A linear account of session types in the pi calculus

Marco Giunti[1] and Vasco T. Vasconcelos[2]

[1] Faculty of Planning, University IUAV of Venice
[2] LaSIGE, Faculty of Sciences, University of Lisbon

Abstract. We present a reconstruction of session types in a conventional pi calculus where types are qualified as linear or unrestricted. Linearly typed communication channels are guaranteed to occur in exactly one thread, possibly multiple times. We equip types with a constructor that denotes the two ends of a same communication channel. In order to assess the flexibility of the new type system, we provide three distinct encodings (from the linear lambda calculus, from the linear pi calculus, and from the pi calculus with polarized variables) into our system. For each language we present operational and typing correspondences, showing that our system effectively subsumes the linear pi calculus as well as foregoing works on session types.

1 Introduction

Session types allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol. Introduced for a dialect of the pi calculus [6, 13], the concept has been transferred to different realms, including functional and object-oriented programming and operating systems; refer to [3] for a recent overview.

By way of motivation, consider a service allowing to create online petitions. Petition creators receive from the petition service a channel on which they provide the title of the petition, the petition text and 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 as follows,

$$\mathit{petitionOnline}(p).\overline{p}\,\mathit{title}.\overline{p}\,\mathit{description}.\overline{p}\,\mathit{dueDate}.\overline{p}\,\mathit{signature}.(\overline{a_1}\,p \mid \ldots \mid \overline{a_n}\,p)$$

where $x(y)$ denotes reading value y on channel x, $\overline{x}\,v$ denotes sending value v on channel x, and the vertical bar denotes parallel composition. Each of the acquaintances (not shown in the example), after reading p on channel a_i, can sign the petition and further distribute the channel at will.

The protocol for channel p can be concisely described by a type T of the form below, composed of an initial linear part that becomes shared (or unrestricted) in the later part.

$$\mathsf{lin}\,!\mathit{String}.\mathsf{lin}\,!\mathit{String}.\mathsf{lin}\,!\mathit{Date}.S \quad \text{where} \quad S = \mathsf{un}\,!\mathit{String}.S$$

The final part is unrestricted because it is desiderable, but not absolutely necessary, that acquaintances (including the petition creator) sign the petition; conversely, the initial part is linear because petitions cannot be signed without first setting up the title, the description and the due date.

In the process above, each channel a_i forwards p at type S. Such a channel may be given the type $\mathsf{lin}!S.\mathsf{un}\ \mathsf{end}$, if we require that acquaintances eventually receive the petition channel; the continuation is $\mathsf{un}\ \mathsf{end}$ (the type of a channel on which no further interaction is possible) allowing channel a_i to be discarded thereafter. Concentrating on the type of *petitionOnline*, we see that petition creators need it a type $S_1 = \mathsf{un}?T.S_1$ so that they may create as many petitions (of type T) as required. It should be easy to see that the service itself sees the same channel at the dual type $S_2 = \mathsf{un}!T.S_2$. The whole system, composed by the service running in parallel with petition creators can be typed by reconciling the two *end point types* S_1 and S_2 in a single, unordered, *channel type* of the form (S_1, S_2).

The language of the pi calculus, when considered in conjunction with a type system with session types, is known to require a means to distinguish the two ends of a session channel (S_1 and S_2 above) in order to preserve type soundness [4, 5, 17]. Alternative solutions not requiring such a distinction rely on the restriction of channel passing to bound output. Such systems include the original formulation of delegation in session types [6] as well as more recent works [2, 11].

Two approaches for distinguishing the ends of a channel are available in the literature: polarized channel variables [5], and form of channel double binder [15]. In the pi calculus with polarities the two ends of a channel x are distinguished by labelling each of its ends with a different label: x^+ and x^- denote the two ends of channel x. Given that from a given channel name one may find its two ends, one can restrict (the two ends of) a channel x with the usual pi calculus restriction operator $(\nu x)P$. Typing contexts, however accept two different entries for the same channel, one labelled with $+$, the other with $-$, as in the typing sequent below.

$$\Gamma, x^+ \colon S_1, x^- \colon S_2 \vdash \overline{x^+}\, v.P_1 \mid x^-(w).P_2$$

A variant of the above work, [15], uses distinct variables to describe the two ends of a same channel. In this case one cannot obtain the second end of a channel from the other end. It is restriction that puts together the two channel ends, by binding them together, as in $(\nu yz)P$. The assumptions in typing contexts are for simple variables, as in the example below where y and z denote the two ends of a same channel.

$$\Gamma, y \colon S_1, z \colon S_2 \vdash \overline{y}\, v.P_1 \mid z(w).P_2$$

The first work can be criticized for using non-conventional typing contexts, where typing information of a same channel x is split among two different entries, x^+ and x^-. The second work uses standard contexts but relies on a new scope restriction operator that binds two variables together. The goal of this work is to equip types with a constructor able to denote the two ends of a same channel. We then have the best of both worlds where we use the standard pi calculus (Milner et al. [10]) with standard typing sequents.

Syntax

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

Rules for structural congruence

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

Rules for reduction

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

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

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

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

Fig. 1. Pi calculus: Syntax and operational semantics

We test the flexibility of our type system by embedding the pi calculus with polarities and session types [5] (hence the conventional pi calculus [10]). We do the same for the linear pi calculus [7], and for the linear (call by value) lambda calculus as in [16]. For each of these languages we prove an operational and a typing correspondence result. From the two first embeddings we learn that our type system is an extension of advanced type systems for pi calculi. The embedding of the linear lambda calculus crucially takes linearity into consideration generating code accordingly (replicated or non replicated) for shared and linear resources.

The outline of the paper is as follows. The next section recalls the pi calculus and introduces our type system. Then, the subsequent three sections present the embeddings of the three languages mentioned above: pi calculus with polarities, linear pi calculus and linear lambda calculus. The last section presents some related as well as future work.

2 Pi Calculus

This section introduces the pi-calculus, its syntax and semantics, as well as our type system. The syntax is in Figure 1. We rely on a set of variables, ranged over

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

Fig. 2. Pi calculus: Types and typing contexts

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 a parallel composition, conditional, scope restriction, replication and the terminated process.

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, substitution of a variable x by a value v in a process P, denoted $P[v/x]$. We follow Barendregt's 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 the same figure. 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 over a terminated process has no effect, and allow exchanging the order of restrictions.

The reduction reduction is the smallest relation on processes including the rules in Figure 1. The [R-Com] rule communicates value v from an output prefixed one $\overline{x}\,v.P$ to an input prefixed process $x(y).Q$; the result is the parallel composition of the continuation processes, where the bound variable y is replaced by value v in the input process. 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.

The syntax of types is described in Figure 2. Types include the boolean type, end point types and channel types. The novelty with respect linear and session-based systems for the pi calculus is 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, a recursive type or a type variable. Each qualifier in a type controls the number of times the channel can be used at that 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. Pre 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 subexpression of the form $\mu a_1 \ldots \mu a_n.a_1$.

Type equality is not syntatic. Instead, we define it as the equality of regular infinite trees obtained by the infinite unfolding of recursive types, *modulo pair commutation*. The formal definition, which we omit, is co-inductive. In this way we use types $(\mu a.\mathsf{lin}!\mathsf{bool}.\mathsf{lin}?\mathsf{bool}.a, \mathsf{un}\ \mathsf{end})$ and $(\mathsf{un}\ \mathsf{end}, \mathsf{lin}!\mathsf{bool}.\mu b.\mathsf{lin}?\mathsf{bool}.\mathsf{lin}!\mathsf{bool}.b)$ 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. Unfolding is bound to terminate due to contractiveness. In other words, we take an equi-recursive view of types [12].

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 $q!T.S_2$. Session type end is dual of itself. Rather than providing a co-inductive definition of duality, we start by defining a function from end-point channels into end-point channels as follows.

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

Then, to check that 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 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 type environments, are inductively defined in Figure 2. In a context $\Gamma, x\colon T$ we assume that x does not occur in Γ; we also assume the various variable bindings in Γ to be unordered. We define predicate un to be true of a) the empty context, as well as of b) context $\Gamma, x\colon \mathsf{bool}$, context $\Gamma, x\colon \mathsf{un}\,p$, and context $\Gamma, x\colon (\mathsf{un}\,p_1, \mathsf{un}\,p_2)$, whenever $\mathsf{un}(\Gamma)$.

Typing relies on the context splitting operation described in Figure 3. It should be easy to understand: unrestricted types are copied into both contexts, linear types are placed in one of the two resulting contexts. The first four rules are standard [16], the last three rules are new to this work; the philosophy however remains the same. We omit three rules, duals to the last three, obtained by interchanging the end point types in the channel type (e.g., $(\mathsf{un}\,p_2, \mathsf{lin}\,p_1)$ in the last rule), for the effect can obtained by a suitable choice of the type in its equivalence class (recall that pair types are unordered).

Context splitting rules

$$\emptyset = \emptyset \cdot \emptyset \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad T = \mathsf{un}\, p \text{ or } (\mathsf{un}\, p_1, \mathsf{un}\, p_2)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \cdot (\Gamma_2, x\colon T)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad T = \mathsf{lin}\, p \text{ or } (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad T = \mathsf{lin}\, p \text{ or } (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)}{\Gamma, x\colon T = \Gamma_1 \cdot (\Gamma_2, x\colon T)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x\colon (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2) = (\Gamma_1, x\colon \mathsf{lin}\, p_1) \cdot (\Gamma_2, x\colon \mathsf{lin}\, p_2)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2) = (\Gamma_1, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2)) \cdot (\Gamma_2, x\colon \mathsf{un}\, p_2)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2) = (\Gamma_1, x\colon \mathsf{un}\, p_2) \cdot (\Gamma_2, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2))}$$

Typing rules for values

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash b\colon \mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad \frac{\Gamma \vdash v\colon (S, \mathsf{un}\, p)}{\Gamma \vdash v\colon S}$$

[T-Bool] [T-Var] [T-Strength]

Typing rules for processes

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \cdot \Gamma_2 \vdash P_1 \mid P_2} \qquad \frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash *P}$$

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

$$\frac{\Gamma_1 \vdash v\colon \mathsf{bool} \qquad \Gamma_2 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2} \qquad \frac{\Gamma, x\colon (S, \overline{S}) \vdash P}{\Gamma \vdash (\nu x)P}$$

[T-If] [T-Res]

$$\frac{\Gamma, x\colon S, y\colon T \vdash P \qquad (*)}{\Gamma, x\colon q?T.S \vdash x(y).P} \qquad \frac{\Gamma_1 \vdash v\colon T \qquad \Gamma_2, x\colon S \vdash P \qquad (**)}{\Gamma_1 \cdot (\Gamma_2, x\colon q\,!T.S) \vdash \overline{x}\, v.P}$$

[T-In],[T-Out]

$$\frac{\Gamma, x\colon (S, S'), y\colon T \vdash P \qquad (*)}{\Gamma, x\colon (q?T.S, S') \vdash x(y).P} \qquad \frac{\Gamma_1 \vdash v\colon T \qquad \Gamma_2, x\colon (S, S') \vdash P \qquad (**)}{\Gamma_1 \cdot (\Gamma_2, x\colon (q\,!T.S, S')) \vdash \overline{x}\, v.P}$$

[T-InC],[T-OutC]

$$(*)\ q = \mathsf{un} \Rightarrow q?T.S = S \qquad (**)\ q = \mathsf{un} \Rightarrow q!T.S = S$$

Fig. 3. Pi calculus: Typing

Equipped with the notions of type duality, unrestricted contexts, and context splitting we are ready to introduce the typing rules in Figure 3. The first two typing rules for values are standard. Rule [T-Strength] is central to our system with channels described as pairs of types; we discuss it after introducing the remaining typing rules.

For processes, rule [T-Inact] says that the terminated process can only be typed in an unrestricted context, ensuring that linear channels are given a chance to be consumed. Rule [T-Par] uses context splitting to partition linearly typed variables 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 unrestricted number of types. 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. Rule [T-Res] allows restricting channels whose end points are dual, making sure that communication on the channel happens according to the plan. Allowing to restrict a end point type S type would not break type preservation, Theorem 1, but we believe that such an alternative rule does not fit well in a linear system, where we expect linear channels to be given an opportunity to be consumed. Even though unrestricted end point types cannot be directly restricted, we can show that, for each derivation of $\Gamma, x\colon S \vdash P$, there is a derivation of $\Gamma, x\colon (S, \mathsf{un}\, p) \vdash P$, thus allowing to apply scope restriction to an otherwise unrestricted channel end.

We have two rules for input, [T-In] and [T-InC], depending on the type for channel x in the context. Rule [T-In] deals with end point types. If x is typed with $q\,?T.S$, we know that the bound variable y is of type T, and we type check P under the extra assumption $y\colon T$. Equally important is the fact that the continuation uses channel x at continuation type S, that is, process $x(y).P$ uses channel x at type $q\,?T.S$ whereas P may use the *same* channel this time at type S. Finally, unrestricted channels, given that they may be shared, must retain their behavior throughout computation, hence the side condition. A solution to the equation in the side condition is $\mu a?T.a$ for a not in T, which we abbreviate to $*?T$ (and similarly for output). Rule [T-InC] follows the same pattern, consuming one end point and keeping the other unchanged. Similarly to input, we have two rules for output. Rule [T-Out], splits the context in two parts, one to check v and the other to check continuation P. Notice that the context in the conclusion, $\Gamma_1 \cdot (\Gamma_2, x\colon q\,!T.S)$ allows to type process $\overline{x}\,x$ with a context $x\colon S$ with type S such that $S = \mathsf{un}!S.S$.

Rule [T-Strength] allows for a fine grained control of the channel ends of a given channel. A process holding the two ends of a given channel x, say $(*!\mathsf{bool}, *?\mathsf{bool})$, may pass the output capability only by using [T-Var] followed by [T-Strength] to obtain $\Gamma, x\colon (*!\mathsf{bool}, *?\mathsf{bool}) \vdash x\colon *!\mathsf{bool}$ and then compose with rule [T-Out] or [T-OutC] in a process of the form $\overline{y}\,x.P$. The rule is also fundamental in establishing the main result of this section.

To lighten the syntax in examples, we omit all unrestricted qualifiers and only annotate linear types. We also omit the trailing $\mathsf{un}\,\mathsf{end}$ in types, as well as the trailing $\mathbf{0}$ in processes. As an example, consider the type $?(\mathsf{lin}!\mathsf{bool}).S$ of an unrestricted channel that receives a linear channel capable of outputting a boolean value. The following sequent is easy to establish,

$$x\colon ?(\mathsf{lin}!\mathsf{bool}).S \vdash x(z).\overline{z}\,\mathsf{true} \mid x(w).\overline{w}\,\mathsf{false}$$

but only for an appropriate type S. Reading rule [T-In], we realize that S must be equivalent to $?(\mathsf{lin}!\mathsf{bool}).S$, that is S must be (equivalent to) $\mu a.?(\mathsf{lin}!\mathsf{bool}).a$, abbreviated to $*?(\mathsf{lin}!\mathsf{bool})$. Continuing with the example, if P is the above process, then $P \mid (\nu y)\overline{x}\,y$ is not typable, for the linear input capability of channel y is never exercised. But $P \mid (\nu y)(\overline{x}\,y \mid y(u))$ is typable under context $x\colon (*?(\mathsf{lin}!\mathsf{bool}), *!(\mathsf{lin}?\mathsf{bool}))$.

Given that a type $(\mathsf{un}\, p_1, \mathsf{lin}\, p_2)$ cannot possibly be restricted in a process (cf. rule [T-Res]), the reader may wonder why we consider them at all. It turns out that free channel output may lead to situations where a thread holds the two ends of a same channel [17]. For instance, process $\overline{z}\, x.\ |\ z(w).w(y).\overline{x}\,\mathsf{true}$, 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-InC] to P we obtain a judgment with a un-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-OutC] gets rid of the un-lin type, yielding $z\colon (*?S, *!S), x\colon (\mathsf{end}, \mathsf{end}) \vdash \mathbf{0}$ typable under rule [T-Inact].

We conclude the section with the main result of our system. Reduction preserves typability only for a certain kind of contexts. To understand why reduction does not preserve typability in the presence of arbitrary contexts, take for P the process $x(z).\mathsf{if}\, z\, \mathsf{then}\, \mathbf{0}\, \mathsf{else}\, \mathbf{0}\ |\ (\nu y)\overline{x}\, y$. We can easily see that P is typable under the (non balanced) context $x\colon (\mathsf{lin!end.end}, \mathsf{lin?bool.end})$. But P reduces to process $(\nu y)\mathsf{if}\, y\, \mathsf{then}\, \mathbf{0}\, \mathsf{else}\, \mathbf{0}$ which is not typable. The whole purpose of balancing is to make sure that the type of y in the output is that of z in the input.

We define predicate *balanced* to be true of a) the empty context, and b) context $\Gamma, x\colon \mathsf{bool}$ and context $\Gamma, x\colon (S, \overline{S})$ whenever Γ is balanced.

Theorem 1 (Type Preservation). *If $\Gamma_1 \vdash P_1$ with Γ_1 balanced and $P_1 \to P_2$, then $\Gamma_2 \vdash P_2$ with Γ_2 balanced.*

Proof (Sketch). Albeit standard, the proof is quite long due to the combinatorics introduced by (the various rules in) context splitting, the lin-un qualifiers and the four rules for input and for output. We rely on several standard auxiliary results, including unrestricted context weakening, context strengthening, a substitution lemma (stating that if $\Gamma_1 \vdash v\colon T$ and $\Gamma_2, x\colon T \vdash P$ and $\Gamma_1 \cdot \Gamma_2$ is defined then $\Gamma_1 \cdot \Gamma_2 \vdash P[v/x]$), and balanced context preservation for structural congruence. In order to proceed by induction on the derivation of the reduction step when [T-Par] is the last rule, we need a stronger statement that details the relation between Γ_1 and Γ_2, namely $\Gamma_2 = \Gamma_1$ or $\Gamma_1 = \Gamma, x\colon (q?T.S, q!T.\overline{S})$ and $\Gamma_2 = \Gamma, x\colon (S, \overline{S})$.

3 Embedding the Pi Calculus with Polarities

This section shows that our type system embeds the polarity system introduced by Gay and Hole [5]. Since Gay and Hole show that the pi calculus with polarities embeds the simply typed pi calculus; by transitivity our language embeds the simply typed pi calculus as well.

In Figure 4 we present the branch-select free fragment of the pi calculus with polarities. Variables may be *polarized*, occurring in processes as well as in typing contexts as x^+ or x^- or simply as x. We write x^p for a general polarized name, where p represents an optional polarity. Duality on polarities, written $\overline{p}$ exchanges $+$ and $-$. The new constructors of the language, input and output,

New syntactic forms

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
	$\hat{}T$	standard channel		$!T.S$	send
	S	session channel		a	type variable
	a	type variable		$\mu a.S$	recursive type

New reduction rules

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

Context updating

$$\begin{aligned} \Gamma + x^p\colon S &= \Gamma, x^p\colon S && \text{if } x^p, x \notin \mathrm{dom}(\Gamma) \text{ and } S = ?T.S, !T.S, \mathsf{end} \\ \Gamma + x\colon T &= \Gamma, x\colon T && \text{if } x, x^+, x^- \notin \mathrm{dom}(\Gamma) \\ \Gamma, x\colon T + x\colon T &= \Gamma, x\colon T && \text{if } T = \hat{}T, \mathsf{bool} \end{aligned}$$

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]

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

[T-Out] [T-OutS]

Fig. 4. Pi calculus with polarities

are in Figure 4; the remaining are taken from Figure 1; 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. From the above description it should be obvious 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 polarized processes all occurrences of $+$ and $-$, to yield a process generated by the grammar in Figure 1. There is an obvious operational correspondence between the two languages, stated in Theorem 2. The converse is clearly not true. Take for P the polarized process $\overline{x^+} \mid x^+()$. Then $\mathrm{erase}(P) = \overline{x} \mid x()$ reduces while P does not.

The language of types includes a distinct category S for (linear) session types. Since we restrict our language to the branch-select free fragment of [5], we ignore subtyping. Duality is defined as in Section 2, with the appropriate changes which amount erasing the qualifiers. Typing contexts now gather assumptions on polarized 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. Context updating is different from splitting (in Figure 3) on what concerns unrestricted types: $\emptyset + (x\colon \hat{}T)$ is defined, whereas $\emptyset \cdot (x\colon \mathsf{un}\, p)$ is not. We say that a context is *unlimited* if it contains no session types, and is *completed* if every session in it is end.

The typing relation is inductively defined by the rules in Figure 4. Rule [T-NewS] requires the types for the two channel end points to be of dual types; contrast with rule [T-Res] in Figure 3: our system merges the two end points in a single variable and requires the two components of the channel type to be of dual types. Rules [T-In] and [T-InS] in Figure 4 have their counterpart in rules [T-In] and [T-InC] in Figure 3. The choice here is not based on whether the type for the input channel is an end point or a channel type but rather on whether the qualifier is linear or unrestricted. The same can be said of rules [T-Out] and [T-OutS].

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 types and contexts for the polarized language (as in Figure 4) to those in our language (Figure 1). The definition is as follows; recall from Section 2 that we use $*?T$ as an abbreviation for $\mu a.?T.a$, for some a not in T. To translate typing contexts we assume that if both x^+ and x^- are in Γ then they occur in contiguous positions (and in this order). The translation of typing contexts is as follows, where the rules must be tried in the given order; the first rule for mapping non-empty contexts is for polarized pairs while the second rule is for single entries.

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

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

Theorem 2 (Polarity-Pi To Pi Correspondence).

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

New syntactic forms

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

Combination of types

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

Combination of contexts

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) + \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \text{ and } x \notin \mathrm{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_2) \text{ and } x \notin \mathrm{dom}(\Gamma_1) \end{cases}$$

Typing rules for processes

$$\frac{\Gamma \vdash_{\mathsf{l}} P_1 \qquad \Gamma \vdash_{\mathsf{l}} P_2}{\Gamma + v : \mathsf{bool} \vdash_{\mathsf{l}} \mathsf{if}\,v\,\mathsf{then}\,P_1\,\mathsf{else}\,P_2} \qquad \frac{\Gamma, x\colon q\,\mathsf{io}\;T \vdash_{\mathsf{l}} P}{\Gamma \vdash_{\mathsf{l}} (\nu x)P} \qquad \text{[T-If] [T-Res]}$$

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

Fig. 5. Linear pi calculus

4 Embedding the Linear Pi Calculus

In this section we analyse (a synchronous variant of) the linear pi calculus [7] 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 5 defines the syntax of types and the typing rules for processes. Types have now the form $q\,c\,T$ where c is a capability formally defined as one of the following sets.

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

The linear discipline is imposed by way of a + combination operation over types, defined in Figure 5. The operator is extended point-wise to typing contexts. Notice that context combination is different from the context splitting operation defined in Figure 3 when in the presence of unrestricted types: context splitting does not allow composing $(\Gamma_1, x\colon \mathsf{un}\,c\,T)$ with Γ_2 whenever $x \notin \mathrm{dom}(\Gamma_2)$ or when $\Gamma_2(x) \neq \mathsf{un}\,c\,T$.

The typing system for the linear pi-calculus is defined by the rules in Figure 3 together with rules [T-Inact], [T-Repl] and [T-Par] in Figure 4. Rule [T-Out] is an adaptation of that in [7] 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 also adapt 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 to channel x type $\mathsf{lin}\,\mathsf{o}\,\mathsf{bool}$; cf. discussion around rule [T-Res] in Section 2.

The compositional encoding of linear types is defined below and is useful to understand the reconstruction of session types introduced in Section 2. A linear input (output) type is embedded as a linear input (output) type whose continuation is $\mathsf{un}\,\mathsf{end}$, meaning that the continuation process cannot further use the channel. Unrestricted input (output) types are mapped into unrestricted recursive input (output) types. For instance, the type $\mathsf{lin}\,\mathsf{i}(\mathsf{lin}\,\mathsf{io}\,(\mathsf{un}\,\mathsf{io}\,\mathsf{bool}))$ is mapped into the type $\mathsf{lin}\,?(\mathsf{lin}\,!T.\mathsf{un}\,\mathsf{end}, \mathsf{lin}\,?T.\mathsf{un}\,\mathsf{end}).\mathsf{un}\,\mathsf{end}$ where $T = (*!\mathsf{bool}, *?\mathsf{bool})$.

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

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

Theorem 3 (Linear-Pi To Pi Correspondence). *If $\Gamma \vdash_{\mathsf{l}} P$ then $[\![\Gamma]\!] \vdash P$.*

5 Embedding the Linear Lambda Calculus

This section shows that the call-by-value linear lambda calculus can be faithfully encoded in our language. We follow the presentation of Walker [16], except that we use an implicitly typed language.

The syntax of the language is in Figure 6; we rely on the set of variables introduced in Section 2 for the pi calculus; the missing non-terminal symbols, q, b and so on, are in Figure 1. Values are qualified, linear or unrestricted, and include boolean values and abstractions. Terms are variables, values, and applications and are evaluated in an abstract machine with an explicit store. The store is a sequence of variable-value pairs, treated as a map from variables into values. To simplify the presentation of the evaluation relation, we use an auxiliary function, $S \overset{q}{\sim} x$ that deallocates the value associated with variable x in S when the qualifier q is lin, and leaves S unchanged otherwise. The evaluation reduction copies values into the store, associating them with a fresh variable (rule [E-val]). For function application the value associated with the function is looked upon in the store; if linear it is then deallocated (rule [E-app]). The remaining two rules implement the call-by-value strategy. We denote by $\rightarrow_\lambda$ the reduction relation in Figure 6.

For typing, rule [T-Var] is that of the pi-calculus (Figure 3). Rule [T-Bool] contrasts with its homonymous in Figure 3 in that values in the linear lambda calculus are qualified, the type of a value inheriting the qualifier of the value. The remaining two rules, for abstraction and application, are standard in the linear lambda calculus; notice the $q(\Gamma)$ in rule [T-abs] requiring an unrestricted

Syntax

v	$::=$	Values:	p	$::=$	Pretypes:
	$q\ b$	boolean		bool	boolean
	$q\ \lambda x.M$	abstraction		$T \to T$	function
M	$::=$	Terms:	T	$::=$	Types:
	x	variable		$q\ p$	qualified pretype
	v	value	S	$::=$	Stores:
	MM	application		$\emptyset$	empty store
				$S, x \mapsto v$	store binding

Store deallocation

$$(S_1, x \mapsto v, S_2) \overset{\mathsf{lin}}{\sim} x = S1, S2 \qquad\qquad S \overset{\mathsf{un}}{\sim} x = S$$

Evaluation

$$(S; v) \to_\lambda (S, x \mapsto v; x) \qquad \frac{S(x_1) = q\lambda y.M}{(S; x_1 x_2) \to_\lambda (S \overset{q}{\sim} x_1; M[x_2/y])} \quad \text{[E-VAL] [E-APP]}$$

$$\frac{(S; M_1) \to_\lambda (S'; M_1')}{(S; M_1 M_2) \to_\lambda (S'; M_1' M_2)} \qquad \frac{(S; M) \to_\lambda (S'; M')}{(S; xM) \to_\lambda (S'; xM')} \quad \text{[E-FUN] [E-ARG]}$$

Typing

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash_\lambda x\colon T} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_\lambda q\, b\colon q\, \mathsf{bool}} \quad \text{[T-VAR] [T-BOOL]}$$

$$\frac{\Gamma, x\colon T_1 \vdash_\lambda M\colon T_2 \qquad q(\Gamma)}{\Gamma \vdash_\lambda q\lambda x.M\colon q\, T_1 \to T_2} \qquad \frac{\Gamma_1 \vdash_\lambda M_1\colon q\, T_1 \to T_2 \qquad \Gamma_2 \vdash_\lambda M_2\colon T_1}{\Gamma_1 \cdot \Gamma_2 \vdash_\lambda M_1 M_2\colon T_2} \quad \text{[T-ABS] [T-APP]}$$

Fig. 6. Linear lambda calculus

function to contain only unrestricted free variables ($\mathsf{un}(\Gamma)$ is defined in Section 2; $\mathsf{lin}(\Gamma)$ is true).

For the translation we rely on a polyadic variant of the pi language, allowing channels to carry an arbitrary (but fixed) number of values. The extension is straightforward to incorporate: for processes we need polyadic output and input, $x(\vec{x}).P$ and $\overline{x}\,\vec{v}.P$; for types (pre types, rather) we need $?\langle\vec{T}\rangle.S$ and $!\langle\vec{T}\rangle.S$. The extension can be incorporated in the base theory or added as an encoding [15].

On what concerns the translation of types below, the interesting cases are the two forms of arrow types. An unrestricted $T_1 \to T_2$ type is translated as a pair of types: the $*?X$ part caters for the resource (the function proper) and the $*!X$ for its clients. Channels describing functions carry a pair X of values: the first element in the pair is the argument to the function ($[\![T_1]\!]$ in X); the second is a channel that will convey the result and that will be used exactly once (a linear channel of type $\mathsf{lin}![\![T_2]\!].\mathsf{un}\ \mathsf{end}$). The type for linear resources is similar, only that

they are linear, rather than unrestricted. The translation of terms follows that of Milner [9] with two exceptions. On the one hand, the value qualifiers are taken into consideration in the translation: a linear value is translated into a simple output (in the case of a boolean value) or a simple input (in the case of an abstraction); only for unrestricted values replication is used. On the other hand, applications of the form xM are partially evaluated, allowing for a simple operational correspondence (cf. [14]).

$$
\begin{array}{rl}
[\![q\ \mathsf{bool}]\!] = & q\ \mathsf{bool} \\
[\![\mathsf{un}\ T_1 \to T_2]\!] = & (*?X, *!X) \\
[\![\mathsf{lin}\ T_1 \to T_2]\!] = & (\mathsf{lin}?X.\mathsf{un\ end}, \mathsf{lin}!X.\mathsf{un\ end}) \\
\text{where } X = & \langle [\![T_1]\!], \mathsf{lin}![\![T_2]\!].\mathsf{un\ end}\rangle \\
[\![\emptyset]\!] = & \mathbf{0} \\
[\![S, x \mapsto v]\!] = & [\![S]\!] \mid [\![x \mapsto v]\!] \\
[\![x \mapsto q\ b]\!] = & [\![q]\!]\overline{x}\, b \\
[\![x \mapsto q\lambda y M]\!] = & [\![q]\!]x(yp).[\![M]\!]_p
\end{array}
\qquad
\begin{array}{rl}
[\![x]\!]_p = & \overline{p}\, x \\
[\![v]\!]_p = & (\nu x)([\![x \mapsto v]\!] \mid \overline{p}\, x) \\
[\![xM]\!]_p = & (\nu r)([\![M]\!]_r \mid r(y).\overline{x}\, yp) \\
[\![MN]\!]_p = & (\nu s)([\![M]\!]_s \mid s(x).(\nu r)(\\
& [\![N]\!]_r \mid r(y).\overline{x}\, yp)) \\
[\![S; M]\!] = & (\nu\, \mathrm{dom}(S))([\![S]\!] \mid [\![M]\!]) \\
[\![\mathsf{un}]\!] = & * \\
[\![\mathsf{lin}]\!] = & \text{the empty string}
\end{array}
$$

We are now in a position to state the main result of this section. Let $\to^*$ be the reflexive and transitive closure of the reduction relation $\to$ defined in Figure 1.

Theorem 4 (Linear-Lambda to Pi Correspondence).

1. *If* $\Gamma \vdash_\lambda M : T$, *then* $[\![\Gamma]\!], p : \mathsf{lin}![\![T]\!].\mathsf{un\ end} \vdash [\![M]\!]_p$.
2. *If* $(S; M) \to_\lambda (S'; M')$, *then* $[\![S; M]\!] \to^* [\![S'; M']\!]$.

6 Conclusion

As mentioned in the introduction, the pi calculus equipped with a polarity-based typing system [5] and the (double binder) pi calculus equipped with a conventional typing system [15] are the works closest to ours. Here we try to obtain the same results, relying on the traditional pi calculus equipped with a conventional type system. Towards this end we introduce an unordered pair constructor denoting, at type level, the two ends of a same channel. In order to distinguish linear from unrestricted variables we use type qualifiers applied to pre types, inspired by Walker's presentation of substructural type systems [16]. Caires and Pfenning take a different approach, closely adhering to linear logic, treating all variables as linear and using exponential "!" to describe shared resources [1].

Algorithmic type checking is left for further work; the language with double binders [15] is equipped with such a system. Gay and Hole address the problem of polarity inference for closed processes under certain restrictions [5]. Particularly promising is the F$^\circ$ ("F-pop") system by Maruzak et al. [8], where a kinding system, instead of type qualifiers, simplifies the use of linearity in functional programming languages, including a novel form of subtyping between linear and unrestricted kinds, which we would like to explore.

Acknowledgments. This work was supported by FCT/MCTES via projects PTDC/EIA–CCO/105359/2008 and CMU–PT/NGN44–2009–12. The authors would like thank Simon Gay and Nils Gesbert for insightful comments and the anonymous referees for constructive criticisms and detailed comments.

References

1. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In this volume.
2. Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of session types. In *PPDP*, pages 219–230. ACM, 2009.
3. Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: an overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer-Verlag, 2010.
4. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*, 207(5):595 – 641, 2009.
5. Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
6. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
7. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
8. Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in system F^o. In *TLDI*, pages 77–88. ACM, 2010.
9. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
10. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.
11. Luca Padovani. Session types at the mirror. *EPTCS*, 12:71–86, 2009.
12. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
13. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
14. Vasco T. Vasconcelos. Lambda and pi calculi, CAM and SECD machines. *Journal of Functional Programming*, 15(1):101–127, 2005.
15. Vasco T. Vasconcelos. *SFM 2009*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag, 2009.
16. David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.
17. Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *SecReT'07*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier Science Publishers, 2007.