Session-based Type Discipline for Pi Calculus with Matching

Marco Giunti[*] Kohei Honda[†] Vasco T. Vasconcelos[*] Nobuko Yoshida[‡]

Introduction. In [7] we have introduced an extension of the first session typing system [10] that allows higher-order session communication. In the new system, the reduction rule for session passing

$$k![k'].P \mid k?(k').Q \to P \mid Q$$

does not allow the transmission of an arbitrary channel. In most situations a receiving process $k?(k'').Q$ can be alpha-converted ahead of communication so that the bound channel k'' syntactically matches the free channel k' in the object of the sending process [11]. The exception happens exactly when k' is free in Q: alpha-conversion becomes impossible (for it would capture the free variable k'), and communication cannot occur.

A more liberal rule allows the transmission of an arbitrary channel, implying a substitution on the client side.

$$k![k'].P \mid k?(x).Q \to P \mid Q[k'/x]$$

Unfortunately this rule breaks subject reduction, a technique that is commonly used to prove that typable processes do not reduce to errors. A counterexample is a process which, possessing one end of a channel, receives the second end. The process:

$$k![k'] \mid k?(k'').k''?(y).k'![1]$$

is typable in the system of [7] under typing $k\colon \perp, k'\colon \perp$, but reduces to process $k'?(x).k'![1]$ which is not typable under the *same typing* [4]. To tackle this problem and recover subject reduction, starting from Gay and Hole [6], many works resort to decorate channel-ends with polarities. In this paper we show that the polarity-free language of reference [7] extended with the more liberal session-passing rule above is type-safe, even though it does not enjoy subject reduction.

A second contribution of the paper is a simple generalization of the theory of session types that allows processes to both send a channel and still use it, under limited conditions. The idea is that processes can safely use channels both for communication and for non-communication operations, such as testing the identity of a session, or storing the channel in some data structure. The example below illustrates the idea.

Consider a system with long running sessions composed of different degrees of trust. The protocol relies on a database of malicious sessions that offers operations to add and to check for untrusted channels; the database runs in parallel with the web service. Untrusted channels are stored in a set implemented as a pi calculus process and are typed with end. By providing for channel matching, we can implement a process $\mathsf{if}\ y \in \mathsf{set}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ that continues as P if channel y is stored in the set of untrusted channels, and continues as Q otherwise. The insertion of a new entry y in the set is done by the process $\mathsf{Store}[y].P$. The code for the database process is below; we annotate variables with the type at which they are used.

$$\mathsf{DB}(d) = \mathsf{accept}\ d(x).x \rhd \{\mathsf{contains} : x?(y^{\mathsf{end}}).\mathsf{if}\ y \in \mathsf{set}\ \mathsf{then}\ x \lhd \mathsf{no}.\mathsf{DB}(d) \mathsf{else}\ x \lhd \mathsf{yes}.\mathsf{DB}(d),$$
$$\mathsf{put} : x?(y^{\mathsf{end}}).\mathsf{Store}[y].\mathsf{DB}(d)\}$$

Instances of the service that need an high level of confidentiality, whenever they receive a channel at some type T, send the channel to the database at type end and wait for an ack ensuring that the session is trusted before using it. The fragment of the code of a client querying the database for the trust of a channel is below. Notice that after passing the session at type end the continuation uses it at type T.

$$x?(y).\mathsf{request}\ d(z).z \lhd \mathsf{contains}.z![y^{\mathsf{end}}].z \rhd \{\mathsf{yes} : P(y^T) \ [\!]\ \mathsf{no} : Q(y^T)\}$$

The service itself is in charge of signaling untrusted channels to the database. Such channels could still be used by instances of the service that do not require confidentiality, for instance when sessions are meant for exchanging public data. The code below describes an instance sending an untrusted channel y at type T to the context and signaling the channel at type end to the database.

$$x![y^T].\mathsf{request}\ d(z).z \lhd \mathsf{put}.z![y^{\mathsf{end}}].$$

[*]Department of Informatics, University of Lisbon
[†]Department of Computer Science, Queen Mary University of London
[‡]Department of Computing, Imperial College of London

$$
\begin{array}{rll}
P & ::= u![e].P \mid u?(x).P \mid P \mid P' \mid (\nu a)P \mid \textsf{if } e \textsf{ then } P \textsf{ else } P' \mid \mathbf{0} & \\
 & \mid \textsf{def } D \textsf{ in } P \mid X[\tilde{e}] & \text{recursion} \\
 & \mid \textsf{request } u(x).P \mid \textsf{accept } u(x).P & \text{session request / acceptance} \\
 & \mid u \lhd l.P \mid u \rhd \{l_i \colon P_i\}_{i \in I} & \text{label selection / branching} \\
e & ::= v \mid [u = v] \mid e \textsf{ and } e' \mid e \textsf{ or } e' \mid \textsf{not } e & \text{expressions} \\
u,v & ::= a \mid x \mid \textsf{true} \mid \textsf{false} & \text{values} \\
D & ::= \{X_i(\tilde{x}_i) = P_i\}_{i \in I} & \text{declaration for recursion}
\end{array}
$$

Figure 1: The top level syntax of processes.

Pi calculus with sessions. We distinguish two languages: the *top-level language* and the *runtime* language. The former constitutes the language programmers program with; the latter includes constructs useful to describe the operational semantics, but that need not be accessible to programmers.

The top level language relies on a few base sets: *names*, ranged over by a, *variables* ranged over by x,y, *labels* ranged over by l, and *process variables* ranged over by X. The syntax is in Figure 1 and includes matching processes of the form if $[u = v]$ then P else P'.

The *runtime* language is the object of the operational semantics. It requires one more base set : *channels* ranged over by k. We indicate runtime processes with Q.

$$
\begin{array}{rll}
Q & ::= \langle \text{Fig. 1} \rangle \mid (\nu k)Q & \text{channel binder} \\
u,v & ::= \langle \text{Fig. 1} \rangle \mid k & \text{linear channel} \\
n & ::= a \mid k & \text{identifiers}
\end{array}
$$

The runtime language differs only in that (synchronization) identifiers include channels, so that, for example $k![\textsf{true}].\mathbf{0}$ is a process (as opposed to $x![\textsf{true}].\mathbf{0}$ in the base language). The bindings for the runtime language are processes $(\nu n)P$, which binds occurrences of the identifier n in P, i.e. $(\nu a)P$ or $(\nu k)P$, and def $\{X_i(\tilde{x}_i) = P_i\}_{i \in I}$ in P, which binds occurrences of each process variable X_i in process P_i. The definition of *bound* and *free* names and channels is standard, and so is the capture-free *substitution* of a variable x with value v in a process P, denoted by $P[v/x]$. We implicitly assume that in all mathematical contexts all bound identifiers are pairwise disjoint and disjoint from the free identifiers. The operational semantics relies on structural congruence, for the syntactic re-arrangement of processes, preparing these for the application of the rules in the reduction relation. *Structural congruence* is the smallest relation including the rules below:

$$
\begin{array}{c}
Q \mid \mathbf{0} \equiv Q \qquad Q \mid Q' \equiv Q' \mid Q \qquad (Q \mid Q') \mid Q'' \equiv Q \mid (Q' \mid Q'') \\
(\nu n)Q \mid Q' \equiv (\nu n)(Q \mid Q') \qquad (\nu n')(\nu n)Q \equiv (\nu n)(\nu n')Q \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \\
\textsf{def } D \textsf{ in } \mathbf{0} \equiv \mathbf{0} \quad \textsf{def } DD' \textsf{ in } Q \equiv \textsf{def } D'D \textsf{ in } Q \quad \textsf{def } D \textsf{ in } (\nu n)Q \equiv (\nu n)\textsf{def } D \textsf{ in } Q \\
(\textsf{def } D \textsf{ in } Q) \mid Q' \equiv \textsf{def } D \textsf{ in } (Q \mid Q') \qquad \textsf{def } D \textsf{ in } (\textsf{def } D' \textsf{ in } Q) \equiv \textsf{ def } D\, D' \textsf{ in } Q
\end{array}
$$

Reduction also relies on a standard *evaluation* function $\downarrow$, reducing expressions to values. We assume $[u = u] \downarrow$ true, and $[u = v] \downarrow$ false whenever $u \neq v$.

The reduction rules are in Figure 2. Sessions between two partners start when an accept process meets a request process, as described in rule [LINK]. The result of such interaction is the creation of a new channel k, that replaces both the bound variable x in the continuation Q of the accept process and the bound variable y in continuation Q' of request. In rule [COM], when a send process $k![e].Q$ meets a receive process $k?(x).Q$, the semantics start by evaluating expression e to a value v, which is passed from the send party to the receive party, replacing variable x in Q. In case the expression evaluates to a channel k, we have session passing.

Reduction may go wrong for a number of reasons. Here we are interested on *communication* errors, problems arising from a mismatch of the expectations of the partners involved in a particular interaction. Examples include the parallel composition of two partners both trying to output, as in $k![\textsf{true}]. \mid k \lhd l.Q$, or even when three partners try to read/write on the same channel, as for example in $k![\textsf{true}]. \mid k \lhd l.Q \mid k?(x).Q'$. The formal definition of what we mean by an error process is below [7].

$$\text{accept } a(x).Q \mid \text{request } a(y).Q' \;\rightarrow\; (\nu k)(Q[k/x] \mid Q'[k/y]) \qquad [\text{LINK}]$$
$$e \downarrow v \;\Rightarrow\; k![e].Q \mid\; k?(x).Q' \;\rightarrow\; Q \mid Q'[v/x] \qquad [\text{COM}]$$
$$k \lhd l_j.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I} \;\rightarrow\; Q \mid\; Q_j \qquad (j \in I) \qquad [\text{LABEL}]$$
$$e \downarrow \text{true} \;\Rightarrow\; \text{if } e \text{ then } Q \text{ else } Q' \;\rightarrow\; Q \qquad [\text{IFT}]$$
$$e \downarrow \text{false} \;\Rightarrow\; \text{if } e \text{ then } Q \text{ else } Q' \;\rightarrow\; Q' \qquad [\text{IFF}]$$
$$\tilde{e} \downarrow \tilde{v} \Rightarrow \text{def } X(\tilde{x}) = Q \text{ in } (X[\tilde{e}] \mid Q') \;\rightarrow\; \text{def } X(\tilde{x}) = Q \text{ in } (Q[\tilde{v}/\tilde{x}] \mid Q') \qquad [\text{DEF}]$$
$$Q \rightarrow Q' \;\Rightarrow\; (\nu n)Q \rightarrow (\nu n)Q' \qquad [\text{SCOP}]$$
$$Q \rightarrow Q' \;\Rightarrow\; Q \mid Q'' \;\rightarrow\; Q'' \mid Q' \qquad [\text{PAR}]$$
$$Q \rightarrow Q' \;\Rightarrow\; \text{def } D \text{ in } Q \;\rightarrow\; \text{def } D \text{ in } Q' \qquad [\text{DEFIN}]$$
$$Q' \equiv Q_1 \text{ and } Q_1 \rightarrow Q_2 \text{ and } Q_2 \equiv Q'' \;\Rightarrow\; Q' \rightarrow Q'' \qquad [\text{STR}]$$

Figure 2: Reduction

Definition 1 (Error Process)**.** *A* k-process *is a process prefixed by channel k; that is:* $k![e].Q$, $k?(x).Q$, $k \lhd l.Q$, *or* $k \rhd \{l_i\colon Q_i\}_{i\in I}$. *A* k-redex *is the parallel composition of two k-processes, either of form* $(k![e].Q \mid k?(x).Q')$ *or* $(k \lhd l.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I})$. *Then Q is an* error *if* $Q \equiv (\nu\tilde{n})(\text{def } D \text{ in } (Q' \mid Q''))$ *where* Q' *is, for some k, the parallel composition of* either *two k-processes that do not form a k-redex,* or *three or more k-processes.*

In the following, we provide for a typing system for top level processes able to filter out all errors that can arise during the computation.

Type Assignment for the Top Level Language. We borrow from [7] the distinction between *sorts*, ranged over by S, and *types*, ranged over by T. Sorts are assigned to values: a boolean value has type bool, a name has a type $\langle T\rangle$ describing the interactive sessions it may engage upon. The interactive session, in turn, is described by a type T with the following grammar:

$$T \;::=\; ?[S].T \;\mid ?[T].T \;\mid\; \&\{l_i\colon T_i\}_{i\in I} \;\mid\; \text{end} \;\mid ![S].T \;\mid ![T].T \;\mid\; \oplus\{l_i\colon T_i\}_{i\in I} \;\mid\; t \;\mid\; \mu t.T$$

Types $![S].T$ and $?[S].T$ describe channels willing to send or to receive a value of sort S (that is a boolean value or a name) and then continue its interaction as prescribed by T. Types $![T].T$ and $?[T].T$ are similar, only that this time the value exchanged is a linear value (a channel) described by a type, rather than a shared value described by a sort. Differently from [7], in our framework the type end can also be used to type, e.g., conditional expressions. Types $\oplus\{l_i\colon T_i\}_{i\in I}$ and $\&\{l_i\colon T_i\}_{i\in I}$ represent channels ready to select (to send) a label or to branch on an incoming label. The two last type constructors allow for recursive type structures.

Duality is a central concept in the theory of session types. The function $\overline{T}$ yields the canonical dual of a session type T by exchanging ! with ?, and & with $\oplus$. The formal definition is below.

$$\overline{?[\alpha].T} = ![\alpha].\overline{T} \qquad \overline{\oplus\{l_i\colon T_i\}_{i\in I}} = \&\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{\text{end}} = \text{end}$$
$$\overline{![\alpha].T} = ?[\alpha].\overline{T} \qquad \overline{\&\{l_i\colon T_i\}_{i\in I}} = \oplus\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{\mu X.T} = \mu X.\overline{T} \qquad \overline{X} = X$$

The type system distinguishes the sorts assigned to names and boolean values, from the types assigned to channels. Types are treated linearly, a map from channels and variables into types T is denoted by the *type environment* Δ. Sorts are treated classically, a map from names and variables into sorts S is denoted by the *sort environment* Γ. Syntax $\Delta\cdot x\colon T$ denotes the disjoint map union of Δ and $(x\colon T)$. The notation is extended straightforwardly to sort and type environments.

We let the merge, noted $\otimes$, be the smallest commutative binary relation over types satisfying $T \otimes \text{end} = T$. We extend the operation to type environments and let $\Delta \otimes (k : T) = \Delta\cdot k : T$ whenever $k \notin \text{dom}(\Delta)$, otherwise if $\Delta(k) \otimes T$ is defined we let $\Delta \otimes k : T = \Delta'$ with Δ' differing from Δ only in $\Delta'(k) = \Delta(k) \otimes T$.

The typing rules for matching expressions are below. Notice that in rule for compare values taken from the type environment the types at which the values are known may be different. Particularly, to test the identity of a value it is sufficient that the value is known at type end.

$$\frac{\Gamma \vdash u\colon S, v\colon S}{\Gamma \vdash [u = v]\colon \text{bool}} \qquad \frac{\Delta \vdash u\colon T, v\colon T'}{\Delta \vdash [u = v]\colon \text{bool}}$$

$$\frac{\Gamma \vdash u\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta\cdot x\colon T}{\Gamma \vdash \mathsf{accept}\ u(x).P \triangleright \Delta} \qquad \frac{\Gamma \vdash u\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta\cdot x\colon \overline{T}}{\Gamma \vdash \mathsf{request}\ u(x).P \triangleright \Delta} \qquad \text{[ACC], [REQ]}$$

$$\frac{\Gamma \vdash e\colon S \qquad \Gamma \vdash P \triangleright \Delta\cdot u\colon T}{\Gamma \vdash u![e].P \triangleright \Delta\cdot u\colon\ ![S].T} \qquad \frac{\Gamma\cdot x\colon S \vdash P \triangleright \Delta\cdot u\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta\cdot u\colon\ ?[S].T} \qquad \text{[SEND], [RCV]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta\cdot u\colon U\cdot v\colon T_2 \quad T = T_1 \otimes T_2}{\Gamma \vdash u![v].P \triangleright \Delta\cdot u\colon\ ![T].U\cdot v\colon T_1} \qquad \frac{\Gamma \vdash P \triangleright \Delta\cdot u\colon U\cdot x\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta\cdot u\colon\ ?[T].U} \qquad \text{[THR],[CAT]}$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta\cdot u\colon T_i \qquad \forall i\in I}{\Gamma \vdash\ u \triangleright \{l_i\colon P_i\}_{i\in I} \triangleright \Delta\cdot u\colon \&\{l_i\colon T_i\}_{i\in I}} \qquad \frac{\Gamma \vdash P \triangleright \Delta\cdot u\colon T_j \qquad j\in I}{\Gamma \vdash u \triangleleft l_j.P \triangleright \Delta\cdot u\colon\ \oplus\{l_i\colon T_i\}_{i\in I}} \qquad \text{[BR], [SEL]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta \otimes \Delta'} \qquad \frac{\Gamma\cdot a\colon S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta} \qquad \text{[CONC],[NRES]}$$

$$\frac{\Gamma \vdash e\colon \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' \triangleright \Delta} \qquad \frac{\Delta \vdash e\colon \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' \triangleright \Delta} \qquad \text{[IFN],[IFC]}$$

$$\frac{}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \frac{\Gamma \vdash \tilde{e}\colon \tilde{S}}{\Gamma\cdot X\colon \tilde{S}\tilde{T} \vdash X[\tilde{e}\tilde{u}] \triangleright \Delta\cdot \tilde{u}\colon \tilde{T}} \qquad \text{[INACT], [VAR]}$$

$$\frac{\Gamma\cdot X\colon \tilde{S}\tilde{T}\cdot \tilde{x}\colon \tilde{S} \vdash P \triangleright \tilde{y}\colon \tilde{T} \qquad \Gamma\cdot X\colon \tilde{S}\tilde{T} \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\tilde{y}) = P\ \mathsf{in}\ P' \triangleright \Delta} \qquad \text{[DEF]}$$

Figure 3: Type system for top level syntax

In rule [THR], process $u![v].P$ is typed with channel environment $\Delta\cdot u\colon\ ![T].U\cdot v\colon T_1$ describing the fact that channel variable v of type T_1 is sent on channel variable u of type $![T].U$. Similarly to [3], in passing channels we split the capabilities and let the continuation process P use the passed channel at type T if is sent at type end, or at type end if it is sent at type T (in such case the value can be used only for comparing it or sending it for comparison), as witnessed by knowing v at type $T = T_1 \otimes T_2$ in the typing environment of the hypothesis. In rule [CONC], the channel environment of $P \mid Q$ is the merge of the environments Δ and Δ' for the two processes, provided the operation is defined: if a value is both in the domain of Δ and Δ', then at least one of the typings is end. Notice that, differently from many works on session and linear types, in rules [INACT] and [VAR] we do not require the type environment to contain only depleted resources (of type end). We discuss this point further in the conclusions. The remaining rules are as in [7] .

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

Theorem 2 (Type Safety)**.** *Let $\Gamma \vdash P \triangleright \Delta$ with P a top-level process. If $P \longrightarrow^* Q$, then Q is not an error.*

Notice that the type system we are considering is meant to type top level processes only. The straightforward extension of the type system to the runtime syntax, which amounts to allow runtime channels k as opposed to channel variables x both in linear environments and in processes, does not satisfy subject reduction, as the following example shows (notice that $a\colon \langle ?[\mathsf{bool}].\mathsf{end}\rangle \vdash P \triangleright \emptyset$):

$$P = \mathsf{request}\ a(x).x![\mathsf{true}].\mathbf{0} \mid \mathsf{accept}\ a(y).y?(z).\mathbf{0} \longrightarrow (\nu k)(k![\mathsf{true}].\mathbf{0} \mid k?(z).\mathbf{0}),$$

The parallel composition on the right-hand side is not typable, for both operands will have k on the channel environment at a type different from end, making the merge of the environments in rule [CONC] not defined.

Outline of the Proof of Type Safety for Top Level Processes, Theorem 2. We define a mapping $[\![\cdot]\!]$ from a different runtime language to the base runtime language, and we prove that typable top-level processes do not reduce to errors by showing operational and error correspondence results .

The syntax for the *double binder runtime language* is obtained by replacing the binder $(\nu k)P$ of the runtime language with a *double binder* $(\nu cd)R$ [], where c,d are distinct identifiers [5]. The syntax of double binder processes R includes processes P of Figure 1; we define a reduction relation $R \hookrightarrow R'$ and a typing system for double binder processes. The encoding $[\![\cdot]\!]$ maps double binder processes $(\nu cd)R$ in runtime processes $(\nu k)Q$ such that both channel-ends c,d related by the double binder are mapped into a single runtime channel k bound in Q.

Let P be a typed top level process, and assume $P \longrightarrow^* Q$. We need to show that Q is not an error. The proof proceeds with the following steps.

1. *Typing Correspondence*. We show that typed top level processes are typed double binder processes.
2. *Operational Correspondence* We prove that the mapping $[\![\cdot]\!]$ is sound:

$$P \to^* Q \implies \exists R \text{ such that } P \hookrightarrow^* R \text{ and } [\![R]\!] \equiv Q$$

3. *Type Safety of Double Binder Language*. We provide for the subject reduction of the double binder runtime language and in turn we prove that typable double-binder processes do not reduce to errors. We apply this result to (1) and infer that

$$R \text{ not an error}$$

4. *Error Correspondence*. We prove that

$$R \text{ not an error} \implies [\![R]\!] \text{ not an error}$$

5. *Error Congruence*. We glue (2) and (4) and show that

$$([\![R]\!] \equiv Q \text{ and } [\![R]\!] \text{ not an error}) \implies Q \text{ not an error}$$

The remainder of paper details the various steps.

The double binder runtime language. As mentioned above we introduce a *double binder* $(\nu cd)R$ construct, that must not be confused with notation $(\nu c,d)R$ often used in the literature to indicate the process $(\nu c)(\nu d)R$. We underline that the top level language is a sub language of both runtime languages.

The semantics of the new language is defined over *configurations* of the form $\sigma \diamond R$ where σ is an irreflexive, symmetric and functional binary relation over channels, which we call *channel-end connection*. We let $\mathrm{dom}(\sigma)$ be the domain of σ. We write $\sigma \cdot (c,d)$ to indicate the union $\sigma \cup (c,d)$, whenever $\{c,d\} \cap \mathrm{dom}(\sigma) = \emptyset$. We let $\equiv_{\mathsf{db}}$ be the structural congruence relation for double binder processes, which is obtained straightforwardly from the structural rules for runtime processes e.g. $(\nu cd)R \mid R' \equiv_{\mathsf{db}} (\nu cd)(R \mid R')$.

The interesting rules for the reduction are below; the remaining rules are obtained by extending the rules in Figure 2 to configurations.

$$\sigma \diamond (\mathsf{accept}\ a(x).R \mid \mathsf{request}\ a(y).R) \;\to\; \sigma \diamond ((\nu cd)(R[c/x] \mid R'[d/y])) \qquad [\textsc{LinkD}]$$
$$c\,\sigma\,d \wedge e \downarrow v \;\Rightarrow\; \sigma \diamond (c![e].R \mid\ d?(y).R') \;\to\; \sigma \diamond (R \mid R'[v/y]) \qquad [\textsc{ComD}]$$
$$c\,\sigma\,d \;\wedge\; j \in I \;\Rightarrow\; \sigma \diamond (c \triangleleft l_j.R \mid d \triangleright \{l_i \colon R_i\}_{i \in I}) \;\to\; \sigma \diamond (R \mid\ R_j) \qquad [\textsc{LabelD}]$$
$$\sigma \cdot (c,d) \diamond R \to \sigma \cdot (c,d) \diamond R' \;\Rightarrow\; \sigma \diamond (\nu cd)R \to \sigma \diamond (\nu cd)R' \qquad [\textsc{ScopD}]$$

The new rule [LinkD] creates, for two processes $\mathsf{accept}\ n(x).R$ and $\mathsf{request}\ n(y).R'$ ready to engage in a session, not one but two channel-ends c and d, one for each partner. The connection between channel-ends c and d is made explicit in the binding (νcd). Rule [ComD] performs value passing from the sending party $c![e].R$ to the receiving partner $d?(y).R'$, provided c and d are related by σ.

The types for the double binder runtime language are those of the base language. The type system now has judgments on configurations of the form $\Gamma \vdash \sigma \diamond R \triangleright \Delta$ with $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\sigma)$.

We add a single new rule w.r.t. the type system for top-level processes of Figure 3:

$$\frac{\Gamma \vdash \sigma \cdot (c,d) \diamond R \triangleright \Delta \cdot c \colon T \cdot d \colon \overline{T}}{\Gamma \vdash \sigma \diamond (\nu cd)R \triangleright \Delta} \qquad [\textsc{CResD}]$$

In rule [CResD], the double binding (νcd) gives raise to a (c,d) entry in σ, in line with the operational semantics. The rule types a channel double-binder, making sure that the two ends, c and d, of a channel have dual types.

Subject reduction and type-safety hold only for balanced environments [11].

Definition 3. *Let σ be an channel-end connection and Δ be a typing such that* $\mathrm{fc}(\Delta) \subseteq \mathrm{dom}(\sigma)$. *We say that Δ is* balanced *by σ whenever $c\,\sigma\,d$ and $\{c,d\} \subseteq \mathrm{dom}(\Delta)$ implies that or (i) $\Delta(c) = \overline{\Delta(d)}$ or (ii) $\Delta(c) \otimes \Delta(d) \downarrow$.*

$$
\begin{array}{rcl}
[\![\sigma \diamond \mathsf{accept}\ u(x).R]\!]_\Sigma & = & \mathsf{accept}\ [\![u]\!]_\Sigma(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond \mathsf{request}\ u(x).R]\!]_\Sigma & = & \mathsf{request}\ [\![u]\!]_\Sigma(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u?(x).R]\!]_\Sigma & = & [\![u]\!]_\Sigma?(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u![e].R]\!]_\Sigma & = & [\![u]\!]_\Sigma![[\![e]\!]_\Sigma].[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond (\nu a)R]\!]_\Sigma & = & (\nu a)[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond (\nu cd)R]\!]_\Sigma & = & (\nu k)[\![\sigma \cdot (c,d) \diamond R]\!]_{\Sigma \cdot (c \to k, d \to k)} \\
[\![\sigma \diamond u \lhd l_j.R]\!]_\Sigma & = & [\![u]\!]_\Sigma \lhd l_j.[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u \rhd \{l_i \colon R_i\}_{i \in I}]\!]_\Sigma & = & [\![u]\!]_\Sigma \rhd \{l_i \colon [\![\sigma \diamond R_i]\!]_\Sigma\}_{i \in I} \\
[\![\sigma \diamond \mathsf{if}\ e\ \mathsf{then}\ R\ \mathsf{else}\ R']\!]_\Sigma & = & \mathsf{if}\ [\![e]\!]_\Sigma\ \mathsf{then}\ [\![R]\!]_\Sigma\ \mathsf{else}\ [\![R]\!]'_\Sigma \\
[\![\sigma \diamond R \mid R']\!]_\Sigma & = & [\![\sigma \diamond R]\!]_\Sigma \mid [\![\sigma \diamond R']\!]_\Sigma \\
[\![\sigma \diamond X[\tilde{e}]]\!]_\Sigma & = & X[[\![\tilde{e}]\!]_\Sigma] \\
[\![\sigma \diamond \mathsf{def}\ \{X_i(\tilde{x}_i) = R_i\}_{i \in I}\ \mathsf{in}\ R]\!]_\Sigma & = & \mathsf{def}\ \{X_i(\tilde{x}_i) = [\![\sigma \diamond R_i]\!]_\Sigma\}_{i \in I}\ \mathsf{in}\ [\![\sigma \diamond R]\!]_\Sigma
\end{array}
$$

Figure 4: The encoding of double binder configurations into runtime processes.

The proof of the subject reduction is involved, because of session passing and of the rule for typing composition that merges possibly overlapping encodings.

Theorem 4 (Subject Reduction for the Double Binder Language)**.** *Let* $\Gamma \vdash \sigma \diamond R \rhd \Delta$ *with* Δ *balanced by* σ. *If* $\sigma \diamond R \to \sigma \diamond R'$, *then there is* Δ' *balanced by* σ *s.t.* $\Gamma \vdash \sigma \diamond R' \rhd \Delta'$.

Theorem 5 (Type Safety for the Double Binder Language)**.** *Let* $\Gamma \vdash \sigma \diamond R \rhd \Delta$ *with* Δ *balanced by* σ. *If* $\sigma \diamond R \to^* \sigma \diamond R'$, *then* $\sigma \diamond R'$ *is not an error.*

Proof. Standard, by using Subject Reduction (Theorem 4). □

From the Double Binder Runtime to the Base Runtime Language. In order to state the correspondence between runtime processes, we define a mapping from the double binder language into the base language. A *forget-distinction function* over a end-channel configuration σ is an injective partial function from channel-ends c,d to runtime channels k such that $c\,\sigma\,d$ and $\Sigma(c) = k$ implies that $\Sigma(d) = k$.

The mapping $[\![\cdot]\!]_\Sigma$ from double binder configurations into runtime processes is defined in Figure 4. We let $[\![e]\!]_\Sigma = \Sigma(e)$ whenever e is a channel, and e otherwise. The compilation of a double binder configuration $\sigma \diamond (\nu cd)R$ with parameter Σ in a runtime process Q involves the generation of a new channel k bound in the encoding of $\sigma \diamond R$ with parameter Σ', where Σ' is obtained by extending Σ with the entries $\Sigma'(c) = k$ and $\Sigma'(d) = k$, so that the free occurrences of c,d in the continuation R will be translated to the channel k.

First, we need a lemma to export values outside the encoding.

Lemma 6. $[\![\sigma \diamond R[v/x]]\!]_\Sigma = [\![\sigma \diamond R]\!]_\Sigma[[\![v]\!]_\Sigma/x]$.

Proof. By induction on the structure of $[\![\sigma \diamond R]\!]_\Sigma$. □

The core of the proof of operational correspondence is summarized the following lemma.

Lemma 7. *If* $[\![\sigma \diamond R]\!]_\Sigma \to Q$, *then there is a double binder process* R' *such that* $\sigma \diamond R \to \sigma \diamond R'$ *with* $[\![\sigma \diamond R']\!]_\Sigma \equiv Q$.

Proof. By induction on the length of the inference $[\![\sigma \diamond R]\!]_\Sigma \rightarrow Q$. The link and communication cases build on lemma 6. □

The next theorem says that structural congruence is preserved by $[\![\cdot]\!]$.

Theorem 8 (Congruence Correspondence)**.** *If $R' \equiv_{\mathsf{db}} R$ then $[\![\sigma \diamond R]\!]_\Sigma \equiv [\![\sigma \diamond R']\!]_\Sigma$.*

Proof. By case analysis on the rules for $\equiv_{\mathsf{db}}$. □

The next theorem establishes the soundness of the encoding.

Theorem 9 (Operational Correspondence)**.** *Let P be a top-level process of Figure 1. If there is a runtime process Q such that $P \rightarrow^n Q$, for $n \geq 0$, then there is a double binder process R such that $\emptyset \diamond P \rightarrow^n \emptyset \diamond R$ with $[\![\emptyset \diamond R]\!]_\emptyset \equiv Q$.*

Proof. By induction on the length n of reduction. When $n = 0$, we have that $Q = P$ and we are done since $[\![\emptyset \diamond P]\!]_\emptyset = P$. The induction step is proved by using Theorems 7 and 8. □

The following lemma says that the mapping $[\![\cdot]\!]_\Sigma$ preserve prefixes.

Lemma 10. *Let $Q = [\![\sigma \diamond R]\!]_\Sigma$. We have that Q is a k-processes if and only if R is a c-process and $\Sigma(c) = k$.*

The close the proof of our main result, we need to show that $[\![\cdot]\!]_\Sigma$ maps correct processes in correct processes.

Theorem 11 (Error Correspondence)**.** *Let $\sigma \diamond R$ be a configuration and assume Σ is a forget-distinction function over σ. If $\sigma \diamond R$ is not an error, then $[\![\sigma \diamond R]\!]_\Sigma$ is not an error.*

Proof. Without loss of generality, let

$$\begin{array}{rcl} R & \equiv & (\nu \tilde{a})(\nu \widetilde{cd}) \mathsf{def}\ \{X_i(\tilde{x}_i) = R_i\}_{i \in I}\ \mathsf{in}\ R_1 \mid \cdots \mid R_n \\ [\![\sigma \diamond R]\!]_\Sigma & \equiv & (\nu \tilde{a})(\nu \tilde{k}) \mathsf{def}\ \{X_i(\tilde{x}_i) = Q_i\}_{i \in I}\ \mathsf{in}\ Q_1 \mid \cdots \mid Q_n \end{array}$$

where $\sigma' = \sigma \cdot (\tilde{c}, \tilde{d})$, $\Sigma' = \Sigma \cdot (\widetilde{cd} \rightarrow \tilde{k})$ and $Q_i = [\![\sigma' \diamond R_i]\!]_{\Sigma'}$, for $j \in 1, \ldots, n$.

We proceed by induction on n and show that if there are distinct $i, j \in 1, \ldots, n$ such that Q_i, Q_j are k-processes then both (a) and (b) hold:

(a) $Q_i \mid Q_j$ is a k-redex

(b) if there exists $r \in 1, \ldots, n$, $r \neq i, j$, such that Q_r is a k'-process, then $k' \neq k$.

From this and congruence correspondence (Theorem 8) we easily obtain that $[\![\sigma \diamond R]\!]_\Sigma$ is not an error. □

Discussion. In [7] we have introduced an extension of the first session typing system [10] that allows higher-order session communication. The calculus and typing system we present here extends these works by admitting session passing or delegation by using the standard communication rule of pi calculus and by providing for the comparison of channels. Besides the pragmatical interest of comparing values, we believe our result makes the theory of session types more general by relaxing the rule for sending channels, and allowing type more processes. Caires and Vieira [3] present a more flexible merge operation on types; we intend to pursue further investigation along these lines.

Starting from [6], many works on session types use polarized channels in order to achieve subject reduction in the presence of session passing or to avoid the runtime checking of free names (e.g., [1, 2, 8, 11]). In [6] the authors considered a typed pi calculus with branching and selection where channel-ends are decorated with polarities; communication and branching/selection occurs on channels with opposite polarities. While the type theory is polarized, polarities of processes can be inferred by a typechecking algorithm, provided the type environments in the rule for parallel composition are disjoint. In contrast, we avoid to use polarities in the type theory and admit overlapping environments in typing compositions; the connection between channel ends are relegated to the proof of type safety and are completely hidden to the programmer. This seems to us more suitable for synchronous implementations; in contrast asynchronous systems should use polarized channels for abstracting low level buffers [5]. We conjecture that our proof should be valid for most session-based calculi with accept and request primitives.

Differently from our framework, many systems for session [6, 7, 11] and linear [9] types require the inert process to be typed under an environment containing depleted resources (of type end). We think that the very technical reason is to let the structural rule $P \mid \mathbf{0} \equiv P$ preserve typing: adding non-empty capabilities to the type environment of P can break typability. For instance for $P = k?()$ in [7] we can have $\emptyset \vdash \mathbf{0} \triangleright k :![T]$ and $\emptyset \vdash P \triangleright k :?[T]$ and in turn $\emptyset \vdash P \mid \mathbf{0} \triangleright k : \perp$, while $\emptyset \not\vdash P \triangleright k : \perp$. By contrast, in our system non-empty capabilities can be added in compositions only to empty capabilities, because of the merge operation; the counter-example above is not typable since P is using the input capability. We believe that our general rule for the inert process should be sound for most systems typing a parallel composition with similar constraints on capabilities, for instance systems where the composed environments are disjoint.

Acknowledgments. This work has been partially sponsored by the project SENSORIA, IST–2005–016004.

References

[1] Roberto Bruni and Leonardo Gaetano Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST*, pages 100–115, 2008.

[2] Roberto Bruni, Rocco De Nicola, Michele Loreti, and Leonardo Gaetano Mezzina. Provably correct implementations of services. In *Trustworthy Global Computing*, 2008.

[3] L. Caires and H. T. Vieira. Conversation types. In *ESOP'09*, LNCS. Springer-Verlag, 2009.

[4] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP*, LNCS, pages 328–352. Springer-Verlag, 2006.

[5] Simon Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. Submitted, 2008.

[6] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[8] Marija Kolundzija. Security types for sessions and pipelines. In *Web-Services and Formal Methods*, 2008.

[9] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[10] 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.

[11] 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*, volume 171(4) of *ENTCS*, pages 73–93, 2007.