not. Also $\mathrm{erase}(P)$ is typable under context $[\![x\colon \hat{}\mathsf{end}, y\colon \mathsf{end}]\!] = x\colon (*?(\mathsf{end}, \mathsf{end}), *!(\mathsf{end}, \mathsf{end})), y\colon (\mathsf{end}, \mathsf{end})$, whereas P is not typable under $x\colon \hat{}\mathsf{end}, y\colon \mathsf{end}$.

5.2. *Embedding the pi calculus with accept/request primitives*

This section shows that the pi calculus with accept and request primitives (Honda et al., 1998) can be embedded in our system. With respect to the original formulation, the calculus considered here does not provide for label selection and branching. On the other hand it features process replication in place of recursion. The syntax of processes and types is in Figure 5. In addition to the set of variables introduced in Section 3, we consider a new (disjoint) *countable* set of channels, ranged over by h, k. Synchronisation names are taken from the set of variables. The new process constructors are exactly accept and request: they allow to install new sessions via synchronisation on names. We remove productions $x(x).P$ and $\overline{x}\, v.P$ in the syntax of processes, and replace them by input/output processes on channels: $k(x).P$ and $\overline{k}\, v.P$. We also introduce a new restriction operator $(\nu k)P$ for channels, in addition to that for variables $(\nu x)P$.

The bindings in the language are introduced by the parenthetical constructs: accept, request, input and restriction. The set of free channels, as well as substitution of variables and boolean values for variables in a given process $P[v/x]$, is defined as usual. Notice that substitution is not defined for channels.

The reduction relation, denoted $\rightarrow_{\mathsf{r}}$, is defined in Figure 5. Rule [R-Link] establishes a new session k when an accept process meets a request process on a given name x. Channel k, bound in both processes, becomes free in each process, but still bound in the parallel composition. Rule [R-Com] is taken from Figure 1, only that the subject of communication are channels and the objects' values do not comprise channels. The last rule, [R-Pass], allows delegating a channel from an output process to an input process. The channel h in the output process must not be free in the input process, for no substitution is performed in channel passing.

Types are (informally) divided in sorts and session types. Session types are as in Figure 4; the definition of the dual function, $\overline{S}$, from session types to session types is defined in Section 5.1. Sorts include the type bool for boolean values as well as types

New syntactic forms (extends Figures 1 and 4)

$P ::= \ldots$		Processes:	$v ::= \ldots$		Values:
	$\mathsf{request}\ x(k).P$	session request		k	channels
	$\mathsf{accept}\ x(k).P$	session acceptance	$T ::=$		Types:
	$\overline{k}\,v.P$	output		bool	boolean
	$k(x).P$	input		S	session type
	$k(k).P$	channel input		$\langle S, S\rangle$	shared type
	$(\nu k)P$	channel restriction		$\bot_S$	used channel

$\boxed{P \to_{\mathsf{r}} P}$ *New reduction rules (extends Figure 1)*

$$\mathsf{accept}\ x(k).P \mid \mathsf{request}\ x(k).Q \to_{\mathsf{r}} (\nu k)(P \mid Q) \qquad \text{[R-Link]}$$

$$\overline{k}\,v.P \mid\ k(x).Q \to_{\mathsf{r}} P \mid Q[v/x] \quad \text{if } v = y, \mathsf{true}, \mathsf{false} \qquad \text{[R-Com]}$$

$$\overline{k}\,h.P \mid\ k(h).Q \to_{\mathsf{r}} P \mid Q \qquad \text{[R-Pass]}$$

$\boxed{\Delta \circ \Delta = \Delta}$ *Linear context composition*

$$\emptyset \circ \Delta = \Delta \qquad \frac{\Delta_1 \circ \Delta_2 = \Delta_3}{(\Delta_1, k\colon S) \circ (\Delta_2, k\colon \overline{S}) = \Delta_3, k\colon \bot_S} \qquad \frac{\Delta_1 \circ \Delta_2 = \Delta_3 \qquad k \notin \Delta_2}{(\Delta_1, k\colon T) \circ \Delta_2 = \Delta_3, k\colon T}$$

$\boxed{\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta}$ *Typing rules for processes*

$$\frac{\Gamma \vdash x\colon \langle S, \overline{S}\rangle \qquad \Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon S}{\Gamma \vdash_{\mathsf{r}} \mathsf{accept}\ x(k).P \triangleright \Delta} \qquad \frac{\Gamma \vdash x\colon \langle S, \overline{S}\rangle \qquad \Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon \overline{S}}{\Gamma \vdash_{\mathsf{r}} \mathsf{request}\ x(k).P \triangleright \Delta}$$

[T-Acc] [T-Req]

$$\frac{\Gamma \vdash_{\mathsf{r}} v\colon T \qquad \Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon S}{\Gamma \vdash_{\mathsf{r}} \overline{k}\,v.P \triangleright \Delta, k\colon !T.S} \qquad \frac{\Gamma, x\colon T \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon S}{\Gamma \vdash_{\mathsf{r}} k(x).P \triangleright \Delta, k\colon ?T.S}$$

[T-Send] [T-Rcv]

$$\frac{\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon S_2}{\Gamma \vdash_{\mathsf{r}} \overline{k}\,h.P \triangleright \Delta, k\colon !S_1.S_2, h\colon S_1} \qquad \frac{\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, h\colon S_1, k\colon S_2}{\Gamma \vdash_{\mathsf{r}} k(h).P \triangleright \Delta, k\colon ?S_1.S_2}$$

[T-Thr] [T-Cat]

$$\frac{\Delta \text{ completed}}{\Gamma \vdash_{\mathsf{r}} \mathbf{0} \triangleright \Delta} \qquad \frac{\Gamma \vdash_{\mathsf{r}} P_1 \triangleright \Delta_1 \qquad \Gamma \vdash_{\mathsf{r}} P_2 \triangleright \Delta_2}{\Gamma \vdash_{\mathsf{r}} P_1 \mid P_2 \triangleright \Delta_1 \circ \Delta_2} \qquad \frac{\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta \qquad \Delta \text{ completed}}{\Gamma \vdash_{\mathsf{r}} !P \triangleright \Delta}$$

[T-Inact] [T-Par] [T-Repl]

$$\frac{\Gamma, x\colon \langle S, \overline{S}\rangle \vdash_{\mathsf{r}} P \triangleright \Delta}{\Gamma \vdash_{\mathsf{r}} (\nu x)P \triangleright \Delta} \qquad \frac{\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon \bot_S}{\Gamma \vdash_{\mathsf{r}} (\nu k)P \triangleright \Delta}$$

[T-Res] [T-ResC]

Fig. 5. Accept/request pi calculus

for names of the form $\langle S_1, S_2\rangle$. In such a type, session type S_1 types the accept end of the name, whereas session type S_2 types the $\mathsf{request}$ end. We are only interested in types where S_1 is the dual of S_2. Type $\bot_S$ denotes a channel on which no further interaction is possible. To guide our encoding, we decorate $\bot$ with the type from which it was generated. We explain the mechanism below.

The type system uses two kinds of contexts: *shared contexts* Γ mapping variables x to sorts (types of the form bool and $\langle S_1, S_2\rangle$), and *linear contexts* Δ mapping channels k to session types S and to $\bot_S$. For linear contexts we define in Figure 5 an operation of *composition* that conjoins two contexts on the disjoint parts, and assigns a $\bot_S$ type to a

channel that is typed at dual types, S and $\overline{S}$, in the two incoming contexts. Composition is not defined when the types for a same channel in the two input contexts are not dual to each other.

The typing system comprises the rules for value typing $\Gamma \vdash v\colon T$ taken from Figure 3, as well as the rules for process typing, $\Gamma \vdash P \triangleright \Delta$ in Figure 5. The rules for value typing feature a pre-condition $\mathsf{un}(\Gamma)$, which we ignore in this system for linear values now belong in context Δ (alternatively we may define $\mathsf{un}(\Gamma)$ as true for all contexts Γ). The typing rule for accept reads the type $\langle S, \overline{S}\rangle$ of value x from the shared context Γ and places a new entry $k\colon S$ in the linear context of the continuation process P. The rule for request is similar, only that the dual type $\overline{S}$ is taken into consideration. The rules for channel passing, [T-Thr] and [T-Cat], are similar to their counterparts in the polarity system, namely rules [T-OutS] and [T-InS] in Figure 4. The rules for value passing, [T-Send] and [T-Rcv], are also similar, only that the value sent, v, or received, x, is read or written on the shared context Γ. The rules for inaction and for replication are similar to those in Figure 4; predicate *completed* is defined in Section 5.1: it is true of contexts containing only end as session types. The rule for parallel composition makes use of the composition operator $\circ$ and marks as "used" (session type $\bot_S$) those channels that can be found at type S in one thread and at type $\overline{S}$ in the other, thus preventing further interaction on such channels. Finally, the rules for scope restriction, [T-Res] and [T-ResC], introduce in the appropriate contexts types for the bound names: a name type $\langle S, \overline{S}\rangle$ in the shared context for name binding, and the $\bot_S$ type in the linear context for channel binding. In the latter case, we see that only channels that have been used to end can be restricted.

We now look at the encoding. For processes we have to encode the constructors in Figure 5, plus those inherited from Figure 4. Given that both the sets of variables and channels are enumerable, we can easily map the two sets into the set of variables; we leave the application of these maps implicit in the sequel. Proceeding in this way, the only cases left are accept and request. We choose request to create a new channel k and use name x to convey its identity to potential accept parties. Accept processes, in turn, receive the newly created channel via a conventional input on name x. We could have chosen the accept party to create the channel; accept and request are symmetric.

$$[\![\mathsf{accept}\ x(k).P]\!] = x(k).[\![P]\!] \qquad\qquad [\![\mathsf{request}\ x(k).P]\!] = (\nu k)\overline{x}\,k.[\![P]\!]$$

We also have to encode contexts, which we do, point-wise, by using an encoding for types. We use an auxiliary encoding $\{\![S]\!\}$ to map session types into session types. Types T are encoded using the $[\![T]\!]$ function. A name type $\langle S_1, S_2\rangle$ is translated into a type describing the two channel ends; recall that $*?T$ abbreviates the recursive end point type $\mu a?T.a$, and similarly for $*!T$. Session type $\bot_S$ is translated into type $(\{\![S]\!\}, \{\![\overline{S}]\!\})$. Note that the decoration S of $\bot_S$ is essential for the reconstruction of the type. A type $\bot_S$ originates from types S and $\overline{S}$ in a process $\overline{k}\,v \mid x(y)$, say, and prevents its composition with, say, a process $\overline{k}\,u$. In our system it is the linear nature of type $(\{\![S]\!\}, \{\![\overline{S}]\!\})$ that prevents the composition. The input type is translated into a type where only the input end point can be used (the other is recorded as end, allowing no interaction); and conversely for the output. The four remaining cases are the identity function for end, bool,

and the type variable a, and $\mu a.[\![S]\!]$ for $[\![\mu a.S]\!]$.

$$\{\!|?T.S|\!\} = \mathsf{lin}?[\![T]\!].\{\!|S|\!\} \qquad \{\!|!T.S|\!\} = \mathsf{lin}![\![T]\!].\{\!|S|\!\}$$
$$[\![\langle S_1, S_2\rangle]\!] = (*?[\![S_1]\!], *![\![S_2]\!]) \qquad [\![\bot_S]\!] = (\{\!|S|\!\}, \{\!|\overline{S}|\!\})$$
$$[\![?T.S]\!] = (\{\!|?T.S|\!\}, \mathsf{end}) \qquad [\![!T.S]\!] = (\mathsf{end}, \{\!|!T.S|\!\})$$

The main result of this section follows.

Theorem 5.2 (Accept/request-pi to pi correspondence).

1 If $\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta$ then $[\![\Gamma]\!], [\![\Delta]\!] \vdash [\![P]\!]$;
2 If $P \to_{\mathsf{r}} Q$ then $[\![P]\!] \to [\![Q]\!]$.

Proof. (1) We proceed by induction on the structure of the derivation of $\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta$, using unrestricted weakening (Lemma 4.3) to add sessions typed with $(\mathsf{end}, \mathsf{end})$ whenever such sessions are in Δ and not in a context of its derivation. To illustrate, we sketch three cases.

For [T-Req] we have $\Gamma \vdash \mathsf{request}\ x(k).P \triangleright \Delta$ with $a\colon \langle S, \overline{S}\rangle \in \Gamma$ and $\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon \overline{S}$. Therefore $[\![\Gamma]\!](a) = (*?[\![S]\!], *![\![S]\!])$. By the induction hypothesis $[\![\Gamma, \Delta, k\colon \overline{S}]\!] \vdash [\![P]\!]$. By applying [T-OutR] we infer $[\![\Gamma, \Delta]\!], k : (\{\!|S|\!\}, \{\!|\overline{S}|\!\}) \vdash \overline{a}\, k.[\![P]\!]$. An application of [T-Res] give us the desired result, $[\![\Gamma, \Delta]\!] \vdash (\nu k)\overline{a}\, k.[\![P]\!]$.

For [T-Thr], let $\Gamma \vdash_{\mathsf{r}} \overline{k}\, k'.P \triangleright \Delta, k\colon !S.S', k'\colon S$ be inferred from $\Gamma \vdash_{\mathsf{r}} P \triangleright \Delta, k\colon S'$. By induction we have $[\![\Gamma, \Delta, k\colon S']\!] \vdash [\![P]\!]$. We use weakening (Lemma 4.3) and infer $[\![\Gamma, \Delta, k\colon S']\!], k'\colon (\mathsf{end}, \mathsf{end}) \vdash [\![P]\!]$. To conclude the case, we apply [T-OutR] to obtain $[\![\Gamma, \Delta]\!], k\colon (\mathsf{end}, \mathsf{lin}\,!\{\!|S|\!\}.\{\!|S'|\!\}),\ k'\colon [\![S]\!] \vdash \overline{k}\, k'.[\![P]\!]$.

Lastly, take case [T-Par], and let $\Gamma \vdash_{\mathsf{r}} P \mid P' \triangleright \Delta \circ \Delta'$ inferred from $\Gamma \vdash_{\mathsf{r}} P_1 \triangleright \Delta_1$ and $\Gamma \vdash_{\mathsf{r}} P_2 \triangleright \Delta_2$. By the induction hypothesis $[\![\Gamma, \Delta_1]\!] \vdash [\![P]\!]_1$ and $[\![\Gamma, \Delta_2]\!] \vdash [\![P_2]\!]$. To conclude by applying [T-Par], two considerations are in order. First, since $[\![\Gamma]\!]$ is unrestricted, we have $[\![\Gamma]\!] \circ [\![\Gamma]\!] = [\![\Gamma]\!]$. Second, we can make the domains of the contexts $[\![\Delta_1]\!]$ and $[\![\Delta_2]\!]$ equal by adding entries $(\mathsf{end}, \mathsf{end})$, thus generating two contexts Δ' from $[\![\Delta_1]\!]$ and Δ'' from $[\![\Delta_2]\!]$. Then, it is easy to show that $\Delta' \circ \Delta''$ is defined and equal to $[\![\Delta]\!]$. We may then apply [T-Par] and infer $[\![\Gamma, \Delta]\!] \vdash [\![P_1 \mid P_2]\!]$.

(2) By induction on the structure of the derivation of $P \to_{\mathsf{r}} P'$. The interesting case is [R-Link], and is established by $a(k).[\![P]\!] \mid (\nu k)\overline{a}\, k.[\![Q]\!] \equiv (\nu k)(a(k).[\![P]\!] \mid \overline{a}\, k.[\![Q]\!])$, by [R-Com]: $a(k).[\![P]\!] \mid \overline{a}\, k.[\![Q]\!] \to [\![P]\!][k/k] \mid [\![Q]\!]$, by [R-Res]: $(\nu k)(a(k).[\![P]\!] \mid \overline{a}\, k.[\![Q]\!]) \to (\nu k)([\![P]\!][k/k] \mid [\![Q]\!])$, and by [R-Struct]: $a(k).[\![P]\!] \mid (\nu k)\overline{a}\, k.[\![Q]\!] \to (\nu k)([\![P]\!][k/k] \mid [\![Q]\!])$. The result then follows by noting that $[\![P]\!][k/k] = [\![P]\!]$. Cases [R-Com],[R-Pass] are analogous; the remaining cases follow straightforwardly from induction. □

The converse does not hold. Take for P the process $\overline{k}\, h \mid k(l).\overline{h}\,\mathsf{true}.l(x)$. It should be easy to see that P does not reduce, whereas $[\![P]\!] = P$ does. Notice that the sub-process $k(l).\overline{h}\,\mathsf{true}.l(x)$ cannot be renamed into a process of the form $k(h).Q$, as required by rule [R-Pass], for h is free in $\overline{h}\,\mathsf{true}.l(x)$. It is also the case that $[\![P]\!]$ is typable under context $[\![h\colon S, k\colon \langle S,\ \overline{S}\rangle]\!]$ where S is the type $!\mathsf{bool}.\mathsf{end}$, whereas P is not typable under context $h\colon S, k\colon \langle S,\ \overline{S}\rangle$.

New syntactic forms (extends Figure 2)

c ::=	Capabilities:	T ::=	Types:
i	input	$q\,c\,T$	channel
o	output	bool	boolean
io	input and output		

$\boxed{T + T = T}$ *Combination of types*

$$\mathsf{bool} + \mathsf{bool} = \mathsf{bool} \qquad \mathsf{un}\,c_1\,T + \mathsf{un}\,c_2\,T = \mathsf{un}\,(c_1 \cup c_2)\,T \qquad \mathsf{lin}\,\mathsf{i}\,T + \mathsf{lin}\,\mathsf{o}\,T = \mathsf{lin}\,\mathsf{io}\;T$$

$\boxed{\Gamma + \Gamma = \Gamma}$ *Combination of contexts*

$$\emptyset + \Gamma = \Gamma \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3}{(\Gamma_1, x\colon T_1) + (\Gamma_2, x\colon T_2) = \Gamma_3, x\colon T_1 + T_2} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \qquad x \notin \Gamma_2}{(\Gamma_1, x\colon T) + \Gamma_2 = \Gamma_3, x\colon T}$$

$\boxed{\Gamma \vdash_{\mathsf{l}} P}$ *Typing rules for processes (extends Figure 3)*

$$\frac{\Gamma_1 \vdash_{\mathsf{l}} P_1 \qquad \Gamma_2 \vdash_{\mathsf{l}} P_2}{\Gamma_1 + \Gamma_2 \vdash_{\mathsf{l}} P_1 \mid P_2} \qquad \frac{\Gamma, x\colon q\,\mathsf{io}\;T \vdash_{\mathsf{l}} P}{\Gamma \vdash_{\mathsf{l}} (\nu x)P} \qquad \text{[T-Par] [T-Res]}$$

$$\frac{\Gamma, y\colon T \vdash_{\mathsf{l}} P}{\Gamma + x\colon q\,\mathsf{i}T \vdash_{\mathsf{l}} x(y).P} \qquad \frac{\Gamma \vdash_{\mathsf{l}} P}{\Gamma + x\colon q\,\mathsf{o}T + v\colon T \vdash_{\mathsf{l}} \overline{x}\,v.P} \qquad \text{[T-In] [T-Out]}$$

Fig. 6. Linear pi calculus

5.3. *Embedding the linear pi calculus*

In this section we analyse (a synchronous variant of) the linear pi calculus (Kobayashi et al., 1999) and provide a typing-preserving encoding into our system.

The syntax of linear pi processes and the reduction relation are described in Figure 1. Figure 6 defines the syntax of types and the typing rules for linear pi processes. Types now have the form $q\,c\,T$ where q is a lin/un qualifier introduced in Figure 2, and c is a capability describing the input, output or input-output nature of the type.

The linear discipline is imposed by way of a type combination operation, defined in Figure 6. The combination of two unrestricted types conjoins the capabilities as follows, by viewing capabilities as sets.

$$\mathsf{i} = \{\mathsf{i}\} \qquad \mathsf{o} = \{\mathsf{o}\} \qquad \mathsf{io} = \{\mathsf{i}, \mathsf{o}\}$$

The combination of un types is only defined for two types, of i and of o capabilities; the result is a type of io capability. There is also a rule symmetrical to the last, namely $\mathsf{lin}\,\mathsf{o}\,T + \mathsf{lin}\,\mathsf{i}\,T = \mathsf{lin}\,\mathsf{io}\;T$. The operator is then extended point-wise to typing contexts.

The typing system for the linear pi-calculus is defined by the rules in Figure 3, where rules [T-Par], [T-Res], [T-In] and [T-Out] are replaced by those in Figure 6. Rule [T-Out] is an adaptation of that in (Kobayashi et al., 1999) to the synchronous setting: we let the continuation be typed with context Γ while in the original paper the premise to the rule is $\mathsf{un}(\Gamma)$ since the (absent) continuation behaves as $\mathbf{0}$. We have also adapted rule [T-Res] to require that the restricted channel uses both capabilities; the original system allows processes of the form $(\nu x)\overline{x}\,\mathsf{true}$ to be typed by assigning type $\mathsf{lin}\,\mathsf{o}\,\mathsf{bool}$ to channel x.

Our system forces a linear behaviour to be described by a type $(\mathsf{lin}\,!\mathsf{bool}, \mathsf{lin}\,?\mathsf{bool})$ which cannot be used to type process $\overline{x}\,\mathsf{true}$, a process that never exercises the input capability of channel x. The variant of the rule we use allows a typing correspondence between the two systems.

A compositional encoding of linear types is defined below. Unrestricted end points are encoded into recursive types; once again recall that $*!T$ abbreviates the type $\mu a.!T.a$, and similarly for $*?T$. Linear end points are encoded into a linear output/input followed by end, meaning that the channel cannot be further used for input or output. An input type is encoded as a channel type composed of an input end point type and an end type. The output type is similar. Input/output types are encoded as channel types, one end for the output, the other for the input capability.

$$\begin{aligned}
[\![\mathsf{lin}\,\mathsf{i}T]\!] &= (\mathsf{lin}?[\![T]\!].\mathsf{end}, \mathsf{end}) & [\![\mathsf{lin}\,\mathsf{o}T]\!] &= (\mathsf{end}, \mathsf{lin}![\![T]\!].\mathsf{end})\\
[\![\mathsf{un}\,\mathsf{i}T]\!] &= (*?[\![T]\!], \mathsf{end}) & [\![\mathsf{un}\,\mathsf{o}T]\!] &= (\mathsf{end}, *![\![T]\!])\\
[\![\mathsf{lin}\,\mathsf{io}\,T]\!] &= (\mathsf{lin}?[\![T]\!].\mathsf{end}, \mathsf{lin}![\![T]\!].\mathsf{end}) & [\![\mathsf{un}\,\mathsf{io}\,T]\!] &= (*?[\![T]\!], *![\![T]\!])\\
[\![\mathsf{bool}]\!] &= \mathsf{bool}
\end{aligned}$$

For instance, taking for T the type $(*!\mathsf{bool}, *?\mathsf{bool})$, the linear pi type $\mathsf{lin}\,\mathsf{i}(\mathsf{lin}\,\mathsf{io}\,(\mathsf{un}\,\mathsf{io}\,\mathsf{bool}))$ is mapped into the type $(\mathsf{lin}\,?(\mathsf{lin}\,!T.\mathsf{end}, \mathsf{lin}\,?T.\mathsf{end}).\mathsf{end}, \mathsf{end})$.

The main result of this section establishes the correspondence between the two systems.

Theorem 5.3 (Linear-pi to pi correspondence). $\Gamma \vdash_{\mathsf{l}} P$ if and only if $[\![\Gamma]\!] \vdash P$.

Proof. For the left to the right direction, we proceed by induction on the structure of the derivation of $\Gamma \vdash_{\mathsf{l}} P$. The proof relies on unrestricted weakening (Lemma 4.3). The rule for context combination in Figure 6 permits to combine two unrestricted types with opposite capabilities. We recover this by weakening each type with the missing capability, in order to make the context splitting operation of Figure 3 defined. To illustrate, consider case [T-In] and let $\Gamma + x\colon q\,\mathsf{i}S \vdash_{\mathsf{l}} x(y).P$ be inferred from $\Gamma, y\colon S \vdash_{\mathsf{l}} P$. The induction hypothesis is $[\![\Gamma, y\colon S]\!] \vdash P$. We have three cases corresponding to (i) $x\colon \mathsf{lin}\,\mathsf{o}S \in \Gamma$, (ii) $x\colon \mathsf{un}\,cS \in \Gamma$, and (iii) x not in Γ. In (i) we can directly apply [T-InR], since the splitting of $[\![\mathsf{lin}\,\mathsf{o}S]\!]$ and $[\![\mathsf{lin}\,\mathsf{i}S]\!]$ is defined. In (ii) we have two sub-cases corresponding to $c = \mathsf{i}$ or $c \neq \mathsf{i}$. In the first case $[\![\mathsf{un}\,cS]\!] \circ [\![\mathsf{un}\,cS]\!]$ is defined, while in the second case it is not. We need to weaken the judgement $[\![\Gamma, y\colon S]\!] \vdash P$ to $[\![\Gamma, y\colon S]\!] \setminus x, x\colon (\mathsf{un}\,*![\![S]\!], \mathsf{un}\,*?[\![S]\!]) \vdash P$. We can now apply [T-InR] and close the result. Case (iii) is analogous, while by weakening we infer the judgement $[\![\Gamma, y\colon S]\!], x\colon (\mathsf{un}\,*![\![S]\!], \mathsf{un}\,*?[\![S]\!]) \vdash P$. In rule [T-Par] we use weakening and add entries $x\colon (\mathsf{end}, \mathsf{end})$ in order to make the domain of the contexts Γ_1, Γ_2 of the hypothesis equal, so that we can apply context splitting in rule [T-Par] of Figure 3. The remaining cases are similar.

For the right to the left direction, we proceed by induction on the structure of the derivation of $[\![\Gamma]\!] \vdash P$. To illustrate, we sketch a couple of cases. We use the notation $\Gamma \setminus x$ to indicate the context obtained by removing the entry for x in Γ.

When the derivation ends with rule [T-InL], we have $[\![\Gamma]\!] \vdash x(y).P$ inferred from $\Gamma' \vdash x\colon (q?T'.S', S'')$ and $\Delta, y\colon T' \vdash P$ where $\Delta = \Gamma'' + x\colon (S', S'')$, and $[\![\Gamma]\!] = \Gamma' \circ \Gamma''$. We infer that $\Gamma' = \Gamma_1, x\colon (q?T'.S', S'')$ with $\mathsf{un}(\Gamma_1)$, and $\Gamma'' = \Gamma_2, x\colon (S_1, S_2)$. Thus

$\Delta = \Gamma_2, x\colon (R_1, R_2)$ where $R_1 = S_1 \circ S'$ and $R_2 = S_2 \circ S''$. Distinguish the two cases $q = \mathsf{lin}$ and $q = \mathsf{un}$. When $q = \mathsf{lin}$ we know that $[\![\Gamma]\!](x) = (\mathsf{lin}?T'.S', R_2)$ and $S_1 = \mathsf{end}$. A case analysis on $[\![\cdot]\!]$ shows that two cases arise for $\Gamma(x)$: (a) $\Gamma(x) = \mathsf{lin}\,\mathsf{i}T$ with $T' = [\![T]\!]$, $S' = \mathsf{end}$ and $R_2 = \mathsf{end}$, or (b) $\Gamma(x) = \mathsf{lin}\,\mathsf{io}\,T$ with $T' = [\![T]\!]$, $S' = \mathsf{end}$, and $R_2 = \mathsf{lin}!T'.\mathsf{end}$. From the rules for context splitting we know that $S_1 = \mathsf{end}$. Thus in case (a) we have $\Delta = \Gamma_2, x\colon (\mathsf{end}, \mathsf{end})$ while in case (b) we have $\Delta = \Gamma_2, x\colon (\mathsf{end}, \mathsf{lin}!T'.\mathsf{end})$. When $q = \mathsf{un}$ we know that $[\![\Gamma]\!](x) = (\mathsf{un}?T'.S', R_2)$. A case analysis on $[\![\cdot]\!]$ shows that two cases arise for $\Gamma(x)$: (c) $\Gamma(x) = \mathsf{un}\,\mathsf{i}T$ with $T' = [\![T]\!]$, $S' = \mathsf{un}?T'.S'$ and $R_2 = \mathsf{end}$, or (d) $\Gamma(x) = \mathsf{un}\,\mathsf{io}\,T$ with $T' = [\![T]\!]$, $S' = \mathsf{un}?T'.S'$, and $R_2 = \overline{S'}$. In case (c) we have $\Delta = \Gamma_2, x\colon (S', \mathsf{end})$ while in case (d) we have $\Delta = \Gamma_2, x\colon (S', \overline{S'})$.

To apply the induction hypothesis we rely on the two following results.

$$[\![\Gamma]\!] \setminus x = \Gamma_2 \circ \mathcal{U}([\![\Gamma]\!] \setminus x) \tag{1}$$

The result is proved by induction on the length of Γ. The case $\mathrm{dom}(\Gamma) = \{x\}$ is clear. Otherwise assume $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n, x\}$, and let the induction hypothesis be $[\![\Gamma]\!] \setminus x_n \setminus x = \Gamma_2 \circ \mathcal{U}([\![\Gamma]\!] \setminus x_n \setminus x)$. A case analysis shows that the following cases arise for $\Gamma_2(x_n)$: (i) $\Gamma_2(x_n) = [\![\Gamma]\!](x_n)$ and $\mathcal{U}([\![\Gamma]\!](x_n)) = (\mathsf{end}, \mathsf{end})$, or (ii) $\Gamma_2(x_n) = [\![\Gamma]\!](x_n)$ and $\mathcal{U}([\![\Gamma]\!](x_n)) = [\![\Gamma]\!](x_n)$ and $\mathsf{un}(\Gamma_2(x_n))$, or (iii) $\Gamma_2(x_n) = (\mathsf{end}, \mathsf{end})$ and $[\![\Gamma]\!](x_n) = (*?[\![T_n]\!], \mathsf{end})$, or (iv) $\Gamma_2(x_n) = (\mathsf{end}, \mathsf{end})$ and $[\![\Gamma]\!](x_n) = (\mathsf{end}, *![\![T_n]\!])$, or (v) $\Gamma_2(x_n) = (*?[\![T_n]\!], \mathsf{end})$ and $[\![\Gamma]\!](x_n) = (*?[\![T_n]\!], *![\![T_n]\!])$, or (vi) $\Gamma_2(x_n) = (\mathsf{end}, *![\![T_n]\!])$ and $[\![\Gamma]\!](x_n) = (*?[\![T_n]\!], *![\![T_n]\!])$, or (vii) $\Gamma_2(x_n) = (\mathsf{end}, \mathsf{end})$ and $[\![\Gamma]\!](x_n) = (*?[\![T_n]\!], *![\![T_n]\!])$. In all cases we obtain $[\![\Gamma]\!](x_n) = \Gamma_2(x_n) \circ \mathcal{U}([\![\Gamma]\!](x_n))$; we thus conclude that $[\![\Gamma]\!] \setminus x = \Gamma_2 \circ \mathcal{U}([\![\Gamma]\!] \setminus x)$.

$$[\![\Gamma]\!], x\colon (\mathsf{end}, \mathsf{end}) \vdash P \text{ implies } [\![\Gamma]\!] \vdash P \tag{2}$$

The implication is deduced by using strengthening (Lemma 4.5), since $(\mathsf{end}, \mathsf{end})$ not in the image of $[\![\cdot]\!]$ implies $x \notin \mathrm{fv}(P)$.

To complete the [T-InL] case, we weaken the hypothesis $\Delta, y\colon T' \vdash P$ by using Corollary 4.4 and infer $\Gamma_2 \circ (\mathcal{U}[\![\Gamma]\!] \setminus x), x\colon (R_1, R_2), y\colon T' \vdash P$. We then apply (1) and infer $[\![\Gamma]\!] \setminus x, x\colon (R_1, R_2), y\colon T' \vdash P$. In case (a) we use (2) and infer $[\![\Gamma]\!] \setminus x, y\colon [\![T]\!] \vdash P$. The induction hypothesis is $\Gamma \setminus x, y\colon T \vdash_{\mathsf{I}} P$. In case (b) the induction hypothesis is $\Gamma \setminus x, x\colon \mathsf{lin}\,\mathsf{o}T, y\colon T \vdash_{\mathsf{I}} P$. In case (c) the induction hypothesis is $\Gamma \setminus x, x\colon \mathsf{un}\,\mathsf{i}T, y\colon T \vdash_{\mathsf{I}} P$. In case (d) the induction hypothesis is $\Gamma \setminus x, x\colon \mathsf{un}\,\mathsf{io}\,T, y\colon T \vdash_{\mathsf{I}} P$. In all cases (a)–(d) we conclude by applying the typing rule for input, [T-In], and we are done.

As further example, consider the case [T-Par]: $[\![\Gamma]\!] \vdash P \mid Q$ with $\Gamma' \vdash P$, $\Gamma'' \vdash Q$, and $[\![\Gamma]\!] = \Gamma' \circ \Gamma''$. We use weakening and obtain: $(*)$ $\Delta_1 \vdash P$ and $(**)$ $\Delta_2 \vdash Q$ where $\Delta_1 = \Gamma' \circ \mathcal{U}([\![\Gamma]\!])$ and $\Delta_2 = \Gamma'' \circ \mathcal{U}([\![\Gamma]\!])$. Next, we find Γ_1, Γ_2 such that $[\![\Gamma_1]\!] = \Delta_1$ and $[\![\Gamma_2]\!] = \Delta_2$. By induction hypothesis we have $\Gamma_1 \vdash_{\mathsf{I}} P$ and $\Gamma_2 \vdash_{\mathsf{I}} Q$. We apply [T-Par] and infer that if $\Gamma_1 + \Gamma_2$ is defined then $\Gamma_1 + \Gamma_2 \vdash_{\mathsf{I}} P \mid Q$. To conclude, we need to show that $\Gamma_1 + \Gamma_2$ is defined and equal to Γ: that is, $\Gamma = \Gamma_1 + \Gamma_2$. First, note that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma_1 + \Gamma_2)$. To see the left to the right inclusion, assume $\Gamma(x) = T$. The case $\mathsf{un}(T)$ is straightforward, since $\Gamma(x) = \Gamma_1 + \Gamma_2(x)$ and $\Delta_i(x) = \mathcal{U}([\![\Gamma]\!])(x) = [\![\Gamma(x)]\!]$, for $i = 1, 2$. Take $T = \mathsf{lin}\,cT'$. When $c = \mathsf{io}$ we know that or $\Gamma'(x) = [\![T]\!]$ and $\Gamma''(x) = (\mathsf{end}, \mathsf{end})$, or vice-versa. Suppose $\Gamma'(x) = [\![T]\!]$; we have $\Delta_1(x) = [\![T]\!]$ and $\Delta_2(x) = (\mathsf{end}, \mathsf{end})$. We conclude that $\Gamma_1 + \Gamma_2(x) = T$, as desired. The case $\Gamma''(x) = [\![T]\!]$ is analogous. Consider

now the case $T = \mathsf{lin}\,\mathsf{io}\,T'$. When $\Gamma'(x) = [\![T]\!]$ and $\Gamma''(x) = (\mathsf{end}, \mathsf{end})$, or $\Gamma'(x) = (\mathsf{end}, \mathsf{end})$ and $\Gamma''(x) = [\![T]\!]$ we proceed as above. Otherwise, we can have $\Gamma'(x) = (\mathsf{lin}\,?[\![T']\!].\mathsf{end}, \mathsf{end})$ and $\Gamma''(x) = (\mathsf{end}, \mathsf{lin}\,![\![T']\!].\mathsf{end})$, or vice-versa. In both cases we have $\Delta_1(x) = \Gamma'(x)$ and $\Delta_2(x) = \Gamma''(x)$. We conclude by noting that $\Gamma_1 + \Gamma_2(x) = T$. To see the right to the left inclusion, assume $\Gamma_1 + \Gamma_2(x) = T$. As before, the case $\mathsf{un}(T)$ is immediate since we have $\Delta_i(x) = \mathcal{U}([\![\Gamma]\!])(x) = [\![\Gamma(x)]\!]$. Thus $\Gamma(x) = T$, as desired. When $T = \mathsf{lin}\,cT'$ and $c \in \mathsf{i}, \mathsf{o}$, we know that $\Gamma'(x) = [\![T]\!]$ and $x \notin \mathrm{dom}(\Gamma'')$, or vice-versa. Thus $\Gamma' \circ \Gamma''(x) = [\![T]\!]$, and in turn $\Gamma(x) = T$, as desired. The case $T = \mathsf{lin}\,\mathsf{io}\,T'$ is similar: we have four options for $\Gamma'(x)$ and $\Gamma''(x)$ but all lead to $\Gamma' \circ \Gamma''(x) = [\![T]\!]$; from this we conclude that $\Gamma(x) = T$, and we are done. □

6. Concluding remarks

We presented a session type system for the pi calculus, the version in (Milner, 1992), where types for channels are pairs of end point types. End point types, in turn, are session types annotated with lin/un qualifiers. Session types may include choice, usually in the form of branch/select constructs. For the sake of simplicity we decided to omit such constructs; their incorporation should raise no difficulty, cf. (Vasconcelos, 2012).

We showed that the type system here proposed handles processes other type systems fail to type, including (Gay and Hole, 2005; Giunti and Vasconcelos, 2010; Honda et al., 1998; Kobayashi et al., 1999; Vasconcelos, 2012), cf. Section 2. We also showed that our type system types all processes three other type systems manage to type (Gay and Hole, 2005; Honda et al., 1998; Kobayashi et al., 1999), cf. Section 5. We believe that similar techniques would allow the embedding of further type systems, including (Giunti and Vasconcelos, 2010; Pierce and Sangiorgi, 1996; Vasconcelos, 2012).

A question arises on the existence of a sound and complete algorithm for the system proposed in Section 3. While the idea of splitting types and contexts is clear and concise, the inherent non-determinism contained in the formulation makes a "forward only" implementation probably unfeasible. A concrete problem emerges from channel types where both end point types are of the same nature (input or output), for such types match both the left and the right rules in the type system. A sound, but not complete, algorithm exists for an earlier version of the system here proposed (Giunti, 2011). We believe that the algorithm could be adapted and still type enough interesting processes. The investigation of a sound and complete algorithm for this system is left for future work.

Acknowledgements. We thank Dimitris Mostrous and the anonymous referees for insightful comments. The work was partially supported by projects PTDC/EIA-CCO/117513/2010 and PTDC/EIA-CCO/122547/2010. The work of Marco Giunti has been done during the period that the author spent at LIX, École Polytechnique, with the support of an ERCIM postdoc fellowship. The author would like to thank INRIA and ERCIM for such an opportunity.

References

Baltazar, P., Caires, L., Vasconcelos, V. T., and Vieira, H. T. (2013). A type system for flexible role assignment in multiparty communicating systems. In *TGC'12*, LNCS. Springer. To appear.

Bruni, R. and Mezzina, L. G. (2008). Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST*, pages 100–115. Springer.

Caires, L. and Vieira, H. (2010). Conversation types. *Theoretical Computer Science*, 411(51–52):4399–4440.

Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S., and Giachino, E. (2009). Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science*, 410(2-3):142–167.

Coppo, M., Dezani-Ciancaglini, M., and Yoshida, N. (2007). Asynchronous session types and progress for object-oriented languages. In *FMOODS*, volume 4468 of *LNCS*, pages 1–31. Springer.

Cruz-Filipe, L., Lanese, I., Martins, F., Ravara, A., and Vasconcelos, V. T. (2008). Behavioural theory at work: program transformations in a service-centred calculus. In *FMOODS*, volume 5051 of *LNCS*, pages 59–77. Springer.

Dezani-Ciancaglini, M. and de'Liguoro, U. (2010). Sessions and session types: an overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer.

Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E., and Yoshida, N. (2007). Bounded session types for object-oriented languages. In *FMCO*, volume 4709 of *LNCS*, pages 207–245. Springer.

Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., and Drossopolou, S. (2006). Session types for object-oriented languages. In *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer.

Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., and Drossopolou, S. (2005). A distributed object-oriented language with session types. In *TGC*, volume 3705 of *LNCS*, pages 299–318. Springer.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. (2006). Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Operating Systems Review*, 40(4):177–190.

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

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

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

Giunti, M. (2011). A type checking algorithm for qualified session types. In *WWV*, volume 61 of *EPTCS*, pages 96–114.

Giunti, M., Honda, K., Vasconcelos, V. T., and Yoshida, N. (2009). Session-based type discipline for pi calculus with matching. In *PLACES*.

Giunti, M. and Vasconcelos, V. T. (2010). A linear account of session types in the pi calculus. In *CONCUR*, volume 6269 of *LNCS*, pages 432–446. Springer.

Honda, K. (1993). Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM Press.

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

Kobayashi, N., Pierce, B. C., and Turner, D. N. (1999). Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947.

Milner, R. (1992). Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141.

Milner, R. (1993). The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77.

Padovani, L. (2012). On projecting processes into session types. *Mathematical Structures in Computer Science*, 22(2):237–289.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pierce, B. C. and Sangiorgi, D. (1996). Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453.

Takeuchi, K., Honda, K., and Kubo, M. (1994). An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer.

Vasconcelos, V. T. (2011). Sessions, from types to programming languages. *Bulletin of the EATCS*, 103:53–73.

Vasconcelos, V. T. (2012). Fundamentals of session types. *Information and Computation*, 217:52–70.

Vasconcelos, V. T., Gay, S. J., and Ravara, A. (2006). Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87.

Yoshida, N. and Vasconcelos, V. T. (2007). Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *SecReT*, volume 171(4) of *ENTCS*, pages 73–93.