On Progress for Structured Communications ⋆

Mariangiola Dezani-Ciancaglini[1], Ugo de' Liguoro[1], and Nobuko Yoshida[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computing, Imperial College London

Abstract. We propose a new typing system for the π-calculus with sessions, which ensures the progress property, i.e. once a session has been initiated, typable processes will never starve at session channels. In the current literature progress for session types has been guaranteed only in the case of nested sessions, disallowing more than two session channels interfered in a single thread. This was a severe restriction since many structured communications need combinations of sessions. We overcome this restriction by inferring the order of channel usage, but avoiding any tagging of channels and names, neither explicit nor inferred. The simplicity of the typing system essentially relies on the session typing discipline, where sequencing and branching of communications are already structured by types. The resulting typing enjoys a stronger progress property than that one in the literature: it assures that for each well-typed process P which contains an open session there is an irreducible process Q such that the parallel composition $P|Q$ is well-typed too and it always reduces, also in presence of interfered sessions.

1 Introduction

Structuring communication to ensure safe interaction of concurrent systems is a central issue in the theory and practice of concurrent and mobile computing. Communication has indeed evolved into a growing number of complex activities, including several kinds of transactions as well as the offer and fruition of services through a large gamma of systems and networks. In this scenario computation consists in exchanging messages between loosely coupled parties, whose number and identity might also change dynamically. A case in point is delegation of activities to third parties in a client/server interaction, which often occurs transparently to the client.

Existing programming languages and standards, while adding communication primitives and syntactical tools to rule interaction, still leave to the programmer much of the responsibility in guarantying that the sequence of messages is well structured and that e.g. the client of a service will complete all needed transactions without getting into some unwanted state. The lack of structuring principles is also a defect of theoretical calculi such as the π-calculus: the economy of its syntax and semantics is an advantage for the elegance of the theory, but a drawback when controlling and disciplining specific kinds of behaviour.

A solution proposed by [2, 9, 10, 12, 21] consists in adding primitives to creat *sessions* to the π-calculus. A session is an abstraction of a series of communications

⋆ Work partially supported by EPSRC GR/T04724, GR/T03208, GR/T03215, IST2005-015905 MOBIUS, FP6-2004-510996 Coordination Action TYPES, and MURST PRIN'05 project "Logical Foundations of Distributed Systems and Mobile Code".

through a private channel between two processes. It is created by a connection over a session channel (we call *shared*), that binds a channel name which, after connection, is substituted by a fresh private name (the *live channel*) in such a way that both privacy and duality are guaranteed, in the sense of the presence of input/output, branching/selection and delegation actions with the same live channel as subject (as it is checked by basic session type systems).

A central motivation for developing sessions and related type systems is to model safe hand-shake communications. In such a context privacy is not the unique desirable property of sessions, whereas compliance should be also guaranteed, namely that any session does not get stuck into some blocking state. To explain this safety issue, let us consider the following simple process with sessions, written in a π-calculus dialect that admits sequential composition (the semicolon):

$$P_1 \;=\; a(x).(x!\langle 3\rangle; x?(z).x!\langle Apple\rangle; P_1')$$

This is a server process that first accepts the session communication through a shared channel a, and then performs a series of communication via the live channel which will replace x: it first outputs an integer, second inputs an integer, then outputs a string, and continue as P_1'. This behaviour is abstracted in the type system of [12] as the session type $!\mathsf{int}.?\mathsf{int}.!\mathsf{string}$.

A client process intended to interact with the server above will have the following communication pattern:

$$Q_1 \;=\; \overline{a}(x).(x?(z).x!\langle 5+z\rangle; x?(z').Q_1')$$

This process requests the session communication through a and then performs the dual actions through x, typed by $?\mathsf{int}.!\mathsf{int}.?\mathsf{string}$. Once the session is established, and provided that only the two connected parties interact together, the communication over the live channel replacing x always *proceeds* at least up to the transmission of the string (and to the end of the session if x does not occur in P_1' nor in Q_1'), since their communication patterns are dual and private.

The main limitation of the approach is that two parties are assumed to interact in one session, and that these should not overlap. On the contrary in the case of e.g. Web Services communications [22], we need to establish more than one session between two or even multiple peers. In such a case, the safety is easily destroyed by the interleaving of two or more sessions. The simplest example is as follows:

$$P_2 = a(x).b(y).(x!\langle 3\rangle; x?(z).y!\langle Apple\rangle; P_2') \qquad Q_2 = \overline{a}(x).\overline{b}(y).(y?(z').x?(z'').x!\langle 5\rangle; Q_2')$$

where the live channels replacing x and y create a circular dependency, causing deadlock. However in the session type systems from the literature, the latter processes are typable since the two sessions, one for x and the other for y, are correctly structured if taken in isolation. Thus progress of communications on live channels cannot be guaranteed when two or more sessions are mixed.

In the present work, we enhance existing session type systems to check progress with respect to live channels belonging to several sessions, while keeping the full session constructions, such as branching/selection, delegation and replication. The calculus is equipped with the construct for *sequencing* by which complex synchronisation behaviours such as joining and forking processes can be modelled. In spite of this

extension, we show that a great simplification w.r.t. existing type systems for partial deadlock-freedom is achieved by relativising progress to session structured processes, avoiding any tagging of channels and names, neither explicit nor inferred. Our type system enjoys a progress property tailored to the soundness of session execution: for each well-typed process P which contains live channels there is an irreducible process Q such that the parallel composition $P|Q$ is well-typed too and it always reduces. The main technical difficulties for progress come from the two central features of the π-calculus: one is name hiding and passing, which can stop communications forever, and the other is process replication, which can destroy the bilinearity of communications.

Related work The present paper moved from the desire to remove the limitations arising from strictly nested sessions in [6, 8], where a similar progress property has been established in the case of an object-oriented, class-based language with communication primitives for sessions and with concurrency disciplined by the use of spawning commands. That result has been obtained under the condition that overlapping sessions can only be nested and that the inner sessions have been ended before the outer ones may proceed. Such a restriction is abandoned here; moreover we leave aside any particular paradigm of programming languages, and consider an extension of the full π-calculus with the session primitives of [12].

A tight relation exists with work by Kobayashi and his colleagues on partial deadlock-freedom. We were inspired by [15–18, 23] in considering the relation between channel names induced by their use. However there are both technical and conceptual differences.

First we do not decorate types by multiplicities, namely we do not record levels of capabilities/obligations. Usages e.g. in [18], as well as "types" in the general framework of [14], are far more concrete behavioral descriptions than session types; hence the usages make sense as internal machinery of an automatic testing procedure, not as interfaces or abstract protocols for the user, we are looking for.

Second the structure of session types allows us to get a significant analysis without any form of tagging (neither by the user, nor by the typing system) and by means of a syntax directed type system, where the number of rules only depends on the richness of the language syntax. This is coherent with the aim of using session primitives and session types directly as the basis for programming language design, rather than as a tool to perform some form of static analysis. We leave for a future work to analyse relationships with the encodings of session types into functional and process linear typing systems [11, 19].

Paper structure Section 2 describes the syntax and the reduction rules of our calculus, and Section 3 discusses the type system. The features of well-typed processes are the subject of Section 4. The full definitions and proofs can be found at `http://www.di.unito.it/~dezani/dly.pdf`.

2 A Calculus for Structured Communications

2.1 Process Syntax

The π-calculus with sessions we consider is an extension of the calculus studied in [12], by means of *sequencing*, which allows to get forks and joins of processes [1]. The syntax is reported in Table 1.

For channels we use names and variables, the latter in place of bound names in accept/request and receive guarded processes. We further distinguish among two sorts of channel names: shared and live. *Shared channels* (called simply "names" in [12]), ranged over by $a, b, \ldots$ are used to open sessions, so that they can be either public or private; *live channels* (the "channels" of [12]), written as $k^p, k_1^q, \ldots$ are instead used only within open sessions, as it becomes clear in the definition of the operational semantics, so that their intended use (enforced by the reduction relation and the type system) is within the scope of the ν operator. The *polarity* $p \in \{+,-\}$ in apices of k^p represents the two end points created by the session initialisation. This notion is originally introduced in [10] to assure subject reduction (see [24] for the detailed discussion).

We write $a(x).P$ and $\overline{a}(x).P$ for the *accept* and *request* primitives of [12]. Instead of the recursive agents, we use *permanent accept*, written $\star a(x).P$, and for shared channels only, to model a server providing for a service to an unbounded number of clients. In case of $a(x).P$, $\star a(x).P$ and $\overline{a}(x).P$ the identifier a represents the public interaction point over which a session may commence. We say that a is the *subject* of the (permanent) accept/request process. The bound variable x represents the actual channel over which the session communications will take place, to be replaced by a live channel when the session has been opened and the connection established.

Constants and expressions of ground types (booleans and integers) are also added to model data, which are sent and received by means of the prefixes $\kappa!\langle e\rangle$ and $\kappa?(x).P$. We write $\kappa \triangleleft l.P$ for *selection*, which chooses an available branch, and $\kappa \triangleright \{l_1 : P_1 [\!] \ldots [\!] l_n : P_n\}$ for *branching*, which offers alternative interaction patterns; these are the same as in

(Shared Channels)

c	::= x, y, z	variable
	$\mid a, b$	name

(Live Channels)

κ	::= x, y, z	variable
	$\mid k^p$	polarised name

(Values)

v	::= a	shared channel name
	$\mid$ `true`, `false`	boolean
	$\mid n$	integer

(Expressions)

e	::= v	value
	$\mid x, y, z$	variable
	$\mid e + e$	sum
	$\mid$ `not`(e)	not
	$\mid \ldots$	

(Processes)

P	::= $\mathbf{0}$	inaction
	$\mid T$	prefixed process
	$\mid P \,;\, Q$	sequencing
	$\mid P \mid Q$	parallel
	$\mid (\nu a)P$	shared channel hiding
	$\mid (\nu k)P$	live channel hiding

(Prefixed processes)

T	::= $c(x).P$	accept
	$\mid \star c(x).P$	permanent accept
	$\mid \overline{c}(x).P$	request
	$\mid \kappa!\langle e\rangle$	data send
	$\mid \kappa?(x).P$	data receive
	$\mid \kappa \triangleleft l.P$	selection
	$\mid \kappa \triangleright \{l_1 : P_1 [\!] \ldots [\!] l_n : P_n\}$	branching
	$\mid \kappa!\langle\!\langle \kappa' \rangle\!\rangle$	session send
	$\mid \kappa?(\!(x)\!).P$	session receive
	$\mid$ `if` e `then` P `else` Q	if-then-else

Table 1. Syntax

[12].

We use $\kappa!\langle\!\langle\kappa'\rangle\!\rangle$ (session send) and $\kappa?(\!(x)\!).P$ (session receive) for throw and catch primitives of [12] respectively. These are called *higher-order session communication primitives* since live channel κ' is passed via live channel κ. This mechanism enables to represent complex but safe delegations without interference by any third party.

In data and session sending, data and session receive, branching and selection, we call the channel κ the *subject* of the prefixed process.

The essential difference with the calculus in [12] is the adding of *sequencing*, written $P\,;Q$, meaning that P is executed before Q. This syntax allows for complex forms of synchronisation as P can include any parallel composition of arbitrary processes.

The precedence of the operators building processes is (from the strongest) "$\lhd, \rhd, \{\}$", ".", ";" and "$\mid$ ". Moreover we convene that "." associates to the right. For example, $\kappa \lhd l.\kappa?(x).P; Q \mid R$ means $((\kappa \lhd l.(\kappa?(x).P)); Q) \mid R$. We often omit $\mathbf{0}$ and write $(\nu ab)(P)$ for $(\nu a)((\nu b)(P))$, etc. The bindings for channels and variables are standard and we write $\mathrm{fn}(P)$, $\mathrm{fv}(P)$ and $\mathrm{bn}(P)$ for free channels, free variables and bound channels respectively.

We say that the following pairs of prefixed processes are *dual*: $\{a(x).P,\ \overline{a}(x).Q\}$, $\{\star a(x).P,\ \overline{a}(x).P\}$, $\{k^p!\langle e\rangle,\ k^{\overline{p}}?(x).P\}$, $\{k^p \lhd l_i.P,\ k^{\overline{p}} \rhd \{l_1 : Q_1 [\!] \ldots [\!] l_n : Q_n\}\}$ where $i \in \{1,\ldots,n\}$, and $\{k^p!\langle\!\langle\kappa\rangle\!\rangle,\ k^{\overline{p}}?(\!(x)\!).Q\}$.

2.2 Operational Semantics

We formalise the operational semantics of the calculus by a one-step reduction relation $\rightarrow$, defined in Table 2, up to the standard structural equivalence $\equiv$ plus the rule $\mathbf{0}; P \equiv P$.

The reduction rules are based on those of the π-calculus with the session primitives [10, 12], taking into account the behaviour of sequencing. By the interplay between parallel composition and sequencing it is handy to introduce evaluation contexts.

Evaluation contexts are defined by:

$$\mathcal{E}[\,] := [\,] \mid \mathcal{E}[\,];P \mid \mathcal{E}[\,] \mid P \mid (\nu a)\mathcal{E}[\,] \mid (\nu k)\mathcal{E}[\,]$$

[Con]	$\mathcal{E}_1[a(x).P] \mid \mathcal{E}_2[\overline{a}(y).Q] \rightarrow (\nu k)(\mathcal{E}_1[P\{k^+/x\}] \mid \mathcal{E}_2[Q\{k^-/y\}])$	(k fresh)
[ConR]	$\mathcal{E}_1[\star a(x).P] \mid \mathcal{E}_2[\overline{a}(y).Q] \rightarrow (\nu k)(\mathcal{E}_1[P\{k^+/x\}] \mid \star a(x).P] \mid \mathcal{E}_2[Q\{k^-/y\}])$	(k fresh)
[ComV]	$\mathcal{E}_1[k^p!\langle e\rangle] \mid \mathcal{E}_2[k^{\overline{p}}?(x).Q] \rightarrow \mathcal{E}_1[\mathbf{0}] \mid \mathcal{E}_2[Q\{v/x\}]$	$(e \downarrow v)$
[Label]	$\mathcal{E}_1[k^p \lhd l_i.P] \mid \mathcal{E}_2[k^{\overline{p}} \rhd \{l_1 : Q_1 [\!] \ldots [\!] l_n : Q_n\}] \rightarrow \mathcal{E}_1[P] \mid \mathcal{E}_2[Q_i]$	$(1 \le i \le n)$
[ComS]	$\mathcal{E}_1[k^p!\langle\!\langle k_1^q\rangle\!\rangle] \mid \mathcal{E}_2[k^{\overline{p}}?(\!(x)\!).Q] \rightarrow Q\{k_1^q/x\} \mid \mathcal{E}_1[\mathbf{0}] \mid \mathcal{E}_2[\mathbf{0}]$	$(k_1 \notin \mathrm{bn}(\mathcal{E}_1[\,]) \,\&\, \mathrm{bn}(\mathcal{E}_2[\,]) \cap \mathrm{fn}(Q) = \emptyset)$
[If1]	$\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \rightarrow P_1$	$(e \downarrow \mathtt{true})$
[If2]	$\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \rightarrow P_2$	$(e \downarrow \mathtt{false})$
[Eval]	$P \rightarrow P' \Rightarrow \mathcal{E}[P] \rightarrow \mathcal{E}[P']$	
[Str]	$P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q \Rightarrow P \rightarrow Q$	

Table 2. Reduction

We say that a processes P is a *head subprocess* of a process Q if $Q \equiv \mathcal{E}[P]$ for some evaluation context $\mathcal{E}[\,]$. Examining the reduction rules it is easy to check that all prefixed processes (but in case of if-branching) in head positions reduce only if a dual subprocess is in head position too.

Rules [CON] and [CONR] are session initiation rules where two polarised fresh names are created, then restricted because the leading parts $P\{k^+/x\}$ and $Q\{k^-/y\}$ now share the channel k to start private interactions via k. In rule [CONR], we write directly the effect of the replication of the accept/request action, and we do not postulate $\star a(x).P \equiv \star a(x).P \,|\, a(x).P$: hence replication is triggered only in presence of a dual session request, a property which simplifies the soundness of the typing system.

Rule [COMV] sends data ($e \downarrow v$ means that the expression e evaluates to the value v). Rule [LABEL] selects the i-th branch.

In rule [COMS] the process which receives the live channel is put in parallel with the evaluation contexts. Notice that this does not happen in the other rules. This rule allows for a safe form of delegation: indeed the process that receives the live channel must proceed, even if it is put in a context of overlapping sessions, as it happens e.g. in Example 4.3 of [12] (Fpt server). This is not guaranteed by the "standard" version of the rule below:

$$\mathcal{E}_1[k^p!\langle\!\langle k_1^q \rangle\!\rangle] \mid \mathcal{E}_2[k^{\overline{p}}?(\!(x)\!).Q] \to \mathcal{E}_1[\mathbf{0}] \mid \mathcal{E}_2[Q\{k_1^q/x\}] \quad (k_1^q \notin \mathrm{bn}(\mathcal{E}_1[\,])$$

In fact by using this rule the process

$$\overline{a}(x).\overline{b}(y).(y?(\!(z)\!).z!\langle 5\rangle);x?(t).\mathbf{0} \,|\, a(x').b(y').y'!\langle\!\langle x' \rangle\!\rangle$$

reduces to $(\nu k)(k^-!\langle 5\rangle; k^+?(t).\mathbf{0})$ which is stuck, while its intended meaning should be that $y?(\!(z)\!).z!\langle 5\rangle$ completes and eventually 5 is communicated along k and replaced to t.

Notice that ";" is essential in order to identify which process must be executed in parallel with the contexts. The example above shows that – without the sequencing operator – we would be not able both to preserve progress and to require that a live channel is received before a communication on other live channels is executed. This is necessary e.g. for modelling a real estate agent who wants to be delegated by the owner before showing the house to potential buyers.

Rule [COMS] *subsumes* the channel passing rule named [PASS] in [12], since the standard version of this rule and [PASS] coincide if we ignore the sequencing. All other reduction rules are as usual.

3 Typing System for Progress Communication

The type system discussed in this section is designed to guarantee linearity of live channels, communication error freedom and progress.

3.1 Types

The full syntax of types is given in Table 3. *Partial session types*, ranged over by σ, represent sequences of communications, where ε is the empty communication, and $\sigma_1.\sigma_2$ consists of the communications in σ_1 followed by those in σ_2. We put $\varepsilon.\sigma = \sigma.\varepsilon = \sigma$ and we consider partial session types modulo this equality. The types !t and ?t express

(direction)	$\dagger$	$::= \; ! \mid ?$
(select/branch)	$\ddagger$	$::= \; \oplus \mid \&$
(partial session type)	σ	$::= \; \varepsilon \mid \dagger \mathsf{t} \mid \dagger \mathsf{s} \mid \sigma.\sigma \mid \ddagger\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$
(ended session type)	s	$::= \; \sigma.\mathsf{end} \mid \ddagger\{l_1 : \mathsf{s}_1, \ldots, l_n : \mathsf{s}_n\}$
(running session type)	τ	$::= \; \sigma \mid \mathsf{s}$
(standard type)	t	$::= \; [\mathsf{s}] \mid \mathsf{bool} \mid \mathsf{int} \mid \ldots$

Table 3. Types

respectively the sending and reception of a value of type t. The types $!\mathsf{s}$ and $?\mathsf{s}$ represent the exchange of a live channel, and therefore of an active session, with remaining communications determined by the ended session type s.

The *selection type* $\oplus\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$ represents the transmission of a label l_i chosen in the set $\{l_1, \ldots, l_n\}$ followed by the communications described by σ_i. The *branching type* $\&\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$ represents the reception of a label l_i chosen in the set $\{l_1, \ldots, l_n\}$ followed by the communications described by σ_i.

An *ended session type* s is a partial session type concatenated either with end or with a selection or branching whose branches in turn are both ended session types. It expresses a sequence of communications with its termination, i.e. no further communications on that channel are allowed at the end.

A *running session type*, τ, ranges over both partial and ended session types.

A *shared session type* $[\mathsf{s}]$ is the type of shared channels, and has one or more endpoints, denoted by end. *Standard types* t are either shared session types or ground types.

Each running session type τ has a corresponding *dual*, denoted $\overline{\tau}$, which is obtained as follows:

- $\overline{?} = !$ $\quad \overline{!} = ?$ $\quad \overline{\oplus} = \&$ $\quad \overline{\&} = \oplus$ $\quad \overline{\varepsilon} = \varepsilon$
- $\overline{\dagger \mathsf{t}} = \overline{\dagger}\mathsf{t}$ $\quad \overline{\dagger \mathsf{s}} = \overline{\dagger}\mathsf{s}$ $\quad \overline{\sigma_1.\sigma_2} = \overline{\sigma_1}.\overline{\sigma_2}$ $\quad \overline{\ddagger\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}} = \overline{\ddagger}\{l_1 : \overline{\sigma_1}, \ldots, l_n : \overline{\sigma_n}\}$
- $\overline{\sigma.\mathsf{end}} = \overline{\sigma}.\mathsf{end}$ $\quad \overline{\ddagger\{l_1 : \mathsf{s}_1, \ldots, l_n : \mathsf{s}_n\}} = \overline{\ddagger}\{l_1 : \overline{\mathsf{s}_1}, \ldots, l_n : \overline{\mathsf{s}_n}\}$.

Note that duality is an involution: $\overline{\overline{\tau}} = \tau$.

3.2 Motivating the Design of the Type System

This subsection discusses the key ideas behind the type system introduced in § 3.4 with some examples, focusing on progress.

Example 3.1 (Circularity of channels). As we explained in the Introduction, the order of session channels should be taken into account. Recall the processes P_2 and Q_2 from the Introduction:

$$P_2 = a(x).b(y).(x!\langle 3\rangle; x?(z).y!\langle Apple\rangle; P_2') \qquad Q_2 = \overline{a}(x).\overline{b}(y).(y?(z').x?(z'').x!\langle 5\rangle; Q_2')$$

These processes use the channels bound by a and b in reverse order, hence they lead to a deadlock. This is prevented by the type systems, which allows instead to compose P_2 e.g. with

$$Q_2'' \equiv \overline{a}(x).\overline{b}(y).(x?(z').x!\langle 5\rangle; y?(z'').Q_2')$$

For a similar reason, we prohibit processes which have self-circularity of a shared channel like:

$$P_3 \equiv a(x).a(y).(x!\langle 3\rangle; y!\langle 5\rangle) \mid \overline{a}(z).\overline{a}(t).(t?(t').z?(w).\mathbf{0})$$

which reduces to the deadlock process $(\nu k k_1)(k^+!\langle 3\rangle; k_1^+!\langle 5\rangle \mid k_1^-?(t').k^-?(w).\mathbf{0})$.

On the other hand, we want to allow self-circularity of live channels. Fortunately we can profit of the expressiveness of the session types to simplify our type system: since sequences of communications are *already structured* by types, we do not have to consider the ordering between the same live channels. For example $P_4 \equiv k^p!\langle 3\rangle; k^p?(y).\mathbf{0}$ and $P_5 \equiv a(x).(x!\langle 3\rangle; x?(y).\mathbf{0})$ shall be typable according to our system.

Example 3.2 (Sequencing and live channels). It is a well known constraint for the linearly typed π-calculi to disallow live channels that occur in repeated processes. For example, $P_6 \equiv \star a(x).k^+!\langle 3\rangle$ in parallel with $P_7 \equiv \overline{a}(y).\mathbf{0} \mid \overline{a}(z).\mathbf{0}$ reduces to $P_6 \mid k^+!\langle 3\rangle \mid k^+!\langle 3\rangle$. This can be easily avoided using a standard technique. However, the sequencing operator requires more careful analysis for preserving progress. Let us consider a slightly different process $P_8 \equiv \star a(x).\mathbf{0}; k^+!\langle 3\rangle$ which does *not* destroy linearity, but progress. For example, $P_7 \mid P_8$ reduces to P_8 where the linearity of k^+ is preserved, but $k^+!\langle 3\rangle$ is blocked forever.

Example 3.3 (Bound shared channels). A bound shared channel which does not have a dual to start a session can block the communication on live channels forever, as in $P_9 \equiv (\nu a)(\overline{a}(x).k^+!\langle 3\rangle) \mid k^-?(y).\mathbf{0}$. The problem does not arise if the shared channel a is free, since we can always compose with a dual process, as in $P_{10} \equiv \overline{a}(x).k^+!\langle 3\rangle \mid k^-?(y).\mathbf{0} \mid a(z).\mathbf{0}$.

Example 3.4 (Shared channel passing). Shared channels can be sent only if their dual processes can communicate without waiting other communications to succeed. For example, consider the processes:

$$P_{11} \equiv \overline{a}(t).t!\langle b\rangle \mid a(x).\overline{c}(y).x?(z).\overline{z}(q).q?(w).y?(w').\mathbf{0} \quad P_{12} \equiv c(s).b(r).(s!\langle 3\rangle; r!\langle 4\rangle)$$

Then $P_{11} \mid P_{12}$ reduces to $(\nu k_b k_c)(k_b^+?(w).k_c^+?(t).\mathbf{0} \mid k_c^-!\langle 3\rangle; k_b^-!\langle 4\rangle)$ which is a deadlock. A safe process is the parallel composition of P_{11} and $P'_{12} \equiv c(s).b(r).(r!\langle 4\rangle; s!\langle 3\rangle)$.

Example 3.5 (Session channel passing). Live channels can be sent only if the receiving process does not contain live channels, as shown by the process:

$$P_{13} \equiv a(x).b(y).x!\langle\!\langle y\rangle\!\rangle \mid \overline{a}(t).\overline{b}(z).t?(\!(t')\!).(t'!\langle 3\rangle; z?(w).\mathbf{0})$$

which reduces to the deadlock process $(\nu k_b)(k_b^+!\langle 3\rangle; k_b^-?(w).\mathbf{0})$. A similar, but sound process is $P'_{13} \equiv a(x).b(y).x!\langle\!\langle y\rangle\!\rangle \mid \overline{a}(t).\overline{b}(z).(t?(\!(t')\!).t'!\langle 3\rangle; z?(w).\mathbf{0})$, where $z?(w).\mathbf{0}$ is not in the body of $t?(\!(t')\!)$.

3.3 Typing Judgements

The typing judgements for expressions and processes are of the shape:

$$\Gamma \vdash e : \mathtt{t} \qquad \Gamma; \mathcal{S}; \mathcal{B} \vdash P : \Delta \,[\!]\, \mathcal{C}$$

where we define:

$$\Gamma ::= \emptyset \mid \Gamma, x:\mathtt{t} \mid \Gamma, a:[\mathtt{s}] \qquad \mathcal{S} ::= \emptyset \mid \mathcal{S}, a \qquad \mathcal{B} ::= \emptyset \mid \mathcal{B}, a$$
$$\Delta ::= \emptyset \mid \Delta, \kappa:\tau \mid \Delta, \diamond \qquad \mathcal{C} ::= \emptyset \mid \mathcal{C}, \lambda \mid \mathcal{C}, \lambda \prec \lambda'$$

Γ is the *standard environment* which associates variables to types and shared channel names to shared session types; $\mathcal{S}$ (resp. $\mathcal{B}$) is the set of *shared channel names* which can be *sent* (resp. *bound*); Δ is the *session environment* which associates live channels to running session types, and it can also contain the special symbol $\diamond$. The session environment Δ represents the open communication protocols of a process; the occurrence of $\diamond$ in Δ is used to prevent that any process sequentially composed with the term to which $\diamond$ has been assigned, might contain any occurrence of free live channels (see the definition of $\Delta \cdot \Delta'$ in Table 5). $\mathcal{C}$ is the *channel relation*, which is intended to give information about the ordering in the usage of channels. In $\mathcal{C}$ the metavariable λ ranges over shared and live channels. A well-formed channel relation is irreflexive w.r.t. shared channel names, and cannot contain cycles (see the next subsection).

3.4 Type System

Table 4 defines the type system. We omit the typing rules for expressions which are standard and identical with [24]. For typing processes, we use the auxiliary operators defined in Table 5. We list the key points of the typing rules for processes.

Session Initiation As discussed in the examples, accept/request processes whose subjects are going to be bound or sent require particular care. The most liberal typing rules are *Acc* and *Req* where the shared channel can neither be bound nor sent. The resulting session environment is obtained by erasing the type of the bound channel x and the resulting channel relation is obtained by replacing x by a to prevent the circular ordering between names.

If the shared channel is a permanent accept, or when it can be bound but not sent, we cannot allow live channels in the continuation processes (see Examples 3.2 and 3.3). In rules *AccB*, *ReqB*, and *Acc**, the satisfaction of this condition is enforced by requiring that the session environment of the body process only contains the current channel and by typing the whole process with the session environment $\{\diamond\}$. Notice that session environments containing $\diamond$ cannot be composed with session environments containing channels by the definition of $\Delta \cdot \Delta'$ given in Table 5.

If the shared channel can be sent but it cannot be bound we need to require that all communications on that channel can be executed without requiring other channels to communicate (Examples 3.4). This can be achieved by asking that the channel is minimal in the current channel relation, i.e. using $\mathcal{C} \setminus\!\setminus x$ (defined in Table 5) in the conclusion. We ask $\mathcal{C} \setminus\!\setminus x$ to be the channel relation in the conclusion of rules *AccS*, *ReqS*, and *Acc***S*, convening that the rules cannot be applied if it is undefined.

Rules *AccBS* and *ReqBS* put the above restrictions together, and are used to type shared channels which can be both bound and sent. In rules *Acc***S*, *AccBS* and *ReqBS* the subject can also be a variable, which will be replaced by a channel name which surely can be sent and possibly can be bound.

Session Communication These rules add relevant information to session environments and to channel relations. Rule *Snd* checks that only shared channels in the set $\mathcal{S}$ are sent. The resulting session environment is $\{\kappa\!:\!\mathtt{t}\}$, where κ is the subject of the sent process and $\mathtt{t}$ is the type of the sent expression. The resulting channel relation contains the name (without polarity) of the subject, where we define $\ell(\kappa) = k$ if $\kappa = k^p$ and $\ell(\kappa) = \kappa$ otherwise.

Rule *Rcv* uses the composition operator defined in Table 5 between session environments, which extends that one between running session types. In this way we can

$$\frac{\Gamma\vdash a:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,x:\mathsf{s}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{S}\cup\mathcal{B}}{\Gamma;\mathcal{S};\mathcal{B}\vdash a(x).P:\Delta\,[\!]\,\mathcal{C}\{a/x\}}\,Acc \qquad \frac{\Gamma\vdash a:[\overline{\mathsf{s}}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,x:\mathsf{s}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{S}\cup\mathcal{B}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \overline{a}(x).P:\Delta\,[\!]\,\mathcal{C}\{a/x\}}\,Req$$

$$\frac{\Gamma\vdash a:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{S}}{\Gamma;\mathcal{S};\mathcal{B}\vdash a(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\{a/x\}}\,AccB \qquad \frac{\Gamma\vdash a:[\overline{\mathsf{s}}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{S}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \overline{a}(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\{a/x\}}\,ReqB$$

$$\frac{\Gamma\vdash a:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{S}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \star a(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\{a/x\}}\,Acc^\star \qquad \frac{\Gamma\vdash c:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \star c(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\setminus\!\setminus x}\,Acc^\star S$$

$$\frac{\Gamma\vdash a:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,x:\mathsf{s}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{B}}{\Gamma;\mathcal{S};\mathcal{B}\vdash a(x).P:\Delta\,[\!]\,\mathcal{C}\setminus\!\setminus x}\,AccS \qquad \frac{\Gamma\vdash a:[\overline{\mathsf{s}}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,x:\mathsf{s}\,[\!]\,\mathcal{C}\quad a\notin\mathcal{B}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \overline{a}(x).P:\Delta\,[\!]\,\mathcal{C}\setminus\!\setminus x}\,ReqS$$

$$\frac{\Gamma\vdash c:[\mathsf{s}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash c(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\setminus\!\setminus x}\,AccBS \qquad \frac{\Gamma\vdash c:[\overline{\mathsf{s}}]\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \overline{c}(x).P:\{\diamond\}\,[\!]\,\mathcal{C}\setminus\!\setminus x}\,ReqBS$$

$$\frac{\Gamma\vdash e:\mathsf{t}\quad \text{if } e=a \text{ then } a\in\mathcal{S}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \kappa!\langle e\rangle:\{\kappa:!\mathsf{t}\}\,[\!]\,\{\ell(\kappa)\}}\,Snd \qquad \frac{\Gamma,x:\mathsf{t};\mathcal{S};\mathcal{B}\vdash P:\Delta\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \kappa?(x).P:\{\kappa:?\mathsf{t}\}\cdot\Delta\,[\!]\,\mathsf{pre}(\{\ell(\kappa)\},\mathcal{C})}\,Rcv$$

$$\frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,\kappa:\tau_i\,[\!]\,\mathcal{C}\quad (1\le i\le n)}{\Gamma;\mathcal{S};\mathcal{B}\vdash \kappa\triangleleft l_i.P:\Delta,\kappa:\oplus\{l_1:\tau_1,\ldots,l_n:\tau_n\}\,[\!]\,\mathsf{pre}(\{\ell(\kappa)\},\mathcal{C})}\,Sel$$

$$\frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P_i:\Delta,\kappa:\tau_i\,[\!]\,\mathcal{C}_i\quad (i=1,\ldots,n)}{\Gamma;\mathcal{S};\mathcal{B}\vdash l_1:P_1[\!]\ldots[\!]l_n:P_n:\Delta,\kappa:\&\{l_1:\tau_1,\ldots,l_n:\tau_n\}\,[\!]\,\mathsf{pre}(\{\ell(\kappa)\},\cup_{1\le i\le n}\mathcal{C}_i)}\,Bra$$

$$\frac{\mathsf{s}\neq\varepsilon.\mathsf{end}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \kappa!\langle\!\langle\kappa'\rangle\!\rangle:\{\kappa:!\mathsf{s},\kappa':\mathsf{s}\}\,[\!]\,\{\ell(\kappa),\ell(\kappa'),\ell(\kappa)\prec\ell(\kappa')\}}\,CSnd$$

$$\frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\{x:\mathsf{s}\}\,[\!]\,\{x\}\quad \mathsf{s}\neq\varepsilon.\mathsf{end}}{\Gamma;\mathcal{S};\mathcal{B}\vdash \kappa?(\!(x)\!).P:\{\kappa:?\mathsf{s}\}\,[\!]\,\{\ell(\kappa)\}}\,CRcv \qquad \frac{\Gamma\vdash e:\mathsf{bool}\quad \Gamma;\mathcal{S};\mathcal{B}\vdash P_i:\Delta\,[\!]\,\mathcal{C}\quad (i=1,2)}{\Gamma;\mathcal{S};\mathcal{B}\vdash \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2:\Delta\,[\!]\,\mathcal{C}}\,If$$

$$\frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta\,[\!]\,\mathcal{C}\quad \Gamma;\mathcal{S};\mathcal{B}\vdash Q:\Delta'\,[\!]\,\mathcal{C}'}{\Gamma;\mathcal{S};\mathcal{B}\vdash P;Q:\Delta\cdot\Delta'\,[\!]\,\mathsf{pre}(\mathcal{C},\mathcal{C}')}\,Seq \qquad \frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta\,[\!]\,\mathcal{C}\quad \Gamma;\mathcal{S};\mathcal{B}\vdash Q:\Delta'\,[\!]\,\mathcal{C}'}{\Gamma;\mathcal{S};\mathcal{B}\vdash P\,|\,Q:\Delta\cup\Delta'\,[\!]\,\mathcal{C}\cup\mathcal{C}'}\,Par$$

$$\frac{\Gamma,a:[\mathsf{s}];\mathcal{S};\mathcal{B}\vdash P:\Delta\,[\!]\,\mathcal{C}\quad a\in\mathcal{B}}{\Gamma;\mathcal{S}\setminus a;\mathcal{B}\setminus a\vdash(\nu a)P:\Delta\,[\!]\,\mathcal{C}\setminus a}\,HidingS \qquad \frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,k^p:\tau,k^{\overline{p}}:\overline{\tau}\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash(\nu k)P:\Delta\,[\!]\,\mathcal{C}\setminus k}\,HidingL$$

$$\frac{}{\Gamma;\mathcal{S};\mathcal{B}\vdash\mathbf{0}:\emptyset\,[\!]\,\emptyset}\,Inact \qquad \frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta\,[\!]\,\mathcal{C}\quad \kappa\notin\mathrm{dom}(\Delta)}{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,\kappa:\varepsilon\,[\!]\,\mathcal{C}}\,Weak1 \qquad \frac{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,\kappa:\varepsilon\,[\!]\,\mathcal{C}}{\Gamma;\mathcal{S};\mathcal{B}\vdash P:\Delta,\kappa:\varepsilon.\mathsf{end}\,[\!]\,\mathcal{C}}\,Weak2$$

Table 4. Typing Rules

prefix by ?t the possible communications on channel κ prescribed by Δ. In the obtained channel relation all channels in $\mathcal{C}$ are bigger than $\ell(\kappa)$ by the definition of $\mathsf{pre}(\mathcal{C},\mathcal{C}')$ given in Table 5.

In rules *Sel* and *Bra* all τ_i are either partial session types or ended session types – this is guaranteed by the syntax of conditional session types (see Table 3).

The condition $\tau \neq \varepsilon.\mathsf{end}$ in rules *CSnd* and *CRcv* allows to exchange only live channels which are not consumed, a reasonable requirement for a good programming discipline. Example 3.5 justifies the requirement that x is the only live channel of P.

Compositional and Structural Rules Rule *Seq* takes into account that all communications in P must be executed before the communications in Q. Instead in rule *Par* the communications in P and Q can be executed in any order, and for this reason we take the unions of session environments and channel relations, with the proviso that $\Delta \cup \Delta'$ is defined only if $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset$. In the rules for restrictions we use $\mathcal{C} \setminus \lambda$ defined in Table 3, while $\mathcal{S} \setminus a$ is simply the set $\mathcal{S}$ without a and similarly for $\mathcal{B} \setminus a$. The weakening rules are standard and necessary to type branching processes.

We assume that the typing rules are applicable only if *all channel relations in the conclusion of typing rules do not contain cycles and do not relate a shared channel with itself*: such channel relations are said to be well-formed. The first condition disallows a cycle between two names, while the second condition disallows $a \prec a$, but it allows both $k \prec k$ and $x \prec x$ in channel relations. These conditions are justified below through Example 3.1.

3.5 Justifying Examples

We end this section by briefly explaining why the negative examples given in § 3.2 cannot be typed, while the positive ones are typable. For the channel relations, we only

---

Composition for Running Session Types and Session Environments

$$\tau \cdot \tau' = \begin{cases} \tau.\tau' & \text{if } \tau \text{ is a partial session type and } \tau' \text{ is a running session type} \\ \bot & \text{otherwise.} \end{cases}$$

$$\Delta \cdot \Delta' = \begin{cases} \Delta & \text{if } \diamond \in \Delta \text{ and } \Delta' \subseteq \{\diamond\}; \\ \Delta \setminus \Delta' \cup \Delta' \setminus \Delta \cup \{\kappa{:}\Delta(\kappa) \cdot \Delta'(\kappa) \mid \kappa \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta')\} \cup \{\diamond \mid \diamond \in \Delta'\} & \\ \quad \text{if } \diamond \notin \Delta \text{ and } \forall \kappa \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') \colon \Delta(\kappa) \cdot \Delta'(\kappa) \neq \bot; & \\ \bot & \text{otherwise.} \end{cases}$$

Operators for Channel Relations

$$\mathcal{C} \setminus \lambda = \{\lambda_1 \prec \lambda_2 \mid \lambda_1 \prec \lambda_2 \in \mathcal{C} \ \&\ \lambda_1 \neq \lambda \ \&\ \lambda_2 \neq \lambda\} \cup \{\lambda' \mid \lambda' \in \mathcal{C} \ \&\ \lambda' \neq \lambda\}$$

$$\mathcal{C} \setminus\!\!\setminus x = \begin{cases} \mathcal{C} \setminus x & \text{if } x \text{ is minimal in } \mathcal{C} \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathsf{pre}(\mathcal{C},\mathcal{C}') = (\mathcal{C} \cup \mathcal{C}' \cup \{\lambda \prec \lambda' \mid \lambda \in \mathcal{C} \ \&\ \lambda' \in \mathcal{C}'\})^*$$

where $\mathcal{C}^*$ is the transitive closure of $\mathcal{C}$ and λ is minimal in $\mathcal{C}$ if $\nexists \lambda' \prec \lambda \in \mathcal{C}$.

Table 5. Operators for Types and Environments

write the order of the channels, omitting the set of channels.

Example 3.1: The channel relation of P_2 and Q_2'' is $\{a \prec b\}$, while the channel relation of Q_2 is $\{b \prec a\}$. Therefore $P_2 \mid Q_2$ creates a cyclic relation, which is not well-formed. Hence it is untypable. On the other hand, $P_2 \mid Q_2''$ is typable. Similarly, P_3 is not typable since $\{a \prec a\}$ is not a well-formed channel relation, while P_4 and P_5 are typable since $\{k \prec k\}$ and $\{x \prec x\}$ are well-formed channel relations.

Example 3.2: The process P_6 cannot be typed since Rules *Acc** and *Acc***S* require the session environment of the body of the repeated accept to contain only x as subject, while the session environment of $k^+!\langle 3\rangle$ contains k^+ as subject. The process P_8 is untypable since $\star a(x).\mathbf{0}$ must be typed by the session environment $\{\diamond\}$ (see rules *Acc** and *Acc***S*), and we cannot sequentially compose $\{\diamond\}$ with $k^+!\langle 3\rangle$ by the definition of "$\cdot$" (used in rule *Seq*).

Example 3.3: The argument of Example 3.2 shows that $\overline{a}(x).k^+!\langle 3\rangle$ cannot be typed by rules *ReqB* and *ReqBS*, hence P_9 is untypable, while P_{10} is typable since we can apply rules *Req* and *ReqS*.

Example 3.4: The process P_{12} cannot be typed by using rules *ReqS* and *ReqBS*, since r is not minimal in its channel relation $\{s \prec r\}$. Instead the process P_{12}' is typable using rules *ReqS* and *ReqBS*.

Example 3.5: The process $t?(\!(t')\!).(t'!\langle 3\rangle; z?(w).\mathbf{0})$ in P_{13} cannot be typed, since rule *CRcv* requires the session environment of the body of the receive to contain only t' as subject. Rule *CRcv* allows to type $t?(\!(t')\!).t'!\langle 3\rangle$ instead, hence P_{13}' is typable.

4 Subject Reduction and Progress

This section discusses the features of our type system. It naturally splits into two parts: subject reduction and progress. Proofs are given in outline, by stating the needed lemmas.

4.1 Subject Reduction

The basic property of substitutivity of values and live channels to variables within derivable typing judgments is easily checked by induction on derivations:

Lemma 4.1 (Substitution Lemma).

1. *If* $\Gamma, x:\mathrm{t}; \mathcal{S}; \mathcal{B} \vdash P : \Delta \,[\!]\, \mathcal{C}$ *and* $\Gamma \vdash v:\mathrm{t}$, *then* $\Gamma; \mathcal{S}; \mathcal{B} \vdash P\{v/x\} : \Delta \,[\!]\, \mathcal{C}$.
2. *If* $\Gamma; \mathcal{S}; \mathcal{B} \vdash P : \Delta, x:\tau \,[\!]\, \mathcal{C}$ *and* k *fresh, then* $\Gamma; \mathcal{S}; \mathcal{B} \vdash P\{k^p/x\} : \Delta, k^p:\tau \,[\!]\, \mathcal{C}\{k^p/x\}$.

Subject Equivalence, namely the invariance of typing judgments w.r.t. structural equivalence is proved straightforwardly by case analysis of the applied equivalence law.

Lemma 4.2 (Subject Equivalence). *If* $\Gamma; \mathcal{S}; \mathcal{B} \vdash P : \Delta \,[\!]\, \mathcal{C}$ *and* $P \equiv Q$, *then* $\Gamma; \mathcal{S}; \mathcal{B} \vdash Q : \Delta \,[\!]\, \mathcal{C}$.

Subject Reduction, namely the invariance of derivable typing judgments w.r.t. reduction, does not hold literally, since session types are shortened by reduction and the channel relation becomes a subrelation of the original one. However a weaker statement, which suffices for the present purposes, can be established modulo inclusion of

channel relations and of prefixing of session environments, called below evaluation order.

Definition 4.3 (Inclusion). *The* inclusion between channel relations $\mathcal{C} \Subset \mathcal{C}'$ *holds if* $\lambda \in \mathcal{C}$ *implies* $\lambda \in \mathcal{C}'$ *and* $\lambda \prec \lambda' \in \mathcal{C}$ *implies* $\lambda \prec \lambda' \in \mathcal{C}'$*, for all* λ, λ'*.*

One might think of an ordering $\mathcal{C}$ as a graph (V,E) where V is the set of channels in $\mathcal{C}$, and E is just the relation $\prec$; therefore $\mathcal{C} \Subset \mathcal{C}'$ holds if and only if $\mathcal{C}$ is a subgraph of $\mathcal{C}'$.

The partial order among pairs of session environments defined next reflects the difference between two running session types before and after one step reduction.

Definition 4.4 (Evaluation Order).

1. $\sqsubseteq$ *is defined as the smallest partial order on running session types such that:* $\varepsilon \sqsubseteq \tau$*;* $\varepsilon.\mathsf{end} \sqsubseteq \mathsf{s}$*;* $\sigma_i \sqsubseteq \ddagger\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$*;* $\mathsf{s}_i \sqsubseteq \ddagger\{l_1 : \mathsf{s}_1, \ldots, l_n : \mathsf{s}_n\}$*; and* $\sigma \sqsubseteq \sigma'$ *implies* $\sigma \cdot \tau \sqsubseteq \sigma' \cdot \tau$*.*
2. $\sqsubseteq$ *is extended to session environments as follows:* $\Delta \sqsubseteq \Delta'$ *if* $\diamond \in \Delta$ *implies* $\diamond \in \Delta'$*; and* $k^p : \tau \in \Delta$ *implies* $k^p : \tau' \in \Delta'$ *and* $\tau \sqsubseteq \tau'$*.*

Before stating Subject Reduction, we recall the important notion of balanced session environments [10]. A session environment Δ is *balanced* if $k^p : \tau$ and $k^{\overline{p}} : \tau' \in \Delta$ imply $\tau' = \overline{\tau}$. The need of restricting to balanced session environments is illustrated by the process $k_1^p!\langle \mathtt{true} \rangle \mid k_1^{\overline{p}}?(x).k_2^p!\langle x+1 \rangle$, which would be typable by unbalanced session environments, whereas it reduces to $k_2^p!\langle \mathtt{true}+1 \rangle$ leading to a run-time error.

Theorem 4.5 (Subject Reduction).

1. *If* $\Gamma \vdash e : \mathsf{t}$ *and* $e \downarrow v$*, then* $\Gamma \vdash v : \mathsf{t}$*.*
2. *If* $\Gamma; \mathcal{S}; \mathcal{B} \vdash P : \Delta \,[\!]\, \mathcal{C}$*, where* Δ *is balanced, and* $P \to Q$*, then* $\Gamma; \mathcal{S}; \mathcal{B} \vdash Q : \Delta' \,[\!]\, \mathcal{C}'$*, for some* $\Delta', \mathcal{C}'$ *such that* Δ' *is balanced,* $\Delta' \sqsubseteq \Delta$ *and* $\mathcal{C}' \Subset \mathcal{C}$*.*

The main part of the theorem, namely (2), says that after a session has begun the required communications are always executed in the expected order specified by channel orderings $\mathcal{C}' \Subset \mathcal{C}$ and session environments $\Delta' \sqsubseteq \Delta$.

4.2 Progress

This subsection discusses the main result of this paper, i.e. that typable processes which contain live channels can always execute, unless there are either accept or request head subprocesses with free subjects waiting for the dual processes. We formalise this property as follows:

Definition 4.6 (Progress). *A process* P *has the* progress property *if* $P \to^* P'$ *implies that either* P' *does not contain live channels or* $P' \,|\, Q \to$ *for some* Q *such that* $P' \,|\, Q$ *is well-typed and* $Q \not\to$*.*

A process P has the progress property if it is not blocked, and a process is blocked if it is some "bad" normal form. In our setting this means that some open session is incomplete. This might happen because some internal communication cannot occur and the obstacle cannot be removed either by internal or by external communications, namely by communications relative to other sessions. This is why we do *not* consider any irreducible process as blocked, rather we say that even an irreducible P has the progress

property whenever it is able to interact in parallel with some Q such that $P \mid Q$ is well-typed: we ask Q itself to be irreducible to ensure that P actually participates in the reduction step.

The goal of this section is to show that any process representing a state in the running of some well-typed "program", has the progress property. Put together with Subject Reduction, this implies the safety of well-typed programs w.r.t. execution. By analogy with the theory of sequential languages, programs are closed processes; moreover they do not contain live channels, since the latters only appear while running. We call closed typable processes without live channels *initial*.

Definition 4.7 (Initial Processes). *A process P is* initial *if $\Gamma;\mathcal{S};\mathcal{B} \vdash P : \emptyset \,[\!]\, \mathcal{C}$ for some Γ not containing variables and some $\mathcal{C}$, with a deduction which does not use rule $HidingL$.*

Notice that initial processes cannot contain free live channels since the session environment is empty, nor bound live channels since to type them rule $HidingL$ is needed.

As in the case of type systems for partial deadlock-freedom, we have first to establish a relation between the ordering in the usage of channels, especially the live ones, and their formal counterpart in our system, namely channel relations. To make this precise we define the auxiliary notion of precedence between prefixed subprocesses.

Definition 4.8 (Precedence). *The* precedence relation *between prefixed processes inside a process is defined by: T precedes T' in P if P contains either $T = C'[T']$ or $C[T];C'[T']$, where $C[\,],C'[\,]$ denote arbitrary contexts.*

The main lemma states that in a process obtained by reducing an initial process a live channel which is minimal in the channel relation can only be preceded by an accept/request on a free channel.

Lemma 4.9. *Let P_0 be initial and $P_0 \to^* (\nu\vec{a}\vec{k})P$ and $\Gamma;\mathcal{S};\mathcal{B} \vdash P : \Delta \,[\!]\, \mathcal{C}$ be derivable and let T be a subprocess of P with subject k^p and let k be minimal in $\mathcal{C}$, then either P contains as head subprocess an accept or request on a free channel, or P contains T as head subprocess.*

Proof. (Sketch) The proof is a consequence of the following properties:

(P1) If P_0 is initial and $P_0 \to^* P$ and T precedes T' in P and k^p is the subject of T, then $k^{\overline{p}}$ cannot be the subject of T'.

(P2) If T precedes T' in a typable P and the subject of T' is a free live channel, then the subject of T is neither an accept/request on a bound channel nor a permanent accept.

(P3) Let P be typable with channel order $\mathcal{C}$ and let T precede T' in P. If k^p is the subject of T and k_1^q is the subject of T' and both k and k_1 are free in P, then we have $k \prec k_1 \in \mathcal{C}$.

Property (P1) can be shown by induction on reduction.

Property (P2) is guaranteed by the use of $\diamond$ in the type system. If the subject of T' is a free live channel, then the session environment for typing a process which contains T' cannot be empty. By the hypothesis P contains either $T = C[T']$ or $C[T];C'[T']$. In both cases, if T is an accept/request on a bound channel or a permanent accept, then T

must be typed by one of the rules *AccB, ReqB, Acc⋆, AccBS, ReqBS, Acc⋆S*. These rules prescribe the session environment of the body of T only contains the channel variable bound by T and the session environment of T itself to be $\{\diamond\}$. So if $T = C[T']$ the thesis follows immediately, otherwise it follows from the definition of "$\cdot$" and the typing rule *Seq*.

For property (P3) notice that by the hypothesis P contains either $T = C[T']$ or $C[T];C'[T']$. In the former case, since k^p is the subject of T, we know that $k \in \mathcal{C}$; similarly, since k_1^q is the subject of T', we know that $k_1 \in \mathcal{C}'$, for some $\mathcal{C}'$ such that $\mathcal{C} = \mathsf{pre}(k, \mathcal{C}')$, since $C[T']$ is the conclusion of one rule among *Rcv, Sel, Bra* (not of *Send* or *CSend* because no prefix could occur inside). This implies that $k_1 \prec k \in \mathcal{C}$ as desired. The case P contains $C[T];C'[T']$ is similar and easier.

A key notion in showing progress is the natural correspondence between communication patterns and shapes of session types.

Definition 4.10. *Define $\propto$ between prefixed processes and partial/ended session types, as follows:*

$$\kappa!\langle e\rangle \propto !\mathsf{t} \quad \kappa?(x).P \propto ?\mathsf{t}$$
$$\kappa \triangleleft l_i.P \propto \oplus\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \quad \kappa \triangleright \{l_1 : P_1 [\!] \ldots [\!] l_n : P_n\} \propto \&\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$$
$$\kappa \triangleleft l_i.P \propto \oplus\{l_1 : \mathsf{s}_1, \ldots, l_n : \mathsf{s}_n\} \quad \kappa \triangleright \{l_1 : P_1 [\!] \ldots [\!] l_n : P_n\} \propto \&\{l_1 : \mathsf{s}_1, \ldots, l_n : \mathsf{s}_n\}$$
$$\kappa!\langle\!\langle \kappa' \rangle\!\rangle \propto !\mathsf{s} \quad \kappa?(\!(x)\!).P \propto ?\mathsf{s}$$

where $i \in \{1, \ldots, n\}$.

Then, by analysis of deductions using standard generation lemmas, we have:

Lemma 4.11. *If P is typable with a session environment Δ such that $\Delta(k^p) = \tau \notin \{\varepsilon, \varepsilon.\mathsf{end}\}$, then P contains at least one prefix with subject k^p. Moreover if T is the prefix with subject k^p which precedes in P all other prefixes with subject k^p, then either $T \propto \tau$ or $\tau = \sigma.\tau'$ and $T \propto \sigma$.*

Since rule *HidingL* only restricts dual live channels with dual session types, we only get session environments which are balanced if we start from initial processes.

Lemma 4.12. *If P_0 is initial and $P_0 \to^* (\nu \vec{a}\vec{k})P$, then there exist $\Gamma, \mathcal{S}, \mathcal{B}, \Delta, \mathcal{C}$ such that $\Gamma; \mathcal{S}; \mathcal{B} \vdash P : \Delta [\!] \mathcal{C}$ and Δ is balanced.*

We eventually come to the Progress Theorem: for each process P obtained by reducing an initial process if P contains an open session, then there is an irreducible process Q such that the parallel composition $P|Q$ is well-typed too and it always reduces, also in presence of interleaved sessions.

Theorem 4.13 (Progress). *All initial processes have the progress property.*

Proof. Let P_0 be initial and $P_0 \to^* P$. If P does not contain live channels or $P \to P'$ there is nothing to prove. No head prefixed process in P is an if-then-else statement: otherwise P would reduce, since P is closed (being P_0 closed) and any closed boolean value is either `true` or `false`. If one head prefixed subprocess in P is an accept/request on a free channel a, then a must be in the domain of the standard environment Γ used to type P_0 and P. Let $\Gamma(a) = [\mathsf{s}]$ and a head prefixed subprocess in P on a be an accept process.

$$
\begin{array}{ll}
\beta(\varepsilon,x) & = (\mathbf{0},\emptyset)\\
\beta(\varepsilon.\mathsf{end},x) & = (\mathbf{0},\emptyset)\\
\beta(!\mathsf{bool}.\tau,x) & = (x!\langle \mathtt{true}\rangle;\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\tau,x)))\\
\ldots & \\
\beta(![\mathsf{s}].\tau,x) & = ((\nu b)x!\langle b\rangle;\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\tau,x))) \quad b \text{ fresh}\\
\beta(?\mathsf{t}.\tau,x) & = (x?(y).\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\tau,x))) \quad y \text{ fresh}\\
\beta(!\mathsf{s}.\tau,x) & = (x!\langle\!\langle y\rangle\!\rangle;\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\tau,x))\cup\{y:\mathsf{s}\}) \quad y \text{ fresh}\\
\beta(?\mathsf{s}.\tau,x) & = (x?(\!(y)\!).\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\tau,x))) \quad y \text{ fresh}\\
\beta(\oplus\{l_1:\sigma_1,\ldots,l_n:\sigma_n\}.\tau,x) & = (x\triangleleft l_1.\mathsf{p}_1(\beta(\sigma_1,x));\mathsf{p}_1(\beta(\tau,x)),\mathsf{p}_2(\beta(\sigma_1,x))\cup\mathsf{p}_2(\beta(\tau,x)))\\
\beta(\&\{l_1:\sigma_1,\ldots,l_n:\sigma_n\}.\tau,x) & = (x\triangleright\{l_1:\mathsf{p}_1(\beta(\sigma_1,x))[\!]\ldots[\!]l_n:\mathsf{p}_1(\beta(\sigma_n,x))\};\mathsf{p}_1(\beta(\tau,x)),\Delta)\\
 & \quad \text{where } \Delta=\bigcup_{1\le i\le n}\mathsf{p}_2(\beta(\sigma_i,x))\cup\mathsf{p}_2(\beta(\tau,x))\\
\beta(\oplus\{l_1:\mathsf{s}_1,\ldots,l_n:\mathsf{s}_n\},x) & = (x\triangleleft l_1.\mathsf{p}_1(\beta(\mathsf{s}_1,x)),\mathsf{p}_2(\beta(\mathsf{s}_1,x)))\\
\beta(\&\{l_1:\mathsf{s}_1,\ldots,l_n:\mathsf{s}_n\},x) & = (x\triangleright\{l_1:\mathsf{p}_1(\beta(\mathsf{s}_1,x))[\!]\ldots[\!]l_n:\mathsf{p}_1(\beta(\mathsf{s}_n,x))\},\bigcup_{1\le i\le n}\mathsf{p}_2(\beta(\mathsf{s}_i,x)))
\end{array}
$$

where $(\ ,\ )$ is a pair constructor and $\mathsf{p}_1(\ )$, $\mathsf{p}_2(\ )$ are standard projections.

Table 6. Mapping β

Then we can build Q as a request process on a which offers in the given order all the communications prescribed by $\overline{\mathsf{s}}$ according to the relation $\propto$. Notice that if $\overline{\mathsf{s}}$ prescribes to send live channels, then the body of Q must contain pairs of accept/request which produce these live channels. We can choose fresh names as subjects of these pairs of accept/request and put them in parallel, the accepts followed by all the communications prescribed by $\overline{\mathsf{s}}$. More precisely, first we define in Table 6 the mapping β from a session type and a channel variable to a pair of a process and a session environment. Second we define the mapping α from a session type and a channel variable to a process as follows:

$$
\begin{array}{l}
\alpha(\mathsf{s},x) = (\nu b_1\ldots b_n)(b_1(y_1).\ldots.b_n(y_n).\mathsf{p}_1(\beta(\mathsf{s},x))\,|\,\overline{b_1}(y_1).\alpha(\overline{\mathsf{s}_1},y_1)\,|\,\ldots\,|\,\overline{b_n}(y_n).\alpha(\overline{\mathsf{s}_n},y_n))\\
\qquad \text{if } \mathsf{p}_2(\beta(\mathsf{s},x))=\{y_1:\mathsf{s}_1,\ldots,y_n:\mathsf{s}_n\} \text{ and } b_1,\ldots,b_n \text{ are fresh.}
\end{array}
$$

Let $\Gamma(a)=[\mathsf{s}]$: then we can take $Q=\overline{a}(x).\alpha(\overline{\mathsf{s}},x)$ if P is an accept on the channel a, or $Q=a(x).\alpha(\mathsf{s},x)$ if P is a request on the channel a. If P_0 is initial then $\Gamma;\mathcal{S};\mathcal{B}\vdash P_0:\emptyset[\!]\mathcal{C}$ for some $\Gamma,\mathcal{S},\mathcal{B},\mathcal{C}$. By Theorem 4.5(2) we get $\Gamma;\mathcal{S};\mathcal{B}\vdash P:\emptyset[\!]\mathcal{C}'$ for some $\mathcal{C}'\Subset\mathcal{C}$. It is easy to verify by induction on the construction of Q that $\Gamma;\mathcal{S};\mathcal{B}\vdash Q:\Delta[\!]\mathcal{C}''$ for some $\mathcal{C}''\Subset\mathcal{C}'$ and Δ, where $\Delta=\emptyset$ if $\mathsf{p}_2(\beta(\mathsf{s},x))=\emptyset$ and $\Delta=\{\diamond\}$ otherwise. Since $\mathcal{C}''\Subset\mathcal{C}'$ implies that $\mathcal{C}'\cup\mathcal{C}''$ is well-formed, we conclude $\Gamma;\mathcal{S};\mathcal{B}\vdash P\,|\,Q:\Delta[\!]\mathcal{C}'\cup\mathcal{C}''$.

Otherwise P does not contain as head subprocess an accept or a request on a free shared channel, but P contains live channels. Let $P\equiv(\nu\vec{a}\vec{k})Q$, where $\vec{a}$ includes the set of all shared channels which are subjects of the head prefixed processes in P and $\vec{k}$ is the set of all live channels which occur in P. By Lemma 4.12, we know that $\Gamma;\mathcal{S};\mathcal{B}\vdash Q:\Delta[\!]\mathcal{C}$ for some $\Gamma,\mathcal{S},\mathcal{B},\Delta$ and $\mathcal{C}$. Let k be a minimal live channel in $\mathcal{C}$. This implies $k^p:\tau\in\Delta$ for some τ such that $\tau\notin\{\varepsilon,\varepsilon.\mathsf{end}\}$. By Lemma 4.12 Δ is balanced and then $k^{\overline{p}}:\overline{\tau}\in\Delta$. By Lemma 4.11 the channels k^p and $k^{\overline{p}}$ must occur in P. By Lemma 4.9 there are two head prefixed processes in P with subjects k^p and its $k^{\overline{p}}$, respectively. Notice that k^p and $k^{\overline{p}}$ have dual types, so that by Lemma 4.11 they are the subject of dual communication actions: it follows that P reduces.

5 Conclusion and Future Works

This paper proposed the first session typing system for the progress property on interleaving sessions, which are not necessarily nested. The resulting typing system ensures a strong progress property for a calculus allowing creation of new names and full concurrency, significantly enlarging the approach taken in [6, 8]. In spite of the richness of the calculus, the typing system is based on the intuitive idea of channel causality without additional information on the syntax of the original session types.

For simplicity, we use the replications rather than the recursive agents [12] for representing infinite behaviours. We conjecture that our approach can be smoothly extended to recursive agents and recursive types. Since our typing system uses standard types, it can be easily integrated with subtyping [10], bounded session polymorphism [9] and correspondence assertions [2], guaranteeing the progress through the additional information represented by the sets of sent and bound channels and the channel relations. Challenging extensions are progress guarantees for choreographic (global) communication dependencies [5], combining more powerful means such as cryptography [3, 7], refinements [20] and logical approach [4], by which more advanced security properties can be ensured.

The main reason for including the sequencing constructor was to provide a basis for the progress straightforwardly expendable to conventional imperative and Web Service languages [5, 8, 13]. In our experience of implementations, the sequencing construct is essential in writing optimal code for the branching structures. In particular, for our ongoing work on session types with advanced exception, we require explicit sequencing to model escaping blocks during session communication and resuming an intermediate session.

Without the sequencing constructor our calculus would only be slightly simpler. We could not get rid of the evaluation contexts, since progress requires that the process which receives a live channel is evaluated in parallel with the contexts, as shown in the example at the end of Section 2. For the same reason we need a terminator for the receiving process, role which is played by sequencing in the current calculus. To sum up without the sequencing constructor we would loose expressivity with the only gain of sparing one typing rule.

We plan to extend the current formulation and typing system for preserving the progress property on live channels, and to apply it to the design of a type safe exception handling for Java with session communication [13].

Acknowledgements We thank Simon Gay, Naoki Kobayashi, Vasco Vasconcelos, the TGC referees and participants for their comments and discussions. The final version of the paper improved due to their suggestions.

References

1. J. Beaton and W. Weijiland. *Process Algebra*, volume 18 of *Cambridge Tracks in Theoretical Computer Science*. CUP, 2000.
2. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.

3. S. Briais and U. Nestmann. A Formal Semantics for Protocol Narrations. In *TGC'05*, volume 3705 of *LNCS*, pages 163–181. Springer-Verlag, 2005.
4. L. Caires. Spatial-Behavioral Types, Distributed Services, and Resources. In *TGC'06*, volume 4661 of *LNCS*, pages 263–280. Springer-Verlag, 2007.
5. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.
6. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer-Verlag, 2007.
7. R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CSF'07*, pages 170–186. IEEE Computer Society, 2007.
8. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
9. S. Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 2007. To appear.
10. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
11. S. Gay and V. T. Vasconcelos. Asynchronous Functional Session Types. TR 2007–251, Department of Computing, University of Glasgow, 2007.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
13. R. Hu, N. Yoshida, and K. Honda. Language and Runtime Implementation of Sessions for Java. In *ICOOOLPS'07*, 2007. `http://www.doc.ic.ac.uk/~rh105/sessiondj.html`.
14. A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
15. N. Kobayashi. A Partially Deadlock-Free Typed Process Calculus. *ACM TOPLAS*, 20(2):436–482, 1998.
16. N. Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177:122–159, 2002.
17. N. Kobayashi. Type Systems for Concurrent Programs. In *Formal Methods at the Crossroads*, volume 2757 of *LNCS*, pages 439–453. Springer-Verlag, 2003.
18. N. Kobayashi. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informatica*, 42(4–5):291–347, 2005.
19. N. Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer-Verlag, 2006.
20. N. Kobayashi. Type Systems for Concurrent Programs. Extended version of [**?**], Tohoku University, 2007.
21. C. Laneve and L. Padovani. The Must Preorder Revisited: An Algebraic Theory for Web Services Contracts. In *CONCUR'07*, volume 4703 of *LNCS*, pages 212–225. Springer-Verlag, 2007.
22. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
23. Web Services Choreography Working Group. Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
24. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus. *Information and Computation*, 191(2):145–202, 2004.
25. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In *SecReT'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.