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

Type Safety without Subject Reduction for Session Types

Vasco T. Vasconcelos[1] and Marco Giunti[2] and Nobuko Yoshida[3] and Kohei Honda[4]

[1] *Department of Informatics, University of Lisbon, Portugal*
[2] *Department of Planning, University IUAV of Venice, Italy*
[3] *Department of Computing, Imperial College of London , UK*
[4] *Department of Computer Science, Queen Mary University of London, UK*

Received 20 February 2010

We present a simple and intuitive type system for a variant of the pi calculus with sessions where typable processes are guaranteed to be exempt from runtime errors. The type system, meant to type programs, cannot be directly used to type the runtime language. Hence, type-safety cannot be proved directly, via Subject Reduction. Instead we define a different language enjoying Subject Reduction and prove our result by suitable correspondences between the two languages. The exercise shows that polarities, now widely used in binary session types, are justified solely by technical reasons, and need not be exposed to programmers neither in the operational semantics nor in the type system.

1. Introduction

In the beginning there was a language and there was a type system and the type system was sound with respect to the operational semantics and typable terms were know to be exempt from a suitable class of errors. In one word, all went well. The language was a variant of the pi calculus equipped with primitives for channel creation and channel communication. The type system carefully distinguished names, meant to be shared by an arbitrary number of processes, from channels, to be used by at most two processes. From names, and with the help of two dual primitives, accept and request, channels were created. Channels were then used, in a continuous series of interactions, by exactly two partners. The type system would then record in a single type the series of interactions the channel would engage in (HVK98). All went well.

But the language had a limitation. Session passing, often called delegation, was restricted to bound output: the sending party and the receiving party had to agree on which channel to pass. If we denote by $k![k'].P$ the process that sends channel k' on channel k and continues as process P, and write $k?(k').Q$ for the process that receives channel k' on channel k and continues as process Q, then the reduction rule for bound session passing can be written as follows. Notice that occurrences of k' in P are free

whereas those in Q are bound.

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

This limitation was soon perceived by several authors, who hastily replaced bound output by free output. The result was a language identical to the original one, with a type system identical to the original system, but with a slight variation in the operational semantics. The new rule for session delegation was now a lot more flexible than the original.

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

The free output rule effectively allowed passing an arbitrary channel k' that would replace the bound variable x in the continuation of the receiving process Q. The exact difference between the two rules comes to light when we consider transmitting a channel already known by the receiver. The former rule would not allow such a reduction to happen for, if k' is free in Q, then the receiving process must be of the form $k?(k'').Q$, which cannot be renamed to a process of the form $k?(k').Q'$, in order to match the pattern required by the rule for bound output.

The new system was soon discovered to be unsound. The delicate balance between the operational semantics and the type system of the original language were broken by fiddling with the operational semantics alone, leaving untouched the rest of the system. And the counterexample came exactly from were the two systems differ: passing a channel already know to the recipient. The process on the left, below, is typable in a certain typing context, whereas that on the right is not typable in the same context, for the receiving thread now holds the two ends of a same channel (DCMYD06; YV07).

$$k![k'].\mathbf{0} \mid k?(x).x![a].k'?(y).\mathbf{0} \;\rightarrow\; k'![a].k'?(y).\mathbf{0}$$

A fix to the problem was soon produced; it encompassed adjusting other parts of the system to fit the new reduction rule. The invention was to syntactically distinguish the two ends of a same channel: for a given channel k, one end was to be called k^+, the other k^-; the $+/-$ annotations become known as polarities. The channel creation reduction rule now distributed one channel end to the request party and another, syntactically distinguished, end to the accept party. The new operational semantics dealt with channel-polarity pairs. The type system had to be adjusted as well. While before typing contexts conventionally described a map from channels into types, the new typing environments mapped channel-polarity pairs into types, crucially allowing two entries for the same channel, one denoted by k^+, the other by k^-. And problem with receiving a known channel was fixed because one would receive the other end of the known channel, and typing contexts allow two distinct entries, one for k'^+ and for k'^- (GH05).

$$k^+![k'^+].\mathbf{0} \mid k^-?(x).x![a].k'^-?(y).\mathbf{0} \;\rightarrow\; k'^+![a].k'^-?(y).\mathbf{0}$$

The question remained: does the original type system guarantee absence of runtime errors for typable processes, in the presence of free output, and in spite of the unsoundness of the type system? The counterexample cannot lead us to think otherwise. This paper shows that one can pick the original language, the original operational semantics

with bound replaced by free output, and a simple variant of the original type system, and still guarantee type safety. Here is how we proceed. We carefully distinguish the programming language, the language available to programmers, from the runtime language. Programmers do not deal with channels directly but manipulate these via conventional programming variables. Channels are runtime entities only: they are created at runtime and eventually replace variables in the source code. The type system is defined for the programming language only, hence knows nothing of channels, and is thus not suitable to type the runtime language.

Typable programs are nevertheless shown not to suffer from runtime errors. How? Certainly not via the most common strategy—subject-reduction and type safety—for there is no type system for the runtime language. Instead we resort to a second language for which we define a type system that enjoys subject-reduction and type safety for an appropriate notion of error. All there remains is to prove two correspondences between the two languages: an operational correspondence from the programming language into the auxiliary language, and an error correspondence between the auxiliary language and the programming language.

How does this auxiliary language look like? Well, it turns out to be the language with polarities. At this point the interested reader may wonder "what is the point of the exercise if the language with polarities is still there?" The language with polarities is a simple technical device, encapsulated in a proof, the proof that typable programs are exempt from errors. Our language, programming or runtime, the operational semantics or the type system at no point use polarities. Programmers willing to understand the operational semantics of the language they are programming with are not exposed to artificial artifacts, such as the distinction between ends of a same channel, a subterfuge employed with the sole purpose of getting the technical details right. And we firmly believe that there are alternatives, more elegant and mind-opening, to polarities.

Given the above introduction, the plan for the the paper should be self-evident. We start by describing the language and its type system (Section 2). We then introduce the operational semantics, state what we understand by runtime errors and enunciate the main result "Well typed programs do not go wrong" (Section 3). The rest of the paper is dedicated to the proof of this result. We introduce the language with polarities, its operational semantics and type system; we present the notion of runtime errors for this language, and prove type safety via subject-reduction (Section 4). To complete the proof we show the two corresponding results: operational and error, in opposite directions (Section 5). We conclude the paper by describing related work and summarising our findings.

2. The Syntax and Type System for Programs

This section describes the syntax and the type system for our programming language. We distinguish the *programming language* from the *runtime language*, introduced in the next section. The former constitutes the language programmers program with; the latter includes constructs required to describe the operational semantics, but that need not be accessible to programmers.

P	$::= \mathsf{request}\ a(x).P$	session request
	$\mid \mathsf{accept}\ a(x).P$	session acceptance
	$\mid v![v].P$	value sending
	$\mid v?(x).P$	value reception
	$\mid v \lhd l.P$	label selection
	$\mid v \rhd \{l_i\colon P_i\}_{i\in I}$	label branching
	$\mid \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ P$	conditional branch
	$\mid P \mid P$	parallel composition
	$\mid \mathbf{0}$	inaction
	$\mid (\nu n)P$	hiding
	$\mid (\mathsf{rec}\ X(\tilde{x}).P)[\tilde{v}]$	recursive process
	$\mid X[\tilde{v}]$	recursive call
v	$::= x \mid \mathsf{true} \mid \mathsf{false}$	values
n	$::= x$	binder

Fig. 1. The syntax of programs.

2.1. *Syntax*

The programming language relies on a few base sets: *variables* (sometimes called *names*) ranged over by a, b and also x, y, *labels* ranged over by l, and *process variables* ranged over by X. The values of our language are names and the boolean literals true and false. The syntax is in Figure 1.

The two first processes in the grammar, $\mathsf{accept}\ a(x).P$ and $\mathsf{request}\ a(x).P$, denote two partners sharing a common name a and willing to establish a communication session; variable x denotes the channel on which the session will be conducted using the next four constructors. Channels are runtime entities to be introduced in Section 3. Processes of the form $u![v].P$ and $u?(x).P'$ exchange a value on channel u. The former *sends* a value v and continues as P; the latter *receives* a value that replaces variable x in the continuation P'. The next two processes, $u \lhd l.P$ and $u \rhd \{l_i\colon P_i\}_{i\in I}$, exchange a label on u. The former *selects* a label l, whereas the latter *branches* on the received label (which must be one of the l_i) and continues with process P_i. In all these cases, the type system for our language makes sure that u is not a boolean value.

The *conditional* is standard and it takes advantage of the only base values we have equipped our language with. *Parallel composition* $P \mid P'$ runs processes P and P' in parallel, and $\mathbf{0}$ denotes the *terminated* process. Processes of the form $(\nu n)P$ *restrict the scope* of identifier n (a name in the programming language) to process P. We have equipped our language with a syntactic category for binders, anticipating the needs of the runtime language. Finally, processes $(\mathsf{rec}\ X(\tilde{x}).P)[\tilde{v}]$ and $X[\tilde{v}]$ allow for recursive definitions, providing for potentially unbounded behaviour.

Our running example is that of an idealized File Transfer Protocol (FTP) server, relying on an authentication server to authenticate potential clients and comprising a fixed pool

of threads to interact with clients, taken from (HVK98). We start by introducing a typical client, a client that needs to upload a *file* on the server. Such a client starts by requesting a session with the server, then sends his username and password. If the authentication succeeds, the client selects option "put", sends the file, and completes the session by selecting option "quit".

$$\mathit{Client} = \mathsf{request}\ a(z).z![\text{“name”}].z![\text{“pass”}].z \triangleright \{\text{sorry}\colon \mathbf{0}, \text{welcome}\colon z \triangleleft \text{put}.z![\mathit{file}].z \triangleleft \text{quit}.\mathbf{0}\}$$

The server is composed of a demon and a pool of m threads, sharing a name b for internal communication; we denote by $\Pi_m P$ the parallel composition of m copies of process P.

$$\mathit{Server} = (\nu b)(\mathit{Ftpd}[b] \mid \Pi_m \mathit{FtpThread}[b])$$

The demon accepts incoming connections from clients by listening on its address a; once established, each such connection is denoted by variable x. It then requests one of its threads using the shared name b. When such a request is granted by an (idle) thread, thus establishing a new session y, the client's session x is delegated to the thread via channel y. The FTP demon then loops waiting for another client connection.

$$\mathit{Ftpd} = \mathsf{rec}\ X(b).(\mathsf{accept}\ a(x).\mathsf{request}\ b(y).y![x].X[b])$$

Each FTP thread starts by accepting a request from the demon on name b, thus establishing a new channel y on which the client's session x is received. The thread reads the client's username and password, and proceeds to the authentication phase.

$$\mathit{FtpThread} = \mathsf{rec}\ Y(b).(\mathsf{accept}\ b(y).y?(x).x?(\mathit{username}).x?(\mathit{passwd}).\mathit{Authenticate})$$

To authenticate a client, a new session must be established, this time with the authentication server *as*. After sending the username and the password, the thread waits for an answer that may come in one of two forms: valid or invalid. Unsuccessful authentications are forwarded to the client, and the thread loops, thus becoming free to accept further requests from the server. Successful authentications are also made known to the clients, but this time the FTP thread provides some basic FTP operations.

$$\begin{aligned}\mathit{Authenticate} = {} & \mathsf{request}\ \mathit{as}(z).z![\mathit{username}].z![\mathit{passwd}].\\ & z \triangleright \{\text{invalid}\colon x \triangleleft \text{sorry}.Y, \text{valid}\colon x \triangleleft \text{welcome}.\mathit{Actions}[x]\}\end{aligned}$$

The last piece of code is a loop that repeatedly reads the client's FTP operation until a quit is received, on which occasion the thread becomes free to accept further requests from the server.

$$\begin{aligned}\mathit{Actions} = \mathsf{rec}\ Z(x).(x \triangleright \{ & \text{get}\colon x?(\mathit{filename}).\ldots.Z[x],\\ & \text{put}\colon x?(\mathit{file}).\ldots.Z[x],\\ & \text{quit}\colon \ldots Y\})\end{aligned}$$

2.2. *Types*

$$
\begin{array}{lll}
S & ::= \mathsf{bool} \mid \langle T \rangle & \text{sort} \\
T & ::= ?[\alpha].T \mid ![\alpha].T \mid \oplus\{l_i\colon T_i\}_{i\in I} \mid \&\{l_i\colon T_i\}_{i\in I} \mid \mathsf{end} \mid t \mid \mu t.T & \text{type} \\
\alpha & ::= S \mid T & \text{sort or type}
\end{array}
$$

Fig. 2. The syntax of types and sorts.

$$
\begin{array}{llll}
\overline{?[\alpha].T} = ![\alpha].\overline{T} & \overline{\oplus\{l_i\colon T_i\}_{i\in I}} = \&\{l_i\colon \overline{T_i}\}_{i\in I} & \overline{\mathsf{end}} = \mathsf{end} & \\
\overline{![\alpha].T} = ?[\alpha].\overline{T} & \overline{\&\{l_i\colon T_i\}_{i\in I}} = \oplus\{l_i\colon \overline{T_i}\}_{i\in I} & \overline{\mu t.T} = \mu t.\overline{T} & \overline{t} = t
\end{array}
$$

Fig. 3. The dual function on session types.

We distinguish *sorts*, ranged over by S, from *types*, ranged over by T; we let α range over sorts and types. The syntax of types and sorts in Figure 2, where we rely on a base set of *type variables*, ranged over by t.

Sorts are assigned to values: a boolean value has type bool, a name has a type describing the interactive sessions it may engage upon. The interactive session, in turn, is described by a type. 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. Type end represents a channel on which no further interaction is possible. 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.

We include recursive session types $\mu t.T$, which are required to be contractive, i.e., containing no subexpression of the form $\mu t_1 \ldots \mu t_n.t_i$, and take an *equi-recursive* view of types, not distinguishing between a type $\mu t.T$ and its unfolding $T[\mu t.T/t]$. Types are understood up to type equivalence, so that, for example, in a typing derivation, types $\mu t.T$ and $T[\mu t.T/t]$ can be used interchangeably.

Duality is a central concept in the theory of session types. The function $\overline{T}$, defined in Figure 3, yields the canonical dual of a session type T. We write $T = \overline{T'}$ on the understanding that we are always working up to type equivalence, so that, for example, $\mu t.?[\mathsf{bool}].t.$ is dual of $![\mathsf{bool}].\mu t.![\mathsf{bool}].t.$

Analysing the types in our running example, one can assign to the authentication server the type of a name whose sessions start by accepting two strings (*username* and *passwd*) and then answer with one of two possibilities—valid or invalid—after which the protocol comes to an end.

$$as\colon \langle ?[\mathrm{String}].?[\mathrm{String}].\oplus\{\mathrm{invalid}\colon \mathsf{end}, \mathrm{valid}\colon \mathsf{end}\}\rangle$$

The type of the FTP server is that of a name whose sessions start by accepting two strings (*username* and *passwd*) and then must be ready to receive the result of the authentication. A "sorry" terminates the protocol, whereas a "welcome" result leads to

$$\Gamma \vdash \mathsf{true}\colon \mathsf{bool} \qquad \Gamma \vdash \mathsf{false}\colon \mathsf{bool} \qquad \Gamma, x\colon S \vdash x\colon S \qquad [\textsc{True}],[\textsc{False}],[\textsc{Name}]$$

Fig. 4. Typing rules for values.

a loop of FTP operations: put, get and quit.

$$\begin{aligned} a\colon \langle T\rangle \quad & \text{where} \\ T = {} & ?[\mathrm{String}].?[\mathrm{String}]. \\ & \oplus\{\mathrm{sorry}\colon \mathsf{end}, \\ & \quad \mathrm{welcome}\colon \mu t.\&\{\mathrm{put}\colon ?[\mathrm{File}].t, \mathrm{get}\colon ?[\mathrm{String}].![\mathrm{File}].t, \mathrm{quit}\colon \mathsf{end}\}\} \end{aligned}$$

Notice that by reading the type of the FTP server, clients cannot infer the delegation process that happens between the demon and its threads, nor the sub-session that is initiated with the authentication server.

Finally, session delegation on name b, between the demon and its threads, is governed by a type $\langle ![T].\mathsf{end}\rangle$ where T is the type of the complete FTP session above.

Concentrating on the client side, we see that his session may be governed by a type that starts by sending two strings and then expects one of two options: "sorry" or "welcome". In the former case the session is terminated; in the latter, the client embarks on a series of operations, so that the session *as seen from the client's side* can be described by:

$$\begin{aligned} \overline{T} = {} & ![\mathrm{String}].![\mathrm{String}]. \\ & \&\{\mathrm{sorry}\colon \mathsf{end}, \\ & \quad \mathrm{welcome}\colon \mu t.\oplus\{\mathrm{put}\colon ![\mathrm{File}].t, \mathrm{get}\colon ![\mathrm{String}].?[\mathrm{File}].t, \mathrm{quit}\colon \mathsf{end}\}\} \end{aligned}$$

This is certainly not the only type for the client; a more concise type would exclude the "get" branch and the recursive type. This is however a possible type for the client, and the one that matches that of the server.

2.3. *Typing Assignment*

The type system distinguishes the sorts assigned to names and to boolean values, from the types assigned to channels. Types are treated linearly; a map from channels and variables into types T is denoted by Δ. Sorts are treated classically; a map from names to sorts S is denoted by Γ. We also record in Γ assignments of process variables X to pairs of sequences $\tilde{S}\tilde{T}$ of sorts (first) and types (then).

We write $\Delta, x\colon T$ to denote the typing environment Δ extended with an entry $x\colon T$ when $x \notin \mathrm{dom}(\Delta)$. We generalize the notation to type environments and denote the disjoint map union of Δ and Δ' by Δ, Δ'.

Type assignment system for the programming language is in Figures 4 and 5. The rules in Figure 4 describe type assignment to values. The first two rules are standard for boolean values. The last rule says that sorts for non-linear values are taken from the sort environment Γ.

The rules in Figure 5 describe typing assignment to processes and use sequents of the

$$\frac{\Gamma \vdash a\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta, x\colon T}{\Gamma \vdash \mathsf{accept}\ a(x).P \triangleright \Delta} \qquad \frac{\Gamma \vdash a\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta, x\colon \overline{T}}{\Gamma \vdash \mathsf{request}\ a(x).P \triangleright \Delta} \quad [\textsc{Acc}],[\textsc{Req}]$$

$$\frac{\Gamma \vdash v\colon S \qquad \Gamma \vdash P \triangleright \Delta, u\colon T}{\Gamma \vdash u![v].P \triangleright \Delta, u\colon ![S].T} \qquad \frac{\Gamma, x\colon S \vdash P \triangleright \Delta, u\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta, u\colon ?[S].T} \quad [\textsc{Send}],[\textsc{Rcv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, u\colon U}{\Gamma \vdash u![v].P \triangleright \Delta, u\colon ![T].U, v\colon T} \qquad \frac{\Gamma \vdash P \triangleright \Delta, u\colon U, x\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta, u\colon ?[T].U} \quad [\textsc{Thr}],[\textsc{Cat}]$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta, u\colon T_i \qquad \forall i \in I}{\Gamma \vdash\ u \triangleright \{l_i\colon P_i\}_{i\in I} \triangleright \Delta, u\colon \&\{l_i\colon T_i\}_{i\in I}} \qquad \frac{\Gamma \vdash P \triangleright \Delta, u\colon T_j \qquad j \in I}{\Gamma \vdash u \triangleleft l_j.P \triangleright \Delta, u\colon\ \oplus\{l_i\colon T_i\}_{i\in I}} \quad [\textsc{Br}],[\textsc{Sel}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta, \Delta'} \qquad \frac{\Gamma \vdash v\colon \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ P' \triangleright \Delta} \quad [\textsc{Conc}],[\textsc{If}]$$

$$\Gamma \vdash \mathbf{0} \triangleright \vec{u}\colon \vec{\mathsf{end}} \qquad \frac{\Gamma, x\colon S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu x)P \triangleright \Delta} \quad [\textsc{Inact}],[\textsc{Res}]$$

$$\frac{\Gamma \vdash \tilde{u}\colon \tilde{S}}{\Gamma, X\colon \tilde{S}\tilde{T} \vdash X[\tilde{u}\tilde{v}] \triangleright \vec{w}\colon \vec{\mathsf{end}}, \tilde{v}\colon \tilde{T}} \qquad \frac{\Gamma \vdash \tilde{u}\colon \tilde{S} \qquad \Gamma, X\colon \tilde{S}\tilde{T}, \tilde{x}\colon \tilde{S} \vdash P \triangleright \tilde{y}\colon \tilde{T}}{\Gamma \vdash (\mathsf{rec}\ X(\tilde{x}\tilde{y}).P)[\tilde{u}\tilde{v}] \triangleright \tilde{v}\colon \tilde{T}} \quad [\textsc{PVar}],[\textsc{Rec}]$$

Fig. 5. Typing rules for programs.

form $\Gamma \vdash P \triangleright \Delta$, read as "the sort environment Γ types process P with type environment Δ". To type a process $\mathsf{accept}\ a(x).P$ with a type environment Δ, rule [Acc] checks that name a has a sort of form $\langle T\rangle$, and that process P can be typed with channel environment $\Delta, x\colon T$, where channel variable x is given the session type T contained in the sort for name a. The case for processes $\mathsf{request}\ a(x).P$ is similar, only that this time channel variable x is given the dual $\overline{T}$ of the session type in the type of name of a.

The next two rules, [Send] and [Rcv], type boolean and name sending and receiving. To type an output process $u![v].P$, rule [Send] finds the sort S for the value v and the channel environment $\Delta, u\colon T$ for the continuation P. In the resulting environment, the type T for y is replaced by a type that first sends S and then continues with T. The rule for reception is similar only that this time we read the sort for parameter x from the shared environment Γ, removing the entry $x\colon T$, since x is bound in the conclusion.

Rules [Thr] and [Cat] deal with channel sending and receiving. This time process $u![v].P$ is typed with channel environment $\Delta, u\colon ![T].U, v\colon T$ describing the fact that channel variable v of type T is sent on channel variable u of type $![T].U$. The continuation process P cannot use name v (as witnessed by the absence of v in the typing environment of the hypothesis), and uses u according to the continuation type U. The rule for channel reception is similar, only that the type T for the received channel x is taken from the type $?[T].U$ of u in the conclusion and placed in the typing environment of the hypothesis, alongside with the continuation type U for u.

Rules [Br] and [Sel] deal with branching and selection. To type a branching process each branch P_i is typed individually, yielding Δ environments that differ only on the type T_i for channel u. The resulting type environment builds a type $\&\{l_i\colon T_i\}_{i\in I}$ from the types of the branches. Dually, the rule for selection starts by typing the continuation

New syntactic forms

$$\begin{array}{rll} v & ::= \ldots \mid k & \text{value} \\ n & ::= \ldots \mid k & \text{binder} \end{array}$$

Fig. 6. The runtime language.

$$Q \mid \mathbf{0} \equiv Q \qquad Q \mid Q' \equiv Q' \mid Q \qquad (Q \mid Q') \mid Q'' \equiv Q \mid (Q' \mid Q'')$$
$$(\nu n)Q \mid Q' \equiv (\nu n)(Q \mid Q') \qquad (\nu n')(\nu n)Q \equiv (\nu n)(\nu n')Q \qquad (\nu n)\mathbf{0} \equiv \mathbf{0}$$

Fig. 7. Structural congruence.

P to obtain an environment assigning type T_i to channel u. The conclusion injects this type on a variant type $\oplus\{l_i\colon T_i\}_{i\in I}$, in the environment for the conclusion.

Proceeding to rule [Conc], the channel environment for $P \mid P'$ is the disjoint union of the environments Δ and Δ' for the two processes, reflecting the linear nature of channels in processes P and P'. Contrarily to [Conc], in rule [If] the two branches P and P' share a same channel environment Δ, since at most one of them will be executed. Rule [Res] is the standard rule for name binding, removing from the environment Δ the entry for the bound name.

Rules [PVar] and [Rec] deal with recursive processes, setting the form of the parameters: first the shared values (of sorts $\tilde{S}$), then the channels (of types $\tilde{T}$). For rule [PVar], the type $\tilde{S}\tilde{T}$ of process variable X is read from the shared environment Γ, then value arguments must have sorts $\tilde{S}$, and channels types $\tilde{T}$. In rule [Rec], process P, associated with process variable X, is typed with the sorts $\tilde{S}$ for the shared parameters $\tilde{x}$ read from the shared environment Γ, and the types $\tilde{T}$ for the channel parameters $\tilde{y}$ read from the linear environment Δ. The juxtaposition of these sorts and types yields the type $\tilde{S}\tilde{T}$ associated with process variable X, which is then placed in the shared environment and used to type the body of the recursive process P.

3. The Simple Runtime System

This section describes a runtime system for the language introduced above. In particular it presents the runtime language, the operational semantics, and the notion of error.

3.1. *Syntax, Bindings, and Substitution*

The runtime language requires one more base set, *channels*, ranged over by k. We extend the syntax by including channels in values and by allowing channels to appear in ν-binding positions; we indicate runtime processes with letter Q. The syntax of the runtime language is in Figure 6; it differs from the programmer's language (Figure 1) in that values include channels, so that, for example, $k![\mathsf{true}].\mathbf{0}$ and $x![k].\mathbf{0}$ are also processes, and channels may appear in ν-binding positions so that $(\nu k)(k![\mathsf{true}].\mathbf{0})$ is a process.

The bindings for the runtime language are processes $(\nu n)Q$, which bind occurrences

$$\mathsf{accept}\ a(x).Q \mid \mathsf{request}\ a(y).Q' \ \rightarrow\ (\nu k)(Q[k/x] \mid Q'[k/y]) \qquad [\textsc{Link}]$$

$$k![v].Q \mid\ k?(x).Q' \ \rightarrow\ Q \mid Q'[v/x] \qquad [\textsc{Com}]$$

$$k \lhd l_j.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I} \ \rightarrow\ Q \mid\ Q_j \qquad (j \in I) \qquad [\textsc{Case}]$$

$$\mathsf{if\ true\ then}\ Q\ \mathsf{else}\ Q' \ \rightarrow\ Q \qquad [\textsc{IfT}]$$

$$\mathsf{if\ false\ then}\ Q\ \mathsf{else}\ Q' \ \rightarrow\ Q' \qquad [\textsc{IfF}]$$

$$(\mathsf{rec}\ X(\tilde{x}).P)[\tilde{v}] \ \rightarrow\ P[\tilde{v}/\tilde{x}][\mathsf{rec}\ X(\tilde{x}).P/X] \qquad [\textsc{Rec}]$$

$$Q \rightarrow Q' \ \Rightarrow\ (\nu n)Q \rightarrow (\nu n)Q' \qquad [\textsc{Scop}]$$

$$Q \rightarrow Q' \ \Rightarrow\ Q \mid Q'' \ \rightarrow\ Q' \mid Q'' \qquad [\textsc{Par}]$$

$$Q' \equiv Q' \text{ and } Q' \rightarrow Q'' \text{ and } Q'' \equiv Q \ \Rightarrow\ Q \rightarrow Q''' \qquad [\textsc{Str}]$$

Fig. 8. Reduction

of the identifier n (a name or a channel) in Q, processes $\mathsf{accept}\ a(x).Q$, $\mathsf{request}\ a(x).Q$ and $v?(x).Q$, which bind variable x in Q, and $(\mathsf{rec}\ X(\tilde{x}).Q)[\tilde{v}]$, which binds both the occurrences of the variables in $\tilde{x}$ and the occurrences of process variable X in process Q. The definition of *bound* and *free* names and channels in a process Q is standard, denoted by $\mathrm{fv}(Q)$ and $\mathrm{fc}(Q)$ respectively, and so is the capture-free *substitution* of a variable x with value v in a process Q, denoted by $Q[v/x]$, as well as the substitution of the part X by $\mathsf{rec}\ X(\tilde{x}).Q$ in a process Q', denoted by $Q'[\mathsf{rec}\ X(\tilde{x}).Q/X]$. We implicitly assume that in all mathematical contexts all bound identifiers are pairwise disjoint and disjoint from the free identifiers.

3.2. *Operational Semantics*

The operational semantics relies on structural congruence for the syntactic rearrangement of processes, preparing these for the application of the rules in the reduction relation. *Structural congruence* is the smallest relation including the rules in Figure 7. The first three rules establish the commutativity and associativity of the relation, and determine that the terminated process is its neutral element. The fourth rule is scope extrusion, and allows the scope of identifier n to encompass process Q' as well. Notice that our assumption on bound and free identifiers makes sure n is not free in Q, so as to avoid the capture of a free name. The next rule says that the binding order in not important, and the last allows for the garbage collection of unused channel or name binders.

The reduction relation $\rightarrow$ is defined in Figure 8. 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. The scope of the new channel encompasses the two processes only, to guarantee absence of interferences from other processes running in parallel. Once such a communication channel is established the interaction progresses either by value exchanging or by selection/branching. In the former case, rule [Com], when a sending process $k![v].Q$ meets a receiving process $k?(x).Q'$, value v is passed from the send party to the receive

party, replacing variable x in Q'. In the latter case, rule [Case], a process $k \lhd l_j.Q$ selects the branch labeled with l_j in the menu of process $k \rhd \{l_i\colon Q_i\}_{i\in I}$ resulting in process Q_j.

Rules [IfT] and [IfF] are conventional. The rule for recursive processes, [Rec], replaces a process $(\mathsf{rec}\ X(\tilde{x}).P)[\tilde{v}]$ by its body P with the appropriate substitutions performed: values $\tilde{v}$ for variables $\tilde{x}$ and part X for part $\mathsf{rec}\ X(\tilde{x}).P$. The two subsequent rules in the figure—[Scop] and [Par]—allow reduction to happen underneath scope restriction and parallel composition, respectively. The last rule incorporates structural congruence $\equiv$ in reduction.

Proceeding with the FTP example, the parallel composition of the FTP *Server* and the *Client* reduces in two steps to the process

$$(\nu bk)(k![\text{“name”}]... \mid \mathsf{request}\ b(y).y![k].\mathit{Ftpd}[b] \mid \Pi_n \mathit{FtpThread}[b])$$

using the recursion expansion rule for the demon followed by the session initiation rule. A new runtime channel k is created to conduct the session established between the client and the FTP server; such a channel replaces variable z in the client and variable x in the demon. Next one of the threads accepts the demon request, yielding process

$$(\nu bkk')(k![\text{“name”}]... \mid k'![k].\mathit{Ftpd}[b] \mid k'?(x).x?(\mathit{username})... \mid \Pi_{n-1} \mathit{FtpThread}[b])$$

where a new session is established on the newly created channel k', between the demon and one the threads. The server side of the session k is then transferred from the demon to the thread, so that the client now talks directly to the FTP thread via channel k.

$$(\nu bkk')(k![\text{“name”}]... \mid \mathit{Ftpd}[b] \mid k?(\mathit{username})... \mid \Pi_{n-1} \mathit{FtpThread}[b])$$

3.3. *Errors and Main Result*

Reduction may go wrong for a number of reasons. Traditional problems include non-boolean values in a conditional, as in $\mathsf{if}\ a\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$, and arity mismatch in process calls, as in $(\mathsf{rec}\ X(xy).Q)[\mathsf{true}]$. 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, where one tries to send a value, whereas the other expects a label to branch upon, as in $k![\mathsf{true}] \mid k \rhd \{l_i\colon Q_i\}_{i\in I}$, or when both partners try to output, as in $k![\mathsf{true}] \mid k \lhd l.Q$, or even when three partners try to read/write on the same channel, as for example in $k![\mathsf{true}] \mid k \lhd l.Q \mid k?(x).Q'$. The formal definition of what we mean by an error process is below, where for simplicity we have omitted the “traditional” problems.

Definition 3.1 (Error Process). A *k-prefixed process* (k-process for short) is a process prefixed by channel k; that is: $k![v].Q$, $k?(x).Q$, $k \lhd l.Q$, 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 $(k![v].Q \mid k?(x).Q')$ or $(k \lhd l.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I})$. Then Q is an *error* if

$$Q \equiv (\nu \tilde{n})(Q' \mid Q'')$$

where Q' is, for some k, the parallel composition of two k-processes that do not form a k-redex.[†]

Let $\rightarrow^*$ denote the reflexive and transitive closure of the reduction relation. We are now in a position to state the main result of the paper.

Theorem 3.2 (Type Safety). If $\Gamma \vdash P \triangleright \Delta$ and $P \rightarrow^* Q$, then Q is not an error.

Proof. See page 22. □

Notice that the type system in Figures 4 and 5 is meant to type programs 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 alone, both in linear environments and in processes, does not satisfy subject reduction. Take a process P ready to initiate a simple session:

$$\mathsf{request}\ a(x).x![\mathsf{true}].\mathbf{0} \mid \mathsf{accept}\ a(y).y?(z).\mathbf{0}$$

We have $a\colon \langle ?[\mathsf{bool}].\mathsf{end}\rangle \vdash P \triangleright \emptyset$. But P reduces to

$$(\nu k)(k![\mathsf{true}].\mathbf{0} \mid k?(z).\mathbf{0})$$

whose parallel composition is not typable, for both operands will have k on the channel environment, making the disjoint union not defined in rule [Conc].

4. The Polarity Runtime Language

This section introduces a polarity-based runtime system for the programmer's syntax. In particular it describes the runtime language, its typing system and operational semantics, and the new notion of error. It concludes by proving the subject reduction and the type safety properties for the new language.

4.1. *Syntax, Semantics, Typing System, and Errors*

The syntax for the polarized runtime language, in Figure 9, is obtained from that in Figure 1 by adding polarized channels. To avoid confusion, we indicate polarized runtime processes with letter R. Polarized channels k^+ and k^- represent the two ends of a single channel k of the base runtime language (defined in Figure 6). We call each of these channel-polarity pairs a *channel-end* or channel for short, when confusion may not arise. The *dual* of polarity p, denoted by $\overline{p}$, is $-$ when p is $+$ and vice-versa.

The polarized language differs from the simple runtime language in Section 3 in that polarized channels replace simple channels as values, so that $k![\mathsf{true}].\mathbf{0}$ is no longer a process, but $k^-![\mathsf{true}].\mathbf{0}$ is. The ν-binders however remains unchanged, as for example in $(\nu k)(k^-![\mathsf{true}].\mathbf{0})$. Given that we did not change the bindings and that polarized channels

[†] This notion of error encompasses that of (HVK98), that included the case of three or more k-processes. In fact, if Q_1, Q_2, Q_3 are three k-processes, than obviously two (at least) of them cannot form a k-redex.

New syntactic forms

$$v ::= x \mid \mathsf{true} \mid \mathsf{false} \mid k^p \qquad \text{values}$$
$$p ::= + \mid - \qquad \text{polarity}$$

New reduction rules

$$\mathsf{accept}\ a(x).R \mid \mathsf{request}\ a(y).R' \ \rightarrow\ (\nu k)(R[k^+/x] \mid R'[k^-/y]) \qquad [\textsc{LinkP}]$$
$$k^p![v].R \mid\ k^{\overline{p}}?(y).R' \ \rightarrow\ R \mid R'[v/y] \qquad [\textsc{ComP}]$$
$$k^p \lhd l_j.R \mid k^{\overline{p}} \rhd \{l_i\colon R_i\}_{i\in I} \ \rightarrow\ R \mid\ R_j \qquad [\textsc{CaseP}]$$

New typing rules

$$\frac{\Gamma \vdash R \triangleright \Delta, k^+ : T, k^- : \overline{T}}{\Gamma \vdash (\nu k)R \triangleright \Delta} \qquad [\textsc{ResP}]$$

Fig. 9. The polarity language.

are indeed (channel-polarity) pairs, the notion of free channels in a process remain the same, for example $\mathrm{fc}((\nu k)(k^-![k'^+].\mathbf{0})) = \{k'\}$.

At this point we have three languages: one *programming* language and two *runtime* languages (the *base* and the *polarized* language) generated by the grammars in Figures 1, 6, and 9, respectively. The programming language is a sub language of both runtime languages. The runtime languages add (νk) bindings; they both add channels as values, either simple k or polarized k^p.

The reduction relation for the new language is given by the rules [LinkP], [ComP], [CaseP] in Figure 9; the remaining rules are taken from Figure 8. The new [LinkP] rule creates, for two processes $\mathsf{accept}\ a(x).R$ and $\mathsf{request}\ a(y).R'$ ready to engage in a session, one channel k and distributes its $+$ polarity to one of the partners and the $-$ polarity to the other. As in the base language, once channels are created interaction progresses either by value exchanging or by selection/branching, only that this time each partner uses its channel-end. For example, rule [ComP] performs value passing from the sending party $k^p![v].R$ to the receiving partner $k^{\overline{p}}?(y).R'$. We write $R \rightarrow_{\mathsf{p}} R'$ when $R \rightarrow R'$ can be deduced by the rules of polarity runtime reduction.

In the case of our running example, the parallel composition of the *Server* and the *Client* reduces in two steps to

$$(\nu bk)(k^-![\text{“name”}]... \mid \mathsf{request}\ b(y).y![k^+].\mathit{Ftpd}[b] \mid \Pi_n \mathit{FtpThread}[b])$$

using the recursion expansion rule for demon followed by the session initiation rule. The client (the request side) keeps the negative polarity, while the server (the accept) uses the positive polarity of a same channel k.

The types for the polarized runtime language are those of the base language, described of Figure 2. Channel environments Δ now associate polarized channels k^p to types T. The type system introduces a single new rule, [ResP] in Figure 9, that incorporates in the linear typing environment Δ two entries, one for k^+, the other for k^-, of dual types. We write $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ whenever the sequent $\Gamma \vdash R \triangleright \Delta$ is provable in the system in Figure 9.

Recalling the example at the end of Section 3, process P now reduces to

$$(\nu k)(k^+![\mathsf{true}].\mathbf{0} \mid k^-?(z).\mathbf{0})$$

which is still typable since there are two *disjoint* typing environments typing processes $k^+![\mathsf{true}].\mathbf{0}$ and $k^-?(z).\mathbf{0}$, namely, $k^+ : ![\mathsf{bool}].\mathsf{end}$ and $k^- : ?[\mathsf{bool}].\mathsf{end}$. The parallel composition of the two processes produces typing $k^+ : ![\mathsf{bool}].\mathsf{end}, k^- : ?[\mathsf{bool}].\mathsf{end}$, and the new restriction rule [ResP] produces the empty typing environment, since the types for k^+ and k^- are dual.

The notion of error processes in the base language (Definition 3.1) is now re-casted into the polarized language.

Definition 4.1 (Runtime error process). A *k-process* is a process prefixed by a polarized channel k^p; that is: $k^p![v].Q$, $k^p?(x).Q$, $k^p \lhd l.Q$, or $k^p \rhd \{l_i : Q_i\}_{i\in I}$. A *k-redex* is the parallel composition of two polarized k-processes of opposite polarities, either of form $(k^p![v].Q \mid k^{\overline{p}}?(x).Q')$ or $(k^p \lhd l.Q \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i\in I})$. Then R is, as before, an *error* if

$$R \equiv (\nu \tilde{n})(R' \mid R'')$$

where R' is, for some k, the parallel composition of two k-processes that do not form a k-redex.

4.2. *Subject Reduction*

Subject reduction and type-safety hold only for balanced environments only (GH05; YV07).

Definition 4.2 (Balanced environment). We say that Δ is balanced if $\{k^+, k^-\} \subseteq \mathrm{dom}(\Delta)$ implies $\Delta(k^+) = \overline{\Delta(k^-)}$.

Lemma 4.3 (Weakening Lemma). Let $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$.

1 $\Gamma, x : S \vdash_{\mathsf{p}} R \triangleright \Delta$.
2 $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, u : \mathsf{end}$.
3 $\Gamma, X : \tilde{\alpha} \vdash_{\mathsf{p}} R \triangleright \Delta$.

Proof. A simple induction on the derivation tree of each sequent. As an example, we prove item 2. We underline that u not in $\mathrm{dom}(\Delta)$ by hypothesis.

[Inact] We have that $\Gamma \vdash \mathbf{0} \triangleright \vec{u} : \vec{\mathsf{end}}$. Clearly $\Gamma \vdash \mathbf{0} \triangleright \vec{u} : \vec{\mathsf{end}}, u : \mathsf{end}$.

[Conc] We have that $\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta$ has been inferred from $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1$ and $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$ with $\Delta = \Delta_1, \Delta_2$. By induction hypothesis we have $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1, u : \mathsf{end}$. We apply the [Conc] rule to above sequent and to $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$ to infer $\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta_1, \Delta_2, u : \mathsf{end}$ as needed.

[Cat] We have that $\Gamma \vdash_{\mathsf{p}} w?(x).R \triangleright \Delta, w : ?[T].U$ has been inferred from $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, w : U, x : T$. We assume that $x \neq u$ (possibly by alpha-renaming x in R). Therefore the induction hypothesis is $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, w : U, x : T, u : \mathsf{end}$. We apply the [Cat] rule and infer $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, w : ?[T].U, u : \mathsf{end}$.

□

Lemma 4.4 (Strengthening Lemma). Let $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$.

1 If $x \notin \mathrm{fv}(R)$, then $\Gamma \setminus x \vdash_{\mathsf{p}} R \triangleright \Delta$.
2 If $k \notin \mathrm{fc}(R)$, then $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta \setminus k^+ \setminus k^-$ and $\Delta(k^p) = \mathsf{end}$ when $k^p \in \mathrm{dom}(\Delta)$.
3 If $X \notin \mathrm{fpv}(R)$, then $\Gamma \setminus X \vdash_{\mathsf{p}} R \triangleright \Delta$.

Proof. A simple induction on the typing derivations. We draw some cases from clause 2.

[RESP] Let $\Gamma \vdash_{\mathsf{p}} (\nu k_1)R \triangleright \Delta$ and $k \notin \mathrm{fc}((\nu k_1)R)$. We assume $k \neq k_1$ (possibly by alpha-renaming k_1 in R) and use the induction hypothesis $\Gamma \vdash_{\mathsf{p}} R \triangleright (\Delta, k_1^+ : T, k_1^- : \overline{T}) \setminus k^+ \setminus k^-$, that is $\Gamma \vdash_{\mathsf{p}} R \triangleright (\Delta \setminus k^+ \setminus k^-), k_1^+ : T, k_1^- : \overline{T}$. We apply the [RESP] rule and infer $\Gamma \vdash_{\mathsf{p}} (\nu k_1)R \triangleright \Delta \setminus k^+ \setminus k^-$, as required. From the induction hypothesis we also know that $\Delta(k_1^p) = \mathsf{end}$ when $k^p \in \mathrm{dom}(\Delta, k_1^+ : T, k_1^- : \overline{T})$, hence when $k^p \in \mathrm{dom}(\Delta)$ since $k \neq k_1$.

[CONC] We have that $\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta$ and $k \notin \mathrm{fc}(R_1 \mid R_2)$. Let $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1$ and $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$ with $\Delta = \Delta_1, \Delta_2$. By the induction hypothesis we have $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1 \setminus k^+ \setminus k^-$ and $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2 \setminus k^+ \setminus k^-$. We apply the [CONC] rule and obtain $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright (\Delta_1 \setminus k^+ \setminus k^-), (\Delta_2 \setminus k^+ \setminus k^-)$ and we are done, since it is easy to see that $\Delta \setminus k^+ \setminus k^- = (\Delta_1 \setminus k^+ \setminus k^-), (\Delta_2 \setminus k^+ \setminus k^-)$. From the induction hypothesis we know that $\Delta_i(k_1^p) = \mathsf{end}$ when $k^p \in \mathrm{dom}(\Delta_i, k_1^p : T, k_1^- : \overline{T})$, hence when $k^p \in \mathrm{dom}(\Delta_i)$, hence when $k^p \in \mathrm{dom}(\Delta)$.

□

Lemma 4.5 (Typing Congruence). If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ and $R \equiv R'$, then $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta$.

Proof. By case analysis on the rules generating relation $\equiv$. The interesting cases are detailed below.

Case $R \mid \mathbf{0} \equiv R$. We show that if $\Gamma \vdash_{\mathsf{p}} R \mid \mathbf{0} \triangleright \Delta$ then $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$. Let the judgement be inferred from $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta_1$ and $\Gamma \vdash_{\mathsf{p}} \mathbf{0} \triangleright \vec{u} : \vec{\mathsf{end}}$, with $\Delta = \Delta_1, \vec{u} : \vec{\mathsf{end}}$. Using Weakening (Lemma 4.3) we obtain $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ as needed. For the other direction we start with derivation $\Gamma \vdash_{\mathsf{p}} \mathbf{0} \triangleright \emptyset$ and then apply rule [CONC].

Case $(\nu k)(R_1 \mid R_2) \equiv (\nu k)R_1 \mid R_2$. We show that if $\Gamma \vdash_{\mathsf{p}} (\nu k)(R_1 \mid R_2) \triangleright \Delta$ then $\Gamma \vdash_{\mathsf{p}} (\nu k)R_1 \mid R_2 \triangleright \Delta$. Let the judgement be inferred from $\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta, k^+ : T, k^- : \overline{T}$. We have that $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1$ and $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$ with $\Delta, k^+ : T, k^- : \overline{T} = \Delta_1, \Delta_2$. First notice that k^+ and k^- can be both in either Δ_i or one in each.
When they are both in Δ_1 we conclude the case by applying [RESP] and [CONC].
When they are both in Δ_2 we apply Weakening to obtain $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1, k^+ : \mathsf{end}, k^- : \mathsf{end}$. By [RESP] we infer $\Gamma \vdash_{\mathsf{p}} (\nu k)R_1 \triangleright \Delta_1$. We apply Strengthening (Lemma 4.4) to R_2 to remove channels k^+ and k^- (since k is not in the free names of R_2), then we use [CONC] and we conclude.
When k^+ is in Δ_1 and k^- in Δ_2, since $k \notin \mathrm{fc}(R_2)$, we apply Strengthening to R_2 and remove k^-. Strengthening also tells us that $\overline{T} = \mathsf{end}$. Then we weaken Δ_1 by adding $k^- : \mathsf{end}$. We use [SCOP] to obtain $\Gamma \vdash_{\mathsf{p}} (\nu k)R_1 \triangleright \Delta$ and [CONC] to conclude.
For the other direction, let the sequent $\Gamma \vdash_{\mathsf{p}} (\nu k)R_1 \mid R_2 \triangleright \Delta$ be inferred from $\Gamma \vdash_{\mathsf{p}}$

$R_1 \triangleright \Delta_1, k^+ : T, k^- : \overline{T}$ and $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$, with $\Delta = \Delta_1, \Delta_2$. We assume $\{k^+, k^-\} \cap \mathrm{dom}(\Delta_2) = \emptyset$, possibly by alpha-renaming k in R_1. We apply [Conc] followed by [ResP] to conclude.

Case $(\nu n)\mathbf{0} \equiv \mathbf{0}$. We must consider the case when n is a channel and when is a name. The most interesting case is when n is a channel k and we start from $\Gamma \vdash \mathbf{0} \triangleright \Delta$. We apply Weakening to introduce two entries $k^+ : \mathsf{end}, k^- : \mathsf{end}$ if not already in Δ, then apply rule [ResP], and finally Weakening again to reintroduce the entries k^p in Δ.

□

Lemma 4.6. Assume $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$. If $x \notin \mathrm{dom}(\Delta)$ then $x \notin \mathrm{fc}(R)$.

Proof. A simple induction on the derivation tree for each sequent. □

Lemma 4.7 (Substitution Lemma).

1 If $\Gamma \vdash_{\mathsf{p}} v : S$ and $\Gamma, x : S \vdash_{\mathsf{p}} R \triangleright \Delta$ then $\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta$.
2 If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, x : T$ then $\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, v : T$.
3 If $\Gamma, X : \vec{S}\vec{T} \vdash_{\mathsf{p}} R \triangleright \Delta$ and $\Gamma, X : \tilde{S}\tilde{T}, \tilde{x} : \tilde{S} \vdash_{\mathsf{p}} R' \triangleright \tilde{y} : \tilde{T}$ then $\Gamma \vdash_{\mathsf{p}} R[\mathsf{rec}\ X(\vec{x}\vec{y}).R'/X] \triangleright \Delta$.

Proof. We proceed by induction on the typing derivation for the process. We first prove (1). Assume $\Gamma \vdash_{\mathsf{p}} v : S$ and $\Gamma, x : S \vdash_{\mathsf{p}} R \triangleright \Delta$. From [True],[Name] we infer $\Gamma = \Gamma^*, v : S$.

[Acc] Let $\Gamma, x : S \vdash_{\mathsf{p}} \mathsf{accept}\ u(y).R \triangleright \Delta$. be inferred from

$$\Gamma, x : S \vdash_{\mathsf{p}} u : \langle T \rangle \qquad \Gamma, x : S \vdash_{\mathsf{p}} R \triangleright \Delta, y : T$$

In case $u = x$ we infer $\Gamma^*, v : S \vdash_{\mathsf{p}} u[v/x] : \langle T \rangle$ otherwise by strenghtening we have $\Gamma \vdash_{\mathsf{p}} u[v/x] : \langle T \rangle$. The induction hypothesis is

$$\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, y : T$$

We apply [Acc] and infer

$$\Gamma \vdash_{\mathsf{p}} (\mathsf{accept}\ u(y).R)[v/x] \triangleright \Delta, y : T$$

[Send] Let $\Gamma, x : S \vdash_{\mathsf{p}} w![u].R \triangleright \Delta, w : ![S^*].T$ be inferred from

$$\Gamma, x : S \vdash_{\mathsf{p}} u : S^* \qquad \Gamma, x : S \vdash_{\mathsf{p}} R \triangleright \Delta, w : T$$

If $u = x$ then we infer $\Gamma^*, v : S \vdash_{\mathsf{p}} u[v/x] : S^*$ otherwise by strenghtening $\Gamma \vdash_{\mathsf{p}} u[v/x] : S^*$.

Notice $w \neq x$. The induction hypothesis is

$$\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, w : T$$

We apply [Send] and infer

$$\Gamma \vdash_{\mathsf{p}} (w![u].R)[v/x] \triangleright \Delta, w : ![S^*].T$$

[Conc] Let $\Gamma, x : T \vdash_{\mathsf{p}} P \mid P' \triangleright \Delta, \Delta'$ be inferred from

$$\Gamma, x : T \vdash_{\mathsf{p}} P \triangleright \Delta \quad \Gamma, x : T \vdash_{\mathsf{p}} P' \triangleright \Delta'$$

The induction hypotheses are

$$\Gamma \vdash_{\mathsf{p}} P[v/x] \triangleright \Delta \quad \Gamma \vdash_{\mathsf{p}} P'[v/x] \triangleright \Delta'$$

We apply [Conc] and by grouping the substitution we obtain

$$\Gamma \vdash_{\mathsf{p}} (P \mid P')[v/x] \triangleright \Delta, \Delta'$$

as requested.

[Inact] We have $\Gamma, x : T \vdash_{\mathsf{p}} \mathbf{0} \triangleright \vec{u}\colon \vec{\mathsf{end}}$ and by strenghtening we obtain

$$\Gamma \vdash_{\mathsf{p}} \mathbf{0} \triangleright \vec{u}\colon \vec{\mathsf{end}}$$

Since the substitution $[v/x]$ has no effect on the inhert process we infer

$$\Gamma \vdash_{\mathsf{p}} \mathbf{0}[v/x] \triangleright \vec{u}\colon \vec{\mathsf{end}}$$

as requested.

Cases [Req],[Rcv],[Br],[Sel],[If] are analogous to [Acc]. The remaining cases are analogous to [Conc] and follow directly from the induction hypothesis.

Next we prove (2). Assume $v \notin \mathrm{dom}(\Delta)$.

[Thr] Let $\Gamma \vdash_{\mathsf{p}} w![u].R \triangleright \Delta, w\colon ![T^*].U, u\colon T^*$ be inferred from

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, w\colon U$$

If $x = w$ the induction hypothesis is

$$\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, v\colon T'$$

We apply [Thr] and infer

$$\Gamma \vdash_{\mathsf{p}} v![u].R[v/x] \triangleright \Delta, v\colon ![T^*].T', u : T^*$$

as requested.

Otherwise assume $x \neq w$. If $u = x$ by Lemma 4.6 we know that $x \notin \mathrm{fc}(R)$ Therefore the substitution $[v/x]$ has no effect on R and we could infer

$$\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, w\colon U$$

We apply [Thr] and infer

$$\Gamma \vdash_{\mathsf{p}} w![v].R[v/x] \triangleright \Delta, w\colon ![T^*].U, v\colon T^*$$

Finally when $u \neq x \neq w$ the result follows directly from the induction hypothesis.

[Cat] Let $\Gamma \vdash u?(y).R \triangleright \Delta, u\colon ?[T^*].U$ be inferred from

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, u\colon U, y\colon T^*$$

When $u \neq x$ the result follows from the induction hypothesis. Assume $u = x$; the induction hypothesis is

$$\Gamma \vdash_{\mathsf{p}} R[v/y] \triangleright \Delta, v\colon U, y\colon T^*$$

We apply [Cat] and infer

$$\Gamma \vdash v?(y).R[v/y] \triangleright \Delta, v\colon ?[T^*].U$$

as requested.

[Conc] Let $\Gamma \vdash R \mid R' \triangleright \Delta, \Delta'$ be inferred from

$$\Gamma \vdash R \triangleright \Delta \qquad \Gamma \vdash R' \triangleright \Delta'$$

Assume $\Delta = \Delta^*, x : T$. The induction hypothesis is

$$\Gamma \vdash R[v/x] \triangleright \Delta^*, v : T$$

Since $x \notin \mathrm{dom}(\Delta')$ by Lemma 4.6 we infer $x \notin \mathrm{fc}(R')$. Therefore the substitution $[v/x]$ has no effect on R'. We use [Conc] and by grouping the subtitution we obtain

$$\Gamma \vdash (R \mid R')[v/x] \triangleright \Delta^*, v : T, \Delta'$$

The case $\Delta' = \Delta^*, x : T$ is analogous.

[Inact] Assume $\Gamma \vdash \mathbf{0} \triangleright \vec{u} \colon \vec{\mathsf{end}}, x : \mathsf{end}$. We apply [Inact] and infer $\Gamma \vdash \mathbf{0}[v/x] \triangleright \vec{u} \colon \vec{\mathsf{end}}, v : \mathsf{end}$.

[Res] Let $\Gamma \vdash (\nu k)R \triangleright \Delta, x : T$ be inferred from

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, x : T, k^+ \colon U, k^- \colon \overline{U}$$

and assume $v \neq k$ (eventually by alpha-renaming k, if necessary). The induction hypothesis is

$$\Gamma \vdash_{\mathsf{p}} R[v/x] \triangleright \Delta, v : T, k^+ \colon U, k^- \colon \overline{U}$$

We apply [Res] and infer

$$\Gamma \vdash (\nu k)R \triangleright \Delta, v : T$$

Cases [Br],[Sel] are analogous to [Cat]. The remaining cases follow directly from the induction hypothesis.

To prove (3), we proceed by induction on the derivation for $\Gamma, X \colon \vec{S}\vec{T} \vdash_{\mathsf{p}} R \triangleright \Delta$. We prove the main cases; the remaining one are analogous.

[PVar] We have $\Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} X[\tilde{u}\tilde{v}] \triangleright \vec{w} \colon \vec{\mathsf{end}}, \tilde{v} \colon \tilde{T}$ inferred from $\Gamma \vdash_{\mathsf{p}} \tilde{u} \colon \tilde{S}$. This together with the hypothesis $\Gamma, X \colon \tilde{S}\tilde{T}, \tilde{x} \colon \tilde{S} \vdash_{\mathsf{p}} R' \triangleright \tilde{y} \colon \tilde{T}$ let us apply [Rec] and infer

$$\Gamma \vdash_{\mathsf{p}} (\mathsf{rec}\ X(\tilde{x}\tilde{y}).R')[\tilde{u}\tilde{v}] \triangleright \tilde{v} \colon \tilde{T}$$

By weakening we infer

$$\Gamma \vdash_{\mathsf{p}} (\mathsf{rec}\ X(\tilde{x}\tilde{y}).R')[\tilde{u}\tilde{v}] \triangleright \vec{w} \colon \vec{\mathsf{end}}, \tilde{v} \colon \tilde{T}$$

that is what we need since

$$\Gamma \vdash_{\mathsf{p}} X[\tilde{u}\tilde{v}][\mathsf{rec}\ X(\tilde{x}\tilde{y}).R'/X] \triangleright \vec{w} \colon \vec{\mathsf{end}}, \tilde{v} \colon \tilde{T}$$

[Conc] We have $\Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta_1, \Delta_2$ since

$$\Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1 \text{ and } \Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2$$

Assume $\Gamma, X \colon \tilde{S}\tilde{T}, \tilde{x} : \tilde{S} \vdash_{\mathsf{p}} R' \triangleright \tilde{y} : \tilde{T}$. By induction hypothesis both

$$\Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_1[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta_1 \text{ and } \Gamma, X \colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_2[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta_2$$

We apply [Conc] and infer

$$\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R_1[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \mid R_2[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta_1, \Delta_2$$

We group the substitution and infer

$$\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} (R_1 \mid R_2)[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta_1, \Delta_2$$

as requested.

[Thr] We have that $\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} u![v].R \triangleright \Delta, u\colon ![T].U, v\colon T$ is inferred from $\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R \triangleright \Delta, u\colon U$. Assume $\Gamma, X\colon \tilde{S}\tilde{T}, \tilde{x}\colon \tilde{S} \vdash_{\mathsf{p}} R' \triangleright \tilde{y}\colon \tilde{T}$.

By induction hypothesis, we infer

$$\Gamma \vdash_{\mathsf{p}} R[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta, u : U$$

We apply [Thr] and infer

$$\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} u![v].(R[\mathsf{rec}X(\vec{x}\vec{y}).R'/X]) \triangleright \Delta, u\colon ![T].U, v\colon T$$

that is equal to

$$\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} (u![v].R)[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta, u\colon ![T].U, v\colon T$$

as requested.

[ResP] We have $\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} (\nu k)R \triangleright \Delta$ inferred from

$$\Gamma, X\colon \tilde{S}\tilde{T} \vdash_{\mathsf{p}} R \triangleright \Delta, k^+\colon T, k^-\colon \overline{T}$$

Assume $\Gamma, X\colon \tilde{S}\tilde{T}, \tilde{x}\colon \tilde{S} \vdash_{\mathsf{p}} R' \triangleright \tilde{y}\colon \tilde{T}$ with $k \notin \mathrm{fc}(R')$. By induction hypothesis, we infer

$$\Gamma \vdash_{\mathsf{p}} R[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta, k^+\colon T, k^-\colon \overline{T}$$

We apply [ResP] and infer

$$\Gamma \vdash_{\mathsf{p}} (\nu k)R[\mathsf{rec}X(\vec{x}\vec{y}).R'/X] \triangleright \Delta$$

□

We are now in possession of all the ingredients we need prove subject reduction.

Theorem 4.8 (Subject Reduction for the Polarized Language). If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ with Δ balanced and $R \rightarrow_{\mathsf{p}} R'$, then $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$ with Δ' balanced and $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta')$.

Proof. By induction on the structure of the inference $R \rightarrow_{\mathsf{p}} R'$. The interesting cases are detailed below. [Str] is obtained from Typing Congruence (Lemma 4.5).

[LinkP] Let $\mathsf{accept}\ a(x).R_1 \mid \mathsf{request}\ a(y).R_2 \rightarrow_{\mathsf{p}} (\nu k)(R_1[k^+/x] \mid R_2[k^-/y])$. Assume

$$\Gamma \vdash_{\mathsf{p}} \mathsf{accept}\ a(x).R_1 \mid \mathsf{request}\ a(y).R_2 \triangleright\ \Delta_1, \Delta_2$$

is inferred from

$$\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1, x\colon T \text{ and } \Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2, y\colon \overline{T} \text{ and } \Gamma \vdash_{\mathsf{p}} a\colon \langle T \rangle \text{ and } \Delta_1, \Delta_2 \text{ balanced}$$

via rules [Conc], [Acc], and [Req]. We show that $\Gamma \vdash_{\mathsf{p}} (\nu k)(R_1[k^+/x] \mid R_2[k^-/y]) \triangleright \Delta_1, \Delta_2$.

The variable convention tells us that $k \notin \mathrm{fc}(R_1, R_2)$ and that k^+, k^- are not in Δ_1 nor in Δ_2. Then we apply the Substitution Lemma to obtain

$$\Gamma \vdash_{\mathsf{p}} R_1[k^+/x] \triangleright \Delta_1, k^+ : T \text{ and } \Gamma \vdash_{\mathsf{p}} R_2[k^-/y] \triangleright \Delta_2, k^- : \overline{T}$$

Finally we apply [Conc], [ResP] to conclude.

[ComP] We analyze first the session passing (throw/catch) case. Let $k^+![v].R_1 \mid k^-?(x).R_2 \rightarrow_{\mathsf{p}} R_1 \mid R_2[v/x]$, and assume that there is Δ balanced such that

$$\Gamma \vdash_{\mathsf{p}} k^+![v].R_1 \mid k^-?(x).R_2 \triangleright \Delta$$

From these hypotheses and rules [Conc], [Thr] and [Cat] we infer that there are types T, U such that

$$\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1, k^+ : U$$
$$\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2, k^- : \overline{U}, x : T$$
$$\Delta = (\Delta_1, k^+ : ![T].U, v : T), (\Delta_2, k^- : ?[T].\overline{U})$$

We show that

$$\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2[v/x] \triangleright \Delta' \text{ with } \Delta' = (\Delta_1, k^+ : U), (\Delta_2, k^- : \overline{U}, v : T)$$

which is what we need since Δ' is balanced and $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta)$.
We apply the second clause of the Substitution Lemma to $\Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2, k^- : \overline{U}, x : T$, followed by rule [Conc] to obtain the result.
In the case of shared value passing (send/receive) we follow the same pattern but use the first clause of the Substitution Lemma.

[CaseP] Let $k^+ \triangleleft l.R \mid k^- \triangleright \{l_i : R_i\}_{i \in I} \rightarrow_{\mathsf{p}} R \mid R_j$, $j \in I$. Assume that there is Δ balanced such that

$$\Gamma \vdash_{\mathsf{p}} k^+ \triangleleft l_j.R \mid k^{\overline{p}} \triangleright \{l_i : R_i\}_{i \in I} \triangleright \Delta$$

From these hypotheses and rules [Conc], [Br] and [Sel] we infer that

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta_1, k^+ : T_j$$
$$\Gamma \vdash_{\mathsf{p}} R_j \triangleright \Delta_2, k^- : \overline{T_i} \quad \forall i \in I$$
$$\Delta = (\Delta_1, k^+ : \oplus\{l_i : T_i\}_{i \in I}), (\Delta_2, k^- : \&\{l_i : \overline{T_i}\}_{i \in I})$$

We show that

$$\Gamma \vdash_{\mathsf{p}} R \mid R_j \triangleright \Delta' \text{ with } \Delta' = \Delta_1, k^+ : T_j, \Delta_2, k^- : \overline{T_j}$$

This closes the proof because Δ' is balanced and $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta)$. We apply [Conc] to conclude the case.

[Rec] Let $(\mathsf{rec}\ X(\tilde{x}\tilde{y}).P)[\tilde{u}\tilde{v}] \rightarrow P[\tilde{u}/\tilde{x}][\tilde{v}/\tilde{y}][\mathsf{rec}\ X(\tilde{x}).P/X]$. Assume that there are $\tilde{v} : \tilde{T}$ balanced and Γ such that

$$\Gamma \vdash_{\mathsf{p}} (\mathsf{rec} X(\tilde{x}\tilde{y}).P)[\tilde{u}\tilde{v}] \triangleright \tilde{v} : \tilde{T} \text{ and } \Gamma \vdash_{\mathsf{p}} \tilde{u} : \tilde{S}$$

We show that $\Gamma \vdash_{\mathsf{p}} P[\tilde{u}/\tilde{x}][\tilde{v}/\tilde{y}][\mathsf{rec} X(\tilde{z}).P/X] \triangleright \tilde{v} : \tilde{T}$. Apply in turn each of the three clauses in the Substitution Lemma to conclude the case.

[Scop] We analyse channel restriction; name restriction is easier. Let $(\nu k)R \to_{\mathsf{p}} (\nu k)R'$ be inferred from $R \to R'$. Assume that $\Gamma \vdash_{\mathsf{p}} (\nu k)R \triangleright \Delta$ with Δ balanced has been inferred via [ResP] from

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta, k^+ : T, k^- : \overline{T}$$

By induction hypothesis there is Δ' balanced with the domain of $\Delta, k^+ : T, k^- : \overline{T}$, such that $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$. Then Δ' must be of the form $\Delta'', k^+ : U, k^- : \overline{U}$. We apply rule [Scop] to obtain $\Gamma \vdash_{\mathsf{p}} (\nu k)R' \triangleright \Delta''$. It is easy to see that Δ'' is balanced and has the same domain as Δ.

[Par] Let $R_1 \mid R_2 \to_{\mathsf{p}} R' \mid R_2$ be inferred from $R_1 \to_{\mathsf{p}} R'$. We assume that $\Gamma \vdash_{\mathsf{p}} R_1 \mid R_2 \triangleright \Delta$ has been inferred from

$$\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1 \text{ and } \Gamma \vdash_{\mathsf{p}} R_2 \triangleright \Delta_2 \text{ with } \Delta = \Delta_1, \Delta_2$$

It is easy to see that Δ_1 is balanced. We can therefore apply the induction hypothesis to $R_1 \to_{\mathsf{p}} R'$ and infer that there is Δ' balanced such that $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta_1)$ and $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$. Then we know that Δ', Δ_2 is defined, balanced and has the same domain as Δ_1, Δ_2. We apply [Par] to conclude the case.

□

Corollary 4.9. If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ with Δ balanced and $R \to_{\mathsf{p}}^* R'$, then $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$ for some Δ' balanced.

Proof. By induction on $R \to_{\mathsf{p}}^n R'$. The case $n = 0$ is trivial. Otherwise assume $R \to_{\mathsf{p}}^n R_1 \to_{\mathsf{p}} R'$. The induction hypothesis is that $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1$ and Δ_1 balanced. We apply Theorem 4.8 to obtain that there is Δ' balanced such that $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$, thus concluding the proof. □

4.3. *Type Safety*

The rest of this section is dedicated to the proof of type safety for the polarized language.

Theorem 4.10. If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ and Δ is balanced, then R is not an error.

Proof. Assume that $\Gamma \vdash_{\mathsf{p}} (\nu \tilde{x}\tilde{k})(R_1 \mid R_2) \triangleright \Delta$ with Δ balanced. After multiple applications of typing rules [ResP] and [Res] we obtain

$$\Gamma, \tilde{x} : \tilde{S} \vdash_{\mathsf{p}} R_i : \Delta_i$$

with $\Delta_1, \Delta_2 = \Delta, \tilde{k^+} : \tilde{T}, \tilde{k^-} : \tilde{\overline{T}}$. Notice that each Δ_i is balanced. Assume that process R_1 is the parallel composition of two k-processes. It is easy to see that R_1 is not typable except when the two k-processes form a k-redex. □

Corollary 4.11 (Type Safety for the Polarized Language). Let $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ with Δ balanced. If $R \to_{\mathsf{p}}^* R'$, then R' is not an error.

Proof. By Corollary 4.9 we know that there is Δ' balanced such that $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta'$. We complete the proof with Theorem 4.10. □

5. Well-typed Programs Do Not go Wrong

At the end of Section 3 we have seen that the straightforward extension of the type system for programs does not satisfy the subject reduction property. This section presents a proof of the main result (Theorem 3.2), via an erase mapping from the polarized runtime language to the base runtime language, using operational and error correspondence results. The erase function maps a polarity runtime process R into a base runtime process Q by simply removing all polarities $+, -$ from the given process. The formal definition is by induction on the structure of process R, which we omit.

The proof of Theorem 3.2 is simple.

Proof. Let P be a program such that $\Gamma \vdash P \triangleright \Delta^{(1)}$, and let Q be a runtime process such that $P \rightarrow^* Q^{(2)}$. We need to show that Q is not an error. The proof proceeds as follows.

— We note that

$$\Gamma \vdash P \triangleright \Delta^{(1)} \implies \Gamma \vdash_{\mathsf{p}} P \triangleright \Delta^{(3)} \text{ and } \Delta \text{ balanced}^{(4)}$$

for the type systems for the two languages coincide on programs.

— We show an operational correspondence (Theorem 5.2) stating that

$$\Gamma \vdash P \triangleright \Delta \text{ and } P \rightarrow^* Q \implies \exists R \text{ such that } P \rightarrow_p^* R \text{ and } \mathrm{erase}(R) = Q;$$

to obtain, from (1) and (2), that $\exists R$ such that $P \rightarrow_p^* R^{(5)}$ and $\mathrm{erase}(R) = Q^{(6)}$.

— We apply (3), (4) and (5) and Subject Reduction for the polarity language (Corollary 4.9) to obtain that $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta'^{(7)}$ and Δ' balanced$^{(8)}$.

— We establish an error correspondence result (Theorem 5.3) stating that

$$\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta \text{ and } \Delta \text{ balanced} \implies \mathrm{erase}(R) \text{ not an error}$$

to obtain, from (7) and (8) that $\mathrm{erase}(R)$ is not an error$^{(9)}$.

— From (6) and (9) we obtain the sought result, namely that Q is not an error.

□

The rest of this section is dedicated of the proofs of the operational and the error correspondence, Lemmas 5.2 and 5.3.

5.1. *Operational Correspondence*

The next Lemma is the core of the proof of operational correspondence.

Lemma 5.1. If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ with Δ balanced and $\mathrm{erase}(R) \rightarrow Q$, then there is a polarized process R' such that $R \rightarrow_{\mathsf{p}} R'$ and $\mathrm{erase}(R') = Q$.

Proof. By induction on the length of the inference $\mathrm{erase}(R) \rightarrow Q$. We outline some cases below.

[Link] Let $\mathrm{erase}(R) = \mathsf{accept}\ a(x).\mathrm{erase}(R_1) \mid \mathsf{request}\ a(y).\mathrm{erase}(R_2)$ and $Q = (\nu k)(Q_1[k/x] \mid Q_2[k/y])$. From [LinkP] we infer

$$\mathsf{accept}\ a(x).R_1 \mid \mathsf{request}\ a(y).R_2 \rightarrow_{\mathsf{p}} (\nu k)R_1[k^+/x] \mid R_2[k^-/y]$$

From $\text{erase}((\nu k)R_1[k^+/x] \mid R_2[k^-/y]) = Q$ we conclude.

[Com] Let $\text{erase}(R) = k![\text{erase}(w)].\text{erase}(R_1) \mid k?(x).\text{erase}(R_2)$ and $Q \equiv \text{erase}(R_1) \mid \text{erase}(R_2)[\text{erase}(w)/x]$. From $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ we infer $\Gamma \vdash \text{erase}(k^p![w].R_1) \triangleright \Delta_1$ and $\Gamma \vdash \text{erase}(k^{p'}?(x).R_2) \triangleright \Delta_2$ with $\Delta = \Delta_1, \Delta_2$. Since Δ is balanced we know that $p' = \overline{p}$. From [ComP] we infer

$$k^p![w].R_1 \mid k^{\overline{p}}?(x).R_2) \rightarrow_{\mathsf{p}} R_1 \mid R_2[w/x]$$

From $\text{erase}(R_1 \mid R_2[w/x]) \equiv Q$ we conclude.

[Scop] Let $\text{erase}(R) = (\nu k)Q'$ and let $(\nu k)Q' \rightarrow Q$ be inferred from $Q' \rightarrow Q''$ with $Q = (\nu k)Q''$. Therefore $R = (\nu k)R'$ with $\text{erase}(R') = Q'$. From $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ we infer that $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta, k^+ : T, k^- : \overline{T}$. Since Δ is balanced, $\Delta, k^+ : T, k^- : \overline{T}$ is balanced. We may apply the induction hypothesis and infer that there is R'' such that $R' \rightarrow_{\mathsf{p}} R''$ and $\text{erase}(R'') \equiv Q''$. We apply [ResP] to obtain

$$(\nu k)R' \rightarrow_{\mathsf{p}} (\nu k)R''$$

From $\text{erase}((\nu k)R'') = (\nu k)Q'' = Q$ we conclude.

[Str] Assume $\text{erase}(R) \rightarrow Q$ is obtained from $\text{erase}(R) \equiv Q'$, $Q' \rightarrow Q''$ and $Q'' \equiv Q$. It's easy to find $R' \equiv_p R$ such that $Q' = \text{erase}(R')$. Let $\Gamma \vdash_{\mathsf{p}} \triangleright\Delta$ with Δ balanced. By typing congruence (Lemma 4.5) we infer $\Gamma \vdash_{\mathsf{p}} R' \triangleright \Delta$. We apply the induction hypothesis and obtain that there is R'' such that $R' \rightarrow_{\mathsf{p}} R''$ and $\text{erase}(R'') \equiv Q''$. From [Str] we infer that $R \rightarrow_{\mathsf{p}} R''$, and we are done, since $\text{erase}(R'') \equiv Q'' \equiv Q$, as requested.

□

We are now in a position to prove operational correspondence.

Theorem 5.2 (Operational Correspondence). Let P be a program such that $\Gamma \vdash P \triangleright \Delta$. If there is a runtime process Q such that $P \rightarrow^n Q$, then there is a polarized process R such that $P \rightarrow_{\mathsf{p}}^n R$ and $\text{erase}(R) = Q$.

Proof. By induction on the length n of reduction. When $n = 0$, we have that $Q = P$ and we are done. For the induction step, assume $P \rightarrow^k Q_1 \rightarrow Q$. We apply the induction hypothesis and infer that there is R_1 such that $P \rightarrow_{\mathsf{p}}^k R_1$ and $\text{erase}(R_1) = Q_1$. From $\Gamma \vdash P \triangleright \Delta$ we know that $\Gamma \vdash_{\mathsf{p}} P \triangleright \Delta$, since the two type systems coincide on the programmers' syntax. Typing environments in the programmer's syntax do not contain polarized channels, hence Δ is balanced. By subject reduction (Theorem 4.8) we know that there is Δ_1 balanced such that $\Gamma \vdash_{\mathsf{p}} R_1 \triangleright \Delta_1$. We apply Lemma 5.1 to conclude □

5.2. *Error Correspondence*

To close the proof of our main result, we need to show that the erase function maps typed polarized processes into error-free processes.

Theorem 5.3 (Error Correspondence). If $\Gamma \vdash_{\mathsf{p}} R \triangleright \Delta$ with Δ balanced, then $\text{erase}(R)$ is not an error.

Proof. Theorem 4.10 tells us that R is not an error, hence that, if $R \equiv (\nu\tilde{n})(R_1 \mid R_2)$ then it is not the case that R_1 is the parallel composition of two (polarized) k-processes that do not form a k-redex, by Definition 4.1. We know that $\text{erase}(R) \equiv (\nu\tilde{n})(\text{erase}(R_1) \mid \text{erase}(R_2))$. Then it is not the case that $\text{erase}(R_1)$ is the parallel composition of two (base) k-processes that do not form a k-redex, by Definition 3.1. Hence $\text{erase}(R)$ is not an error. □

6. Conclusion

Related work. An embryo of this work can be found in (GHVY09). In the introduction we present those works most closely related to our investigation; here we review additional work in adjoining areas. Refer to (DCd10) for an overview of sessions and session types, including language extensions, extensions to typing, embeddings into different programming paradigms and implementations.

From the original formulation in a variant of the pi calculus, the concept of session types is now used in different realms, including in functional languages (GV10; VGR06; Vas09a), the higher-order pi calculus (MY07; MY09), web services (CHY07), multicore programming (YVPH08), to describe components (VVR06), in object-oriented languages (BCDC$^+$08; CCDC$^+$09; CDCY07; DCDGY07; HYH08; GVR$^+$10; DCDMY09), in logics (BHY08; VLC09), in relation to intuitionistic linear logic (CP09), to mention a few. The concept of sessions describing the interaction between to partners was eventually extended to allow the description of interacting involving a larger number of partners (BC08; HYC08; CV09; YDBH10; MYH09; BCD$^+$08; BCD$^+$09).

Polarities, or more generally the ability to syntactically distinguish the ends of a same channel (GH05; YV07; Vas09b), are used in most works on binary sessions. They do become particularly relevant in the presence of buffered (or asynchronous) semantics, where each of the two partners involved in an interaction is supposed to have its own message buffer. To send a message is to write on the partner's buffer; to receive a message is to read from one's own buffer. A two-way communication channel is then transformed into a pair of buffers, allowing each partner to proceed at its own pace. The semantics (static and dynamic) then work with two distinct channel ends, usually denoted by k^+ and k^- for a single channel k. In the case of buffered semantics, the distinction between the two ends in a channel can be justified based solely on the underlying computational model.

Summary. We have presented a simple and intuitive operational semantics for a variant of the pi calculus with sessions where typable processes are guaranteed to be exempt from runtime errors. The type system, meant to type programs, cannot be directly used to type the runtime language. In the absence of a simple extension of the type system for the runtime language, precluding proving type safety via the traditional subject reduction property, we resorted to a different runtime language. This new language, based on polarities, enjoys subject reduction; the sought result is then obtained by proving suitable correspondences between the two languages. The exercise shows that polarities, now

widely used in binary session types, are justified solely by technical reasons, and need not be exposed to programmers neither in the operational semantics nor in the type system.

Acknowledgements. Giunti and Vasconcelos were partially supported by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004). Vasconcelos was partially supported by the Large-Scale Informatics Systems Laboratory, Portugal. Honda and Yoshida are partially supported by EPSRC GR/T03208, GR/T03215, EP/F002114 and EP/F003757. Vasconcelos, Yoshida and Konda were partically supported by the Treaty of Windsor Anglo-Portuguese Joint Research Programme B–4/08.

References

Eduardo Bonelli and Adriana B. Compagnoni. Multipoint session types for a distributed calculus. In *TGC*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.

Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.

Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and union types for object oriented programming. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 659–680. Springer, 2008.

Martin Berger, Kohei Honda, and Nobuko Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In *ICALP*, volume 5126 of *LNCS*, pages 99–111. Springer, 2008.

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

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous session types and progress for object-oriented languages. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17, 2007.

Luís Caires and Frank Pfenning. A concurrent interpretation of intuitionistic linear logic. submitted, 2009.

Luís Caires and Hugo Torres Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: an overview. In *WS-FM'09*, LNCS. Springer, 2010. To appear.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded session types for object-oriented languages. In *Proceedings of FMCO'07*, volume 4709 of *LNCS*, pages 207–245. Springer, 2007.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Inf. Comput.*, 207(5):595–641, 2009.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. In *Proceedings of ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

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

Marco Giunti, Kohei Honda, Vasco T. Vasconcelos, and Nobuko Yoshida. Session-based type discipline for pi calculus with matching. PLACES'09, 2009.

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

Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM Press, 2010.

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, 1998.

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

Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Proceedings of ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA*, volume 4583 of *Lecture Notes in Computer Science*. Springer, 2007.

Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332, 2009.

Vasco T. Vasconcelos. Session types for linear multithreaded functional programming. In *PPDP*, pages 1–6. ACM Press, 2009.

Vasco T. Vasconcelos. *SFM*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer, 2009.

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

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *APLAS*, volume 5904 of *LNCS*, pages 194–209. Springer, 2009.

Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4), 2006.

Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FoSSaCs*, LNCS, 2010.

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*, number 171(4) in ENTCS, pages 73–93, 2007.

Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. Session-based compilation framework for multicore programming. In *FMCO*, volume 5751 of *Lecture Notes in Computer Science*, pages 226–246. Springer, 2008.