A type checking algorithm for qualified session types

Marco Giunti

A type checking algorithm for qualified session types

Marco Giunti
INRIA Saclay and LIX, École Polytechnique, France

We present a type checking algorithm for establishing a session-based discipline in the pi calculus of Milner, Parrow and Walker. Our session types are qualified as linear or unrestricted. Linearly typed communication channels are guaranteed to occur in exactly one thread, possibly multiple times; afterwards they evolve as unrestricted channels. Session protocols are described by a type constructor that denotes the two ends of one and the same communication channel. We ensure the soundness of the algorithm by showing that processes consuming all linear resources are accepted by a typing system preserving typings during the computation and that type checking is consistent w.r.t. structural congruence.

1 Introduction

Session types allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol. Introduced for a dialect of the pi calculus [6, 12], the concept has been transferred to different realms, including functional and object-oriented programming and operating systems; refer to [2] for a recent overview.

To illustrate, consider the problem of designing a web system for the scheduling of meetings. In our example, the system is implemented by means of a web service repeatedly waiting for requests to create a poll. Once invoked, the service instantiates a fresh session for the poll and launches a thread for managing it. In the pi calculus [9] the session could be modeled as a communication channel for the exchange of the messages required by the scheduling protocol. The fresh channel for the poll is forwarded back to the invoker on the channel she has provided in order to receive the information needed for the start of the poll: the title and a tentative date for the meeting. Afterwards the thread repeatedly waits for possible date proposals from the participants of the poll.

$$P_1 = !x(y).(\nu p)(\overline{y}\langle p\rangle.p(\mathsf{title}).p(\mathsf{date}).!p(\mathsf{date}))$$

In order to have some guarantee on the behavior of the executable system, a static analysis of its code should be performed during the compilation. A typed analysis permits indeed to verify the desired properties of the protocol, namely that there is exactly one title and at least one date proposal for the meeting. To this aim we need to enforce that the capability forwarded to the caller consists in (i) send a string for the title and afterwards (ii) send one or more dates. This behavior could be described by relying on polymorphic types qualified as linear or unrestricted. The idea is to introduce qualifiers for types describing a session and to allow a linear usage of a session to evolve to an unrestricted usage. This approach has been indeed advocated as effective independently from any programming language [14]. A qualified session type for the poll channel sent to the invoker is the one below.

$$S_2 = \mathsf{lin}\,!\mathsf{string}.\mathsf{lin}\,!\mathsf{date}.S_4 \qquad S_4 = \mathsf{un}\,!\mathsf{date}.S_4$$

The session type first describes the sending of a string to set the title of the meeting; such usage is qualified as linear because a title for the schedule is required. Similarly, the continuation type for sending

To appear in EPTCS.

the date of the schedule is qualified as linear because a date has to be set in order to start the poll. Lastly zero or more date proposals could be send on the poll channel; this behavior is described by the unrestricted recursive type S_4. The continuation of the service P_1 is described by the type S_1 below that could be seen as the "dual" of S_2.

$$S_1 = \mathsf{lin}\,?\mathsf{string}.\mathsf{lin}\,?\mathsf{date}.S_3 \qquad S_3 = \mathsf{un}\,?\mathsf{date}.S_3.$$

The session type describes the behavior of receiving a title and one or more date proposals for the schedule. The receiving of the title and of the date proposal are both qualified as linear because this information is mandatory. Eventually, zero or more date proposals will arrive afterwards. The unbounded behavior of receiving such proposals is described by the unrestricted recursive type S_3. The usage of the poll channel is described by a type constructor (S_1, S_2) representing the concurrent behavior of the two channel ends [5]. The intuition is that in typing (the continuation of) service P_1 the type (S_1, S_2) is split into two parts: the linear output end point is used to type the delegation of one end of the session to the invoker while the linear input end point is used to type the continuation process.

While the idea of split types and contexts is clear and concise, the inherent non-determinism contained in its formulation makes a direct implementation infeasible. Algorithmic solutions for linear functional languages avoid to split the context into parts before checking a complex expression by passing the entire context as input to the first subexpression and have it return the unused portion as an output [15]. In the setting of concurrent computations, the idea is that when typing a parallel process $P \mid Q$ the set of linear identifiers used by P must be calculated in order to remove it before type checking Q. This approach, previously outlined for linear types of pi calculus in [7], has been implemented in the session system of [3] by representing each channel end with a distinct identifier.

In this paper, we propose an algorithm to check protocols described by types of the form (S_1, S_2) where each S_i is a qualified session type depicting one end of the communication. Channels could evolve from linear to unrestricted usage. Reasoning at the type level, we do implement split by forbidding the utilization of used parts of types and by a careful analysis of qualifiers. This construction permits us to show that (i) type checked processes are accepted by a typing system satisfying subject reduction and that (ii) type checking preserves structural congruence.

More in detail, type checking relies on the definition of several unambiguous patterns. The patterns for linear input and output processes do return a *marked* context. In the body of the function a recursive call to type check the continuation is launched. If an exception is not raised, this call returns in output a context. First, to ensure a subsequent linear usage to be finished within the continuation we verify the type for the variable in the context to be unrestricted. Second, to prohibit the use of the variable in the next thread we return a context with an "unusable" mark for the type of the variable. Similarly, in delegating a channel end of a session we pass to the checking function for the continuation a context with an unusable mark for the delegated type. Under replication, we do no admit to return new typings marked as unusable, which would imply consumption of a linear resource. Lastly, the algorithm succeeds if the context returned by the top-level call of the type checking function does not contain linear types.

The remainder of the paper is as follows. In Section 2 we introduce session types and pi calculus. Section 3 presents the type checking algorithm. Section 4 is devoted to establish the soundness of our approach. In the last part of the section we investigate the expressiveness of the algorithm. Some examples of the concrete execution of the algorithm are illustrated in Section 5. We conclude in Section 6 by discussing limitations and future work.

Types and Processes

q	::=	Qualifiers:	T	::=	Types:
	lin	linear		S	end point
	un	unrestricted		(S,S)	channel
p	::=	Pre Types:	P	::=	Processes:
	$?T.S$	receive		$\overline{x}\langle y\rangle.P$	output
	$!T.S$	send		$x(y).P$	input
	end	termination		$P \mid P$	composition
S	::=	End Point Types:		$(\nu x : T)P$	restriction
	$q\ p$	qualified channel		$!P$	replication
	a	type variable		$\mathbf{0}$	inaction
	$\mu a.S$	recursive type			

Rules for structural congruence

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad !P \equiv P \mid !P$$
$$(\nu x : T)P \mid Q \equiv (\nu x : T)(P \mid Q) \qquad (\nu x : T_1)(\nu y : T_2)P \equiv (\nu y : T_2)(\nu x : T_1)P$$
$$(\nu x : \mathsf{un}\, p)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x : (\mathsf{un}\, p_1, \mathsf{un}\, p_2))\mathbf{0} \equiv \mathbf{0}$$

Rules for reduction

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

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

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

Figure 1: Pi calculus

2 Pi calculus

This section introduces the syntax and the semantics of the typed pi calculus. The definition is in Figure 1. We consider channel types of the form (S,S) where S is a type describing the behavior of a channel end point. An end point type S can be a pre type qualified with lin or un, a recursive type or a type variable. Each qualifier in a type controls the number of times the channel can be used at that point: exactly once for lin; zero or more times for un. A pre type of the form $!T.S$ describes a channel end able to send a variable of type T and to proceed as prescribed by S. Similarly, pre type $?T.S$ describes a channel end able to receive a variable of type T and continue as S. Pre type end describes a channel end on which no further interaction is possible. For recursive (end point) types we rely on a set of type variables, ranged over by a. Recursive types are required to be contractive, that is, containing no subexpression of the form $\mu a_1 \ldots \mu a_n.a_1$. *Type equality* is not syntactic. Instead, we define it as the equality of regular infinite trees obtained by the infinite unfolding of recursive types, *modulo pair commutation*. The formal definition, which we omit, is co-inductive. In this way we use types $(\mu a.\mathsf{lin}!\mathsf{un}\,\mathsf{end}.\mathsf{lin}?\mathsf{un}\,\mathsf{end}.a, \mathsf{un}\,\mathsf{end})$

and $(\mathsf{un}\,\mathsf{end}, \mathsf{lin}!\mathsf{un}\,\mathsf{end}.\mu b.\mathsf{lin}?\mathsf{un}\,\mathsf{end}.\mathsf{lin}!\mathsf{un}\,\mathsf{end}.b)$ interchangeably, in any mathematical context. This allows us never to consider a type $\mu a.S$ explicitly (or a for that matter). Instead, we pick another type in the same equivalence class, namely $S[\mu a.S/a]$. If the result of the process turns out to start with a μ, we repeat the procedure. Unfolding is bound to terminate due to contractiveness. In other words, we take an equi-recursive view of types [10].

The syntax and the semantics of pi calculus processes are those of [9] but for restriction, for which we require type annotation. This is only to facilitate type checking and has no impact on the semantics. We rely on a set of variables, ranged over by x, y, z. For processes we have (synchronous, unary) output and input, in the forms $\overline{x}\langle y\rangle.P$ and $x(y).P$, as well as a parallel composition, annotated scope restriction, replication and the terminated process. The binders for the language appear in parenthesis: y is bound in both $x(y).P$ and $(\nu y : T)P$. Free and bound variables in processes are defined accordingly, and so is alpha conversion, substitution of a variable x by a variable z in a process P, denoted $P[z/x]$. We follow Barendregt's variable convention, requiring bound variables to be distinct from each other and from free variables in any mathematical context.

Structural congruence is the smallest relation on processes including the rules in Figure 1. The first three rules say that parallel composition is commutative, associative and has **0** as neutral element. The last rule on the first line captures the essence of replication as an unbounded number of identical processes. The rules in the second and third line deal with scope restriction. The first, scope extrusion, allows the scope of x to encompass Q; due to variable convention, x bound in $(\nu x : T)P$, cannot be free in Q. The next rule allows exchanging the order of restrictions. The rules on the third line state that restricting over a terminated process has no effect. Since it makes poor sense to declare a new variable with a linear type for a terminated process, we require the type annotation to be unrestricted. The reduction is the smallest relation on processes including the rules in Figure 1. The [R-COM] rule communicates a variable z from an output prefixed one $\overline{x}\langle z\rangle.P$ to an input prefixed process $x(y).Q$; the result is the parallel composition of the continuation processes, where the bound variable y is replaced by the variable z in the input process. The rules on the last line allow reduction to happen underneath scope restriction and parallel composition, and incorporate structural congruence into reduction.

3 Type checking algorithm

In this section we present an algorithm for type checking a pi calculus process given a typing context. Type checking relies on the definition of several patterns which, for the sake of clarity, we present in a declarative style. Lastly, in Figure 2 we present an excerpt of the *ML* implementation.

Contexts. We let Γ be a map from variables to types and the *void* symbol, noted $\circ$; a void symbol permits to mark an end point as unusable.

$$\begin{array}{rcll} M, N, O & ::= & S \mid \circ & \text{entry} \\ \Gamma & ::= & \emptyset \mid \Gamma, x : M \mid \Gamma, x : (M, N) & \text{context} \end{array}$$

Context updating, noted $\uplus$, is the procedure effected by the typing system to transform a void entry in an end point entry: $\Gamma, x : \circ \uplus x : M = \Gamma, x : M$. A *safe* context is a map from variable to safe entries; we let the predicate $\mathsf{safe}(\Gamma)$ hold whenever $x \in \mathrm{dom}(\Gamma)$ implies $\mathsf{safe}(\Gamma(x))$. A linear channel type is safe if (i) the type of the variable sent in output corresponds to the type expected in input and (ii) the expected type for the input is safe and (iii) the continuation is safe. For an unrestricted channel type we require (i) and

(ii): (iii) will be enforced by the type system.

$$\begin{array}{ll}
\mathsf{safe}(M) & \\
\mathsf{safe}((M_1,M_2)) & \exists i \in \{1,2\}.\ M_i = \circ, \mathsf{un\,end} \\
\mathsf{safe}((\mathsf{lin}\,?T.S_1, \mathsf{lin}\,!T.S_2)) = \mathsf{safe}(T) \wedge \mathsf{safe}((S_1,S_2)) & \\
\mathsf{safe}((\mathsf{un}\,?T.S_1, \mathsf{un}\,!T.S_2)) = \mathsf{safe}(T) &
\end{array}$$

A context is *unrestricted* if it contains only unrestricted or void entries. We let $\mathsf{un}(\Gamma)$ whenever $x \in \mathrm{dom}(\Gamma)$ implies $\mathsf{un}(\Gamma(x))$.

$$\begin{array}{l}
\mathsf{un}(\circ) \\
\mathsf{un}(\mathsf{un}\,p) \\
\mathsf{un}((M,N)) = \mathsf{un}(M) \wedge \mathsf{un}(N)
\end{array}$$

Patterns. We present typing rules for processes of the form $\Gamma_1 \vdash P \triangleright \Gamma_2$ where Γ_1 is a context received in input and Γ_2 is a context produced as output. Given that Γ_1 is a context such that $\mathsf{safe}(\Gamma_1)$, the rules are chosen deterministically by inspecting (i) the shape of the context and (ii) the shape of the process, in the following way. Each rule is implemented as a pattern of a function with signature $\mathsf{check}(\mathsf{g} : \mathsf{context}, \mathsf{p} : \mathsf{process}) : \mathsf{context}$. For each function call with a safe context parameter, zero or one pattern does match; in the first case a pattern exception indicating the reject of the process is raised while in the second case a context is returned in output to the caller. The rules for variables have the form $\Gamma_1 \vdash v : T \triangleright \Gamma_2$ and are implemented as patterns of a function with signature checkVar (g:context, v:var):context . In the rules below the output context is obtained by setting to void the linear assumptions used to type the variable. The last three rules permit to resolve any ambiguity in typing an unrestricted end point type with an unrestricted channel type.

$$\frac{\Gamma = \Gamma_1, x\colon \mathsf{lin}\,p, \Gamma_2}{\Gamma \vdash x\colon \mathsf{lin}\,p \triangleright \Gamma_1, x\colon \circ, \Gamma_2} \qquad \frac{\Gamma = \Gamma_1, x\colon \mathsf{un}\,p, \Gamma_2}{\Gamma \vdash x\colon \mathsf{un}\,p \triangleright \Gamma} \qquad \text{[A-V-L],[A-V-U]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{lin}\,p_1, \mathsf{lin}\,p_2), \Gamma_2}{\Gamma \vdash x\colon (\mathsf{lin}\,p_1, \mathsf{lin}\,p_2) \triangleright \Gamma_1, x\colon (\circ,\circ), \Gamma_2} \qquad \text{[A-V-LL-L]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{lin}\,p_1, \mathsf{lin}\,p_2), \Gamma_2}{\Gamma \vdash x\colon (\mathsf{lin}\,p_2, \mathsf{lin}\,p_1) \triangleright \Gamma_1, x\colon (\circ,\circ), \Gamma_2} \qquad \text{[A-V-LL-R]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{lin}\,p, N), \Gamma_2}{\Gamma \vdash x\colon \mathsf{lin}\,p \triangleright \Gamma_1, x\colon (\circ, N), \Gamma_2} \qquad \frac{\Gamma = \Gamma_1, x\colon (M, \mathsf{lin}\,p), \Gamma_2}{\Gamma \vdash x\colon \mathsf{lin}\,p \triangleright \Gamma_1, x\colon (M, \circ), \Gamma_2} \qquad \text{[A-V-L-R],[A-V-L-L]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{un}\,p_1, \mathsf{un}\,p_2), \Gamma_2}{\Gamma \vdash x\colon (\mathsf{un}\,p_1, \mathsf{un}\,p_2) \triangleright \Gamma} \qquad \frac{\Gamma = \Gamma_1, x\colon (\mathsf{un}\,p_1, \mathsf{un}\,p_2), \Gamma_2}{\Gamma \vdash x\colon (\mathsf{un}\,p_2, \mathsf{un}\,p_1) \triangleright \Gamma} \qquad \text{[A-V-UU-L],[A-V-UU-R]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{un}\,p, N), \Gamma_2 \quad \mathsf{un}\,p \neq N}{\Gamma \vdash x\colon \mathsf{un}\,p \triangleright \Gamma} \qquad \frac{\Gamma = \Gamma_1, x\colon (M, \mathsf{un}\,p), \Gamma_2 \quad \mathsf{un}\,p \neq M}{\Gamma \vdash x\colon \mathsf{un}\,p \triangleright \Gamma} \qquad \text{[A-V-U-L],[A-V-U-R]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (\mathsf{un\,end}, \mathsf{un\,end}), \Gamma_2}{\Gamma \vdash x\colon \mathsf{un\,end} \triangleright \Gamma} \qquad \text{[A-V-EE]}$$

Rule [A-OUT-L] is to type processes sending variables on a channel used in linear mode given that the type for the channel in the context is an end point. The context changed by setting the channel to void is

used to check the sent variable at the expected type and in turn to return a new context. The new context updated with the continuation type for the linear channel is passed as parameter in the call for checking the continuation process. To ensure a linear use of the channel to be finished within the continuation, we verify that the context returned by the call for the continuation does contain an unrestricted typing for the channel. Finally, the returned context is given as output with the typing for the channel set to void.

$$\frac{\Gamma_1, x:\circ \vdash y\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \uplus x\colon S \vdash P \triangleright \Gamma_3, x:M \qquad \mathsf{un}(M)}{\Gamma_1, x\colon \mathsf{lin}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x:\circ} \quad \text{[A-OUT-L]}$$

Rules [A-OUT-L-L],[A-OUT-L-R] are used when the entry for the linear output in the context is a channel type. The rules are implemented by the pattern [A-OUT-L]. In returning the context we set one end of the channel type to void while we leave the other end as it has been received in input.

$$\frac{\Gamma_1, x\colon \mathsf{lin}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x:\circ}{\Gamma_1, x\colon (\mathsf{lin}\,!T.S, N) \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x:(\circ, N)} \quad \text{[A-OUT-L-L]}$$

$$\frac{\Gamma_1, x\colon \mathsf{lin}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x:\circ}{\Gamma_1, x\colon (M, \mathsf{lin}\,!T.S) \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x:(M, \circ)} \quad \text{[A-OUT-L-R]}$$

For sending a variable on an unrestricted channel we require the sent variable to be typable by the same context received in input; that is, the type for the unrestricted output channel must be recursive. The context obtained by the typing for the variable is then used to call the checking function for the continuation process.

$$\frac{\Gamma_1, x:S \vdash v\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \vdash P \triangleright \Gamma_3 \qquad S = \mathsf{un}\,!T.S}{\Gamma_1, x\colon \mathsf{un}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3} \quad \text{[A-OUT-UN]}$$

$$\frac{\Gamma_1, x\colon (S,N) \vdash v\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \vdash P \triangleright \Gamma_3 \qquad S = \mathsf{un}\,!T.S}{\Gamma_1, x\colon (\mathsf{un}\,!T.S, N) \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3} \quad \text{[A-OUT-UN-L]}$$

$$\frac{\Gamma_1, x\colon (M,S) \vdash v\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \vdash P \triangleright \Gamma_3 \qquad S = \mathsf{un}\,!T.S}{\Gamma_1, x\colon (M, \mathsf{un}\,!T.S) \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3} \quad \text{[A-OUT-UN-R]}$$

To type a linear usage of an input we require the expected type to agree with the type of the input channel and the continuation type for the channel to be consumed within the continuation. This is implemented by requiring that the context returned by the call for the continuation does map the variable to an unrestricted type. We also require a linear usage for the variable bound by the input to be finished within its local scope. Lastly, the call returns a context (i) with the type for the linear variable set to void and (ii) pruned by the variable bound by the input prefix. This is the rationale of rule [A-IN-L] and or rules [A-IN-L-L],[A-IN-L-R] which are used for variables having respectively an end point or a channel type.

$$\frac{\Gamma_1, x\colon S, y\colon T \vdash P \triangleright \Gamma_2, x : M, y : O \qquad \mathsf{un}(M) \qquad \mathsf{un}(O)}{\Gamma_1, x\colon \mathsf{lin}?T.S \vdash x(y).P \triangleright \Gamma_2, x : \circ} \qquad \text{[A-IN-L]}$$

$$\frac{\Gamma_1, x\colon \mathsf{lin}?T.S \vdash x(y).P \triangleright \Gamma_2, x : \circ}{\Gamma_1, x\colon (\mathsf{lin}?T.S, N) \vdash x(y).P \triangleright \Gamma_2, x : (\circ, N)} \qquad \text{[A-IN-L-L]}$$

$$\frac{\Gamma_1, x\colon \mathsf{lin}?T.S \vdash x(y).P \triangleright \Gamma_2, x : \circ}{\Gamma_1, x\colon (M, \mathsf{lin}?T.S) \vdash x(y).P \triangleright \Gamma_2, x : (M, \circ)} \qquad \text{[A-IN-L-R]}$$

The rules for unrestricted input take the context received in input and add the bound variable at the expected type in order to type the continuation. The context returned by the call of the checking function for the continuation needs to be first verified to ensure that the type for the bound variable is unrestricted, and then pruned by the variable to be returned in output.

$$\frac{\Gamma_1, x\colon S, y\colon T \vdash P \triangleright \Gamma_2, y : O \qquad \mathsf{un}(O) \qquad S = \mathsf{un}?T.S}{\Gamma_1, x\colon \mathsf{un}?T.S \vdash x(y).P \triangleright \Gamma_2} \qquad \text{[A-IN-UN]}$$

$$\frac{\Gamma_1, x\colon (S, N), y\colon T \vdash P \triangleright \Gamma_2, y : O \qquad \mathsf{un}(O) \qquad S = \mathsf{un}?T.S}{\Gamma_1, x\colon (\mathsf{un}?T.S, N) \vdash x(y).P \triangleright \Gamma_2} \qquad \text{[A-IN-UN-L]}$$

$$\frac{\Gamma_1, x\colon (M, S), y\colon T \vdash P \triangleright \Gamma_2, y\colon O \qquad \mathsf{un}(O) \qquad S = \mathsf{un}?T.S}{\Gamma_1, x\colon (M, \mathsf{un}?T.S) \vdash x(y).P \triangleright \Gamma_2} \qquad \text{[A-IN-UN-R]}$$

To type an inert process by using [A-INACT] any context suffices; the context received in input is forwarded in output. To type a parallel process in [A-PAR] we check the first thread with the context received in input. This operation returns in output a context that is used to type-check the next thread. The context returned by the last typing is forwarded in output. While imposing an order on parallel processes could appear restrictive, in Section 4 we will show that the chosen order makes no difference.

$$\Gamma \vdash \mathbf{0} \triangleright \Gamma \qquad \frac{\Gamma_1 \vdash P \triangleright \Gamma_2 \qquad \Gamma_2 \vdash Q \triangleright \Gamma_3}{\Gamma_1 \vdash P \mid Q \triangleright \Gamma_3} \qquad \text{[A-INACT],[A-PAR]}$$

In order to type a process generating a new channel, in rule [A-RES] we require the typing for the channel to be safe; if it is not, the algorithm stops and an exception is raised. Similarly to the input cases, if a linear usage is prescript for the new variable then it must be finished within its scope.

$$\frac{\mathsf{safe}(T) \quad \Gamma_1, y\colon T \vdash P \triangleright \Gamma_2, y : O \quad \mathsf{un}(O)}{\Gamma_1 \vdash (\nu y\colon T) P \triangleright \Gamma_2} \qquad \text{[A-RES]}$$

The rule for replication [A- REPL] is below. In the call for checking the process under the replication we require the context returned in output to be equal to the one received in input. Indeed, a change in the output context would be obtained by introducing a void symbol indicating that a linear resource has been consumed. This must clearly be forbidden under replication. On contrast, we allow to return linear entries in order to type check the next thread.

$$\frac{\Gamma_1 \vdash P \triangleright \Gamma_2 \quad \Gamma_2 = \Gamma_1}{\Gamma_1 \vdash\, !P \triangleright \Gamma_2} \qquad \text{[A- REPL]}$$

```
datatype qualifier = Lin | Un;
datatype preType   = In of sessionType * endpointType | Out of sessionType * endpointType | End
and   endpointType = Qualified of qualifier * preType | Void
and   sessionType  = EndPoint of endpointType | Channel of endpointType * endpointType;
type context       = (string * sessionType) list;
datatype process   = Zero | Replication of process | Parallel of process * process
                   | Input of string * string * process  | Output of string *  string * process
                   | New of string * sessionType * process;
fun safe           (g:context):context;
fun unVar          (g:context,v:string):context;
fun remove         (g:context,v:string):context;
fun setVoid        (g:context,v:string):context;
fun checkVar       ((x, ((EndPoint (Qualified (Lin,p) ))))::g,
                    (z, ((EndPoint (Qualified (Lin,r) ))))) =
                      if p=r then (x, (EndPoint Void))::g                              (* A-V-L *)
 |  checkVar       ((x, ((EndPoint (Qualified (Un,p) ))))::g,
                    (z, ((EndPoint (Qualified (Un,r) ))))) =
                      if p=r then ((x, ((EndPoint  ( Qualified (Un,p) ))))::g)      (* A-V-U *)
 |  checkVar       ((x, ((Channel  (Qualified (Lin,p), Qualified (Lin,s) ))))::g,
                    (z, ((Channel  (Qualified (Lin,r), Qualified (Lin,t) )))))=
                      if p=r andalso s=t then
                        ((x, ((Channel  ( Void, Void ))))::g)
                      else
                        if p=t andalso s=r
                          then ((x, ((Channel  ( Void, Void ))))::g)           (* A-V-LL-L+R *)
 |  checkVar       ((x, ((Channel  (Qualified (Lin,p), Void ))))::g,
                    (z, ((EndPoint (Qualified (Lin,r)))))) =
                      if p=r
                          then  ((x, ((Channel ( Void,Void ))))::g)               (* A-V-L-R *)
 |  checkVar       ((x, ((Channel  (Qualified (Lin,p), Qualified (Un,s) ))))::g,
                    (z, ((EndPoint (Qualified (Lin,r)))))) =
                      if p=r
                          then  ((x, ((Channel ( Void, Qualified (Un,s)))))::g)   (*A-V-L-R  *)
(*                | A-V-L-L   | A-V-UU-L  | A-V-U-L  | A-V-U-R    | A-V-E-E                  *);
fun check          (g:context,Zero:process)=
                      g                                                            (* A-INACT *)
 |  check          (g:context,Replication p)=
                      if g = check(g,p)
                         then g                                                    (* A-REPL  *)
 |  check          (g:context, Parallel (p1,p2))  =
                      check ( check (g,p1) , p2 )                                  (* A-PAR   *)
 |  check          ((z,(Endpoint (Qualified (Lin,In (a,c)) )))::t, Input (x,y,p))=
                      let val d = check ((x,Endpoint (c))::t,p) in
                        setVoid ( remove (unVar (unVar  (d,x), y), y) , x )
                      end                                                          (* A-IN-L  *)
  | check          (g:context,New (x,t,p) )  =
                      remove ( unVar ( check ( ( safe([(x,t)]))@g, p) , x ) , x)  (* A-RES   *)
(*                | A-IN-L-l  | A-IN-L-r  | A-IN-Un  | A-IN-Un-l  | A-IN-Un-r | A-OUT-L
                  | A-OUT-L-l | A-OUT-L-r | A-OUT-Un | A-OUT-Un-l | A-OUT-Un-r               *);
fun typeCheck      (g:context,p:process)  =
                      un ( check ( safe(g) , p ) );
```

Figure 2: ML code of the algorithm (excerpt)

Lemma 3.1. *If* $\mathsf{safe}(\Gamma_1)$ *and* $\Gamma_1 \vdash P \triangleright \Gamma_2$ *then* $\mathrm{dom}(\Gamma_2) = \mathrm{dom}(\Gamma_1)$ *and* $\mathsf{safe}(\Gamma_2)$.

Type checking. Having defined typing rules corresponding to patterns of the checking function, we devise an algorithm for establish a session-based type discipline. Figure 2 presents the ML definition for types, processes and the type checking function. Type context associates variables to entries which are formed apart the end point and the channel type. The function safe returns in output the same context received in input whenever the context satisfies the safe predicate, otherwise it generates an exception. Function unVar takes as parameters a context and a variable and verifies that the type for the variable in the context is unrestricted; in this case the context is returned in output, otherwise an exception is raised. Functions remove and setVoid do perform the required operations and return the updated context. We also need auxiliary functions to push and pop entries to and from the context stack; we omit all the details.

The check function, the kernel of the type checking procedure, is defined by the union of the patterns for the rules introduced in the current section. In order to illustrate the mechanism, we draw the translation of some patterns. In patterns for variables and in [A-IN-L] we assume the variable on the top of the context *z* to be equal to the variable *x* respectively for the value to type and for the input prefix of the process. The checkVar function is called in patterns for output in order to type the sent variable and obtain in output a context to pass together with the the continuation to the checking function. In [A-IN-L] we launch the recursive call of the check function by passing as parameters the updated context and the continuation process. After checking that the type for both channel *x* and the variable bound by the input are unrestricted in the returned context, we return the context with the type for *x* set to void. In the pattern for [A-RES] we launch the check function by passing as parameters the context with the new entry and the continuation process. The inner call of the safe function immediately raises an exception if the type for the bound variable is not safe. Lastly, we first control that in the returned context the variable is unrestricted and then we return the context pruned by the variable. The algorithm is implemented by the typeCheck function. The function receives in input a context and a process. If the context received in input is not safe then the function exits immediately. Otherwise, a context is returned in input provided that an exception has not been raised. The exception could raise (i) when no pattern matching is possible for the chosen derivation or (ii) when a call of the safe function in [A-RES] fails or (iii) when call of unVar function fails. Since the choice of patterns is deterministic for safe contexts, no backtracking is needed. Lastly, the process is accepted by the algorithm whenever the returned context satisfies the un predicate defined in Section 2.

Lemma 3.2. *If* $\mathsf{safe}(\Gamma)$ *then* $\mathsf{check}(\Gamma, P)$ *matches zero or one patterns.*

Lemma 3.3. *If* $\mathsf{check}(\Gamma', P')$ *has been recursively invoked by* $\mathsf{typeCheck}(\Gamma, P)$ *then we have* $\mathsf{safe}(\Gamma')$.

Proof. A call is a match of a pattern $\Gamma_1 \vdash P \triangleright \Gamma_2$ which is an axiom whenever $P = \mathbf{0}$, and has been inferred from an hypothesis starting with a type environment Δ on the left otherwise. We proceed by induction and show a stronger result, namely that $\mathsf{safe}(\Delta)$ implies $\mathsf{safe}(\Gamma_1)$. We close the proof by applying Lemma 3.1, and eventually by exploiting transitivity in cases for output and parallel composition. □

Corollary 3.4. *If* $\mathsf{check}(\Gamma', P')$ *is a call invoked during the execution of* $\mathsf{typeCheck}(\Gamma, P)$ *then there are zero or one patterns to match.*

Context splitting rules

$$\emptyset = \emptyset \cdot \emptyset \qquad \frac{I = I_1 \cdot I_2 \qquad T = \mathsf{un}\, p \text{ or } (\mathsf{un}\, p_1, \mathsf{un}\, p_2)}{I, x\colon T = (I_1, x\colon T) \cdot (I_2, x\colon T)}$$

$$\frac{I = I_1 \cdot I_2 \qquad T = \mathsf{lin}\, p \text{ or } (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)}{I, x\colon T = (I_1, x\colon T) \cdot I_2} \qquad \frac{I = I_1 \cdot I_2 \qquad T = \mathsf{lin}\, p \text{ or } (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)}{I, x\colon T = I_1 \cdot (I_2, x\colon T)}$$

$$\frac{I = I_1 \cdot I_2}{I, x\colon (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2) = (I_1, x\colon \mathsf{lin}\, p_1) \cdot (I_2, x\colon \mathsf{lin}\, p_2)}$$

$$\frac{I = I_1 \cdot I_2}{I, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2) = (I_1, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2)) \cdot (I_2, x\colon \mathsf{un}\, p_2)}$$

$$\frac{I = I_1 \cdot I_2}{I, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2) = (I_1, x\colon \mathsf{un}\, p_2) \cdot (I_2, x\colon (\mathsf{lin}\, p_1, \mathsf{un}\, p_2))}$$

Typing rules for values

$$\frac{\mathsf{un}(I)}{I, x\colon T \vdash_D x\colon T} \qquad \frac{I \vdash_D v\colon (S, \mathsf{un}\, p)}{I \vdash_D v\colon S} \qquad \text{[T-Var] [T-Strength]}$$

Typing rules for processes

$$\frac{\mathsf{un}(I)}{I \vdash_D \mathbf{0}} \qquad \frac{I_1 \vdash_D R_1 \qquad I_2 \vdash_D R_2}{I_1 \cdot I_2 \vdash_D R_1 \mid R_2} \qquad \text{[T-Inact] [T-Par]}$$

$$\frac{I \vdash_D R \qquad \mathsf{un}(I)}{I \vdash_D !R} \qquad \frac{I, x\colon T \vdash_D R \qquad \mathsf{safe}(T)}{I \vdash_D (\nu x\colon T) R} \qquad \text{[T-Repl] [T-Res]}$$

$$\frac{I, x\colon S, y\colon T \vdash_D R \qquad (*)}{I, x\colon q?T.S \vdash_D x(y).R} \qquad \frac{I_1 \vdash_D v\colon T \qquad I_2, x\colon S \vdash_D R \qquad (**)}{I_1 \cdot (I_2, x\colon q!T.S) \vdash_D \overline{x}\langle v \rangle.R} \qquad \text{[T-In],[T-Out]}$$

$$\frac{I, x\colon (S, S'), y\colon T \vdash_D R \qquad (*)}{I, x\colon (q?T.S, S') \vdash_D x(y).R} \qquad \frac{I_1 \vdash_D v\colon T \qquad I_2, x\colon (S, S') \vdash_D R \qquad (**)}{I_1 \cdot (I_2, x\colon (q!T.S, S')) \vdash_D \overline{x}\langle v \rangle.R} \qquad \text{[T-InC],[T-OutC]}$$

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

Figure 3: Split-based typing system

4 Soundness

This section is devoted to establishing the soundness of the algorithm. To this aim we project the pattern rules presented in Section 3 into the typing system of Figure 3, which satisfies subject reduction [5]. The syntax of types and processes occurring in Figure 3 is that of Figure 1. Contexts I are a map from variables to types T:

$$I ::= \emptyset \mid I, x\colon T\ .$$

Typing rules in Figure 3 are based on a declarative definition of context splitting; the intuition is that unrestricted types are copied into both contexts, while linear types are placed in one of the two resulting contexts. We refer to [5] for the details.

We introduce preliminary Lemmas and Definitions which will be useful to prove the main result of this section. Given a judgment $\Gamma_1 \vdash P \triangleright \Gamma_2$ of the algorithmic system of Section 3 , we let the used closure of a type context Γ_1 w.r.t. Γ_2, noted $\Gamma_1 \triangleright \Gamma_2$, be the typing context $\emptyset$ whenever $\Gamma_1 = \emptyset$, and be defined by

$(\Gamma_1 \triangleright \Gamma_2)(x) = \Gamma_1(x) \triangleright \Gamma_2(x)$ otherwise:

$$\circ \triangleright \circ = \circ \qquad \mathsf{lin}\, p_1 \triangleright \mathsf{lin}\, p_1 = \circ$$
$$\mathsf{lin}\, p_1 \triangleright \circ = \mathsf{lin}\, p_1 \qquad \mathsf{un}\, p_1 \triangleright \mathsf{un}\, p_1 = \mathsf{un}\, p_1$$
$$(M,N) \triangleright (M',N') = (M \triangleright M', N \triangleright N')\ .$$

The map operation projects a type environment Γ into a context I of Figure 3. When applied to a used closure, it permits to map linear typings which do not change from Γ_1 to Γ_2 into the $\mathsf{un}\,\mathsf{end}$ type.

$$\mathsf{map}(\circ) = \mathsf{un}\,\mathsf{end} \qquad \mathsf{map}(S) = S$$
$$\mathsf{map}((M,N)) = (\mathsf{map}(M), \mathsf{map}(N)) \qquad \mathsf{map}(\Gamma) = \bigcup_{x \in \mathrm{dom}(\Gamma)} x : \mathsf{map}(\Gamma(x))$$

Lemma 4.1. *Assume* $\mathsf{safe}(\Gamma_1)$. *If* $\Gamma_1 \vdash P \triangleright \Gamma_2$ *then* $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)$ *is defined.*

A used closure generated by the algorithmic system is sufficient to type a process with the system $\vdash_D$, as we will show in a nontrivial manner below. We need a couple of lemmas for strengthening judgments of the algorithmic system and weaken judgments of the split-based system.

Lemma 4.2 (Algorithmic strengthening)**.** *The following hold.*

1. *If* $\Gamma_1, x : \mathsf{lin} p \vdash P \triangleright \Gamma_2, : x : \mathsf{lin} p$ *then* $\Gamma_1 \vdash P \triangleright \Gamma_2$*;*
2. *If* $\Gamma_1, x : (\mathsf{lin} p, S) \vdash P \triangleright \Gamma_2, x : (\mathsf{lin} p, N)$ *then* $\Gamma_1, x : S \vdash P \triangleright \Gamma_2, x : N'$*;*
3. *If* $\Gamma_1, x : (M, \mathsf{lin} p) \vdash P \triangleright \Gamma_2, x : (M', \mathsf{lin} p)$ *then* $\Gamma_1, x : M \vdash P \triangleright \Gamma_2, x : M'$*;*
4. *If* $\Gamma_1, x : \circ \vdash P \triangleright \Gamma_2, : x : \circ$ *then* $\Gamma_1 \vdash P \triangleright \Gamma_2$*;*
5. *If* $\Gamma_1, x : (\circ, N) \vdash P \triangleright \Gamma_2, : x : (\circ, N')$ *then* $\Gamma_1, x : N \vdash P \triangleright \Gamma_2, x : N'$*;*
6. *If* $\Gamma_1, x : (M, \circ) \vdash P \triangleright \Gamma_2, x : (M', \circ)$ *then* $\Gamma_1, x : M \vdash P \triangleright \Gamma_2, x : M'$*;*
7. *If* $\Gamma_1, x : \mathsf{un}\, p \vdash P \triangleright \Gamma_2, x : \mathsf{un}\, p$ *and* $x \notin \mathrm{fv}(P)$ *then* $\Gamma_1 \vdash P \triangleright \Gamma_2$*;*
8. *If* $\Gamma_1, x : (\mathsf{un}\, p_1, \mathsf{un}\, p_2) \vdash P \triangleright \Gamma_2, x : (\mathsf{un}\, p_1, \mathsf{un}\, p_2)$ *and* $x \notin \mathrm{fv}(P)$ *then* $\Gamma_1 \vdash P \triangleright \Gamma_2$.

Lemma 4.3 (Weakening)**.** $I, x : S \vdash_D P$ *implies* $I, x : (S, \mathsf{un}\, p) \vdash_D P$.

We have all the ingredients to prove the following result which is the wedge of the proof of soundness.

Lemma 4.4. *Assume* $\mathsf{safe}(\Gamma_1)$. *The following hold.*

1. *If* $\Gamma_1 \vdash v\colon T \triangleright \Gamma_2$ *then* $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2) \vdash_D v\colon T$*;*
2. *If* $\Gamma_1 \vdash P \triangleright \Gamma_2$ *then* $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2) \vdash_D P$.

Proof. We first prove (1). Assume $\Gamma_1, x\colon \mathsf{lin}\, p, \Gamma_2 \vdash x\colon \mathsf{lin}\, p \triangleright \Gamma_1, x\colon \circ, \Gamma_2$. Notice that $I = \mathsf{map}((\Gamma_1, \Gamma_2) \triangleright (\Gamma_1, \Gamma_2))$ is a safe type context such that $\mathsf{un}(I)$, i.e. it contains only unrestricted typings, and that $\mathsf{lin}\, p \triangleright \circ = \mathsf{lin}\, p$. We apply [T-VAR] and infer $I, x : \mathsf{lin}\, p \vdash_D x : \mathsf{lin}\, p$. The cases for typing a linear or unrestricted channel type, or an unrestricted channel type are analogous. Assume $\Gamma_1, x\colon (\mathsf{lin}\, p, N), \Gamma_2 \vdash x\colon \mathsf{lin}\, p \triangleright \Gamma_1, x\colon (\circ, N), \Gamma_2$. Let $I = \mathsf{map}((\Gamma_1, \Gamma_2) \triangleright (\Gamma_1, \Gamma_2))$. We have $I(x) = (\mathsf{lin}\, p, S)$ with $S = \mathsf{un}\, p'$ or $S = \mathsf{un}\,\mathsf{end}$. From these results and [T-VAR] we infer $I, x : (\mathsf{lin}\, p, S) \vdash x : \mathsf{lin}\, p$. Now assume that $\Gamma \vdash_D x\colon \mathsf{un}\, p \triangleright \Gamma$ with $\Gamma = \Gamma_1, x\colon (\mathsf{un}\, p, N), \Gamma_2$. From $\mathsf{un}(\mathsf{map}(\Gamma \triangleright \Gamma))$ and [T-VAR] we infer that there is $S = \mathsf{un}\, p'$ or $S = \mathsf{un}\,\mathsf{end}$ such that $\mathsf{un}(\mathsf{map}(\Gamma \triangleright \Gamma)) \vdash_D x\colon (\mathsf{un}\, p, S)$. We apply [T-STRENGTH] and infer the desired result: $\mathsf{map}(\Gamma \triangleright \Gamma) \vdash_D x\colon \mathsf{un}\, p$.

To prove (2) we proceed by induction on the length of the derivation for $\Gamma_1 \vdash P \triangleright \Gamma_2$. We prove the most interesting cases. We use the notation $\Gamma \backslash x$ to indicate the context Γ' whenever $\Gamma = \Gamma', x : M$ or $\Gamma = \Gamma', x : (M, N)$.

[A-PAR] We have $\Gamma_1 \vdash P \mid Q \triangleright \Gamma_3$ inferred from $\Gamma_1 \vdash P \triangleright \Gamma_2$ and $\Gamma_2 \vdash Q \triangleright \Gamma_3$. We proceed by case analysis on Γ_1. If $\Gamma_1 = \emptyset$ we are done by applying the first rule for context splitting. Otherwise assume $\Gamma_1 = \Gamma, x : T$. We exploit Lemma 4.1 in order to infer the type of $\Gamma_2(x)$ and $\Gamma_3(x)$.

($T = \circ$). We have $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)(x) = \mathsf{un}\,\mathsf{end}$ and $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3)(x) = \mathsf{un}\,\mathsf{end}$. We apply [T-PAR] to the I.H by using the second rule for context splitting to $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2) \vdash_D P$ and $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3) \vdash_D Q$.

($T = (\circ, \circ)$). Analogous to the previous case.

($T = \mathsf{lin}\, p$). We have two cases for $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)(x)$ corresponding to (i) $\mathsf{lin}\, p$ and (ii) $\mathsf{un}\,\mathsf{end}$. In case (i) we have $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3) = \mathsf{un}\, p'$. This is because by definition of map we have that $\Gamma_2(x) = \circ$. By applying Lemma 4.3 we weaken $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)$ and obtain an environment Δ equal to $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)$ but for the entry x which is weakened to $(\mathsf{lin}\, p, \mathsf{un}\, p')$. We apply the I.H. and infer the desired result by applying the fifth rule for context splitting in [T-PAR]: $\Delta \vdash_D P$ and $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3) \vdash_D Q$. In case (ii) we have $\Gamma_2(x) = \mathsf{lin} p$. In sub case $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3)(x) = \mathsf{lin}\, p$ we apply the I.H. and proceed by weakening the type to $(\mathsf{un}\,\mathsf{end}, \mathsf{lin}\, p)$ in order to apply the sixth rule for splitting in [T-PAR]. In sub-case $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3)(x) = \mathsf{un}\,\mathsf{end}$ we apply the second splitting rule.

($T = (\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)$). We have four cases for $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2)(x)$ corresponding to (iii) $(\mathsf{lin}\, p_1, \mathsf{lin}\, p_2)$ and (iv) $(\mathsf{lin}\, p_1, \mathsf{un}\,\mathsf{end})$ and (v) $(\mathsf{un}\,\mathsf{end}, \mathsf{lin}\, p_2)$ and (vi) $(\mathsf{un}\,\mathsf{end}, \mathsf{un}\,\mathsf{end})$. In case (iii) we infer $\Gamma_2(x) = (\circ, \circ)$. We apply Lemma 4.2 and strengthen the algorithm's judgment by removing the entry for x in Γ_2: $\Gamma_2 \backslash x \vdash Q \triangleright \Gamma_3 \backslash x$. We apply the I.H. and by [T-PAR] we infer the desired result by applying the fourth rule for context splitting to $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2) \vdash_D P$ and $\mathsf{map}(\Gamma_2 \backslash x \triangleright \Gamma_3 \backslash x) \vdash_D Q$. In case (iv) we have $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3)(x) = (\mathsf{un}\,\mathsf{end}, S)$ where $S = \mathsf{lin}\, p_2$ or $S = \mathsf{un}\,\mathsf{end}$. If $S = \mathsf{lin} p_2$ we know that $\Gamma_2(x) = (\circ, \mathsf{lin}\, p_2)$. We apply Lemma 4.2 and infer both $\Gamma, x : \mathsf{lin}\, p_1 \vdash P \triangleright (\Gamma_2 \backslash x), x : \circ$ and $(\Gamma_2 \backslash x), x : \mathsf{lin} p_2 \vdash Q \triangleright (\Gamma_3 \backslash x), x : \circ$. We apply the I.H. and infer the desired result by applying [T-PAR] with the fourth rule for context splitting: $\mathsf{map}(\Gamma, x : \mathsf{lin}\, p_1 \triangleright (\Gamma_2 \backslash x), x : \circ) \vdash_D P$ and $\mathsf{map}((\Gamma_2 \backslash x), x : \mathsf{lin} p_2 \triangleright (\Gamma_3 \backslash x), x : \circ) \vdash_D Q$. Otherwise when $S = \mathsf{un}\,\mathsf{end}$ by strengthening and I.H. we have $\mathsf{map}(\Gamma, x : \mathsf{lin}\, p_1 \triangleright (\Gamma_2 \backslash x), x : \circ) \vdash_D P$ and $\mathsf{map}(\Gamma_2 \backslash x \triangleright \Gamma_3 \backslash x) \vdash_D Q$; we conclude by applying the third rule for context splitting.

($T = (\mathsf{lin}\, p_1, \circ), = (\circ, \mathsf{lin}\, p_2)$). Similar to the previous case.

($T = \mathsf{un}\, p, = (\mathsf{un} p_1, \mathsf{un} p_2), = (\mathsf{un} p_1, \circ), = (\circ, \mathsf{un}\, p_2)$). The result follows by applying the I.H. and the second rule for context splitting in [T-PAR] to $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2) \vdash_D P$ and $\mathsf{map}(\Gamma_2 \triangleright \Gamma_3) \vdash_D Q$.

[A-OUT-L] We have $\Gamma_1, x\colon \mathsf{lin}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x : \circ$ inferred from $\Gamma_1, x : \circ \vdash v\colon T \triangleright \Gamma_2$ and $\Gamma_2 \uplus x\colon S \vdash P \triangleright \Gamma_3, x : M$ provided $\mathsf{un}(M)$. By strengthening we infer $\Gamma_1 \vdash v\colon T \triangleright \Gamma_2 \backslash x$. By I.H. we infer $\mathsf{map}(\Gamma_1, x : S \triangleright \Gamma_3, x : M) \vdash_D P$. Let Δ be equal to the environment $\mathsf{map}(\Gamma_1, x : S \triangleright \Gamma_3, x : M)$ for all entries but for x, for which we let $\Delta(x) = \mathsf{lin}\,!T.(S \triangleright M)$. We apply [T-OUT] and the fourth rule for context splitting and we conclude: $\mathsf{map}(\Gamma_1 \triangleright \Gamma_2 \backslash x) \cdot \Delta$.

[A-OUT-L-L] We have $\Gamma_1, x\colon (\mathsf{lin}\,!T.S, N) \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x : (\circ, N)$ inferred from $\Gamma_1, x\colon \mathsf{lin}\,!T.S \vdash \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x : \circ$. By I.H. we infer $\mathsf{map}(\Gamma_1, x\colon \mathsf{lin}\,!T.S \triangleright \Gamma_3, x : \circ) \vdash_D \overline{x}\langle y\rangle.P$ which we rewrite as $\mathsf{map}(\Gamma_1 \triangleright \Gamma_3), x : \mathsf{lin}\,!T.S \vdash_D \overline{x}\langle y\rangle.P$. Since $N \triangleright N$ is unrestricted, by weakening we infer $\mathsf{map}(\Gamma_1 \triangleright \Gamma_3), x : (\mathsf{lin}\,!T.S, N \triangleright N) \vdash_D \overline{x}\langle y\rangle.P$. This is the requested result since $\mathsf{map}(\Gamma_1 \triangleright \Gamma_3), x : (\mathsf{lin}\,!T.S, N \triangleright N) = \mathsf{map}(\Gamma_1, x\colon (\mathsf{lin}\,!T.S, N)) \triangleright \Gamma_3, x : (\circ, N)$.

[A- REPL] We have $\Gamma_1 \vdash\, !P \triangleright \Gamma_2$ inferred from $\Gamma_1 \vdash P \triangleright \Gamma_2$ provided $\Gamma_1 = \Gamma_2$. By I.H. we have $\mathsf{map}(\Gamma_1 \triangleright \Gamma_1) \vdash_D P$. Since $\Gamma_1 \triangleright \Gamma_1$ is an unrestricted context, so is $\mathsf{map}(\Gamma_1 \triangleright \Gamma_1)$. We apply [T-REPL] and we conclude: $\mathsf{map}(\Gamma_1 \triangleright \Gamma_1) \vdash_D\, !P$.

[A-INACT] We apply [T-INACT] and infer $\mathsf{map}(\Gamma \triangleright \Gamma) \vdash \mathbf{0}$.

□

By relying on this result we establish the soundness of the algorithm.

Corollary 4.5 (Soundness)**.** *If* $\mathsf{typeCheck}(I,P)$ *then* $I \vdash_D P$.

Proof. If the algorithm succeeds then we have $I \vdash P \triangleright \Gamma$ with $\mathsf{un}(\Gamma)$. Consider $x \in \mathrm{dom}(I)$. If $I(x) = \mathsf{lin}p$ then we know that $\Gamma(x) = \circ$. Therefore $\mathsf{map}(I \triangleright \Gamma)(x) = \mathsf{lin}p$. Similarly, if $I(x) = (\mathsf{lin}p_1, \mathsf{lin}p_2)$ then $\mathsf{map}(I \triangleright \Gamma)(x) = I(x)$. The last possibility is $I(x) = \mathsf{un}p, = (\mathsf{un}\,p_1, \mathsf{un}\,p_2)$ and we conclude that $\mathsf{map}(I \triangleright \Gamma)(x) = I(x)$. From these facts we infer $\mathsf{map}(I \triangleright \Gamma) = I$. The result then follows by applying Lemma 4.4. □

The hypothesis $\mathsf{safe}(I)$ in $\mathsf{typeCheck}(I,P)$ allows us to infer that typings are preserved by the system in Figure 3, in the following sense [5].

Lemma 4.6 (Subject reduction)**.** *Assume* $\mathsf{safe}(I)$. *If* $I \vdash_D P$ *and* $P \Rightarrow P'$ *then* $I' \vdash_D P'$ *with* $\mathsf{safe}(I')$.

Finally we prove an important result, namely that the algorithm preserves structural congruence. To tackle the proof, we need a construction similar to the one of Lemma 4.4.

Lemma 4.7. *Let* $\Gamma_1 \vdash P \triangleright \Gamma_2$. *We have* $\Gamma_1 \triangleright \Gamma_2 \vdash P \triangleright \nabla_{\Gamma_1}$ *with*

$$\nabla_\Gamma(x) = \begin{cases} \circ & \Gamma(x) = \mathsf{lin}, p \\ (\circ, \circ) & \Gamma(x) = (\mathsf{lin}\,p_1, \mathsf{lin}\,p_2), = (\mathsf{lin}\,p_1, \circ), = (\circ, \mathsf{lin}\,p_2) \\ \Gamma(x) & x \in \mathrm{dom}(\Gamma) \end{cases}$$

Given Γ_1, Γ_2 with the same domain we define the update of contexts Γ_1, Γ_2 as the operation below:

$$\Gamma_1 \uplus \Gamma_2 = \begin{cases} M_1 \uplus M_2 & \Gamma_1(x) = M_1, \Gamma_2(x) = M_2 \\ (M_1 \uplus N_1, M_2 \uplus N_2) & \Gamma_1(x) = (M_1, N_1), \Gamma_2(x) = (M_2, N_2) \end{cases}$$

Lemma 4.8 (Algorithmic weakening)**.** *Let* $\Gamma_1 \vdash P \triangleright \Gamma_2$. *The following hold.*

1. *if* $x \notin \mathrm{dom}(\Gamma)$ *then (i)* $\Gamma_1, x : M \vdash P \triangleright \Gamma_2, x : M$ *and (ii)* $\Gamma_1, x : (M,N) \vdash P \triangleright \Gamma_2, x : (M,N)$*;*
2. *if* $\Gamma_1 \uplus \Gamma$ *is defined then* $\Gamma_1 \uplus \Gamma \vdash P \triangleright \Gamma_2 \uplus \Gamma$.

Lemma 4.9 (Structural congruence)**.** *Assume* $P \equiv Q$. *We have* $\Gamma_1 \vdash P \triangleright \Gamma_2$ *if and only if* $\Gamma_1 \vdash Q \triangleright \Gamma_2$.

Proof. The most interesting case is parallel composition. Assume $\Gamma_1 \vdash P \mid Q \triangleright \Gamma_3$ inferred from $\Gamma_1 \vdash P \triangleright \Gamma_2$ and $\Gamma_2 \vdash Q \triangleright \Gamma_3$. By Lemma 4.7 we have $\Gamma_1 \triangleright \Gamma_2 \vdash P \triangleright \nabla_{\Gamma_1}$ and $\Gamma_2 \triangleright \Gamma_3 \vdash Q \triangleright \nabla_{\Gamma_2}$. In fact, it holds $\nabla_{\Gamma_1} = \nabla = \nabla_{\Gamma_2}$. Let Γ_4 be the solution of the linear system defined by equations $\Gamma_1 = (\Gamma_2 \triangleright \Gamma_3) \uplus \Gamma_4$ and $\Gamma_4 = (\Gamma_1 \triangleright \Gamma_2) \uplus \Gamma_3$. Such a solution does exist (see the Appendix). By $\Gamma_2 \vdash Q \triangleright \Gamma_3$ and Lemma 4.7 we infer $\Gamma_2 \triangleright \Gamma_3 \vdash Q \triangleright \nabla$. By using Lemma 4.8 we have $\Gamma_2 \triangleright \Gamma_3 \uplus \Gamma_4 \vdash Q \triangleright \nabla \uplus \Gamma_4$. Next take $\Gamma_1 \triangleright \Gamma_2 \vdash P \triangleright \nabla$ obtained by applying Lemma 4.7 to $\Gamma_1 \vdash P \triangleright \Gamma_2$. We apply Lemma 4.8 and infer $(\Gamma_1 \triangleright \Gamma_2) \uplus \Gamma_3 \vdash P \triangleright \nabla \uplus \Gamma_3$. Since the update of ∇ with a type environment Γ, whenever defined, satisfies the equation $\nabla \uplus \Gamma = \Gamma$, the judgments above could be rewritten as $\Gamma_1 \vdash Q \triangleright \Gamma_4$ and $\Gamma_4 \vdash P \triangleright \Gamma_3$. We apply [A-PAR] and obtain $\Gamma_1 \vdash Q \mid P \triangleright \Gamma_3$, as required. The other direction for the parallel case is analogous. The second rule for congruence of parallel processes is straightforwardly obtained from the definition of [A-PAR]. The cases for replication and inaction follow easily from the fact that the context received in output is equal to the context received in input. The cases for scope restriction follow from the definition of [A-RES]

and from algorithmic strengthening and weakening (Lemmas 4.2 and 4.8). To illustrate, take the rule $(\nu x : \mathsf{un}\, p)\mathbf{0} \equiv \mathbf{0}$. Assume $\Gamma \vdash \mathbf{0} \triangleright \Gamma$ and let $x \notin \mathrm{dom}(\Gamma)$, eventually by alpha-renaming x in the left process. By weakening we infer $\Gamma, x : \mathsf{un}\, p \vdash \mathbf{0} \triangleright \Gamma, x : \mathsf{un}\, p$. We apply [A-RES] and conclude: $\Gamma \vdash (\nu x : \mathsf{un}\, p)\mathbf{0} \triangleright \Gamma$. Now assume $\Gamma \vdash (\nu x : \mathsf{un}\, p)\mathbf{0} \triangleright \Gamma_1$ inferred from $\Gamma, x : \mathsf{un}\, p \vdash \mathbf{0} \triangleright \Gamma_1, x : O$. From the fact that this judgment has been inferred by using [A-INACT], we infer $\Gamma_1 = \Gamma$ and $O = \mathsf{un}\, p$. Since $x \notin \mathrm{fv}(\mathbf{0})$, by applying strengthening we infer the desired result, $\Gamma \vdash \mathbf{0} \triangleright \Gamma$. □

Theorem 4.10. *The* typeCheck *algorithm is effective for establishing a session-based type discipline.*

Proof. Apply Corollaries 3.4, 4.5 and Lemmas 4.6 and 4.9. □

4.1 Towards semantic completeness

The algorithm is unable to type check some process that is typable by the type system in Figure 3. This is trivially true for all processes typed by unsafe contexts, but also for typings of the form:

$$\Gamma, x : (\mathsf{lin}?T.S, \mathsf{lin}!T.\overline{S}) \vdash_D x(y).P \qquad P \equiv C[\overline{x}\langle z\rangle.P']$$

$$\Gamma, x : (\mathsf{lin}?T.S, \mathsf{lin}!T.\overline{S}) \vdash_D \overline{x}\langle x\rangle \qquad T = \mathsf{lin}?T.S\,.$$

As argued in other works on session types (e.g. [3, 1]), it seems that ruling out such processes does not comport an issue since they appear to be deadlocked. To deploy a formal proof of this statement, we have developed a typed observational theory where the behavior of processes is contrasted w.r.t. the typed knowledge of the observer [4]. The discerning capability of the observer is regulated by the type checker; in particular, type checking forces contexts to not interfere with a session shared by two participants. Behaviorally equivalent pi calculus processes exhibit the same observables in all type checked contexts. To avoid universal quantification, we rely on a proof technique based on bisimulation over typed labelled semantics.

The aim is to prove that if $I \vdash_D P$ has been inferred by using [T-INC] or [T-OUTC] with a linear channel type, then P is indistinguishable from $\mathbf{0}$ in all contexts type checked by a type environment Y *compatible* with I, noted $Y \models P \cong \mathbf{0}$. To illustrate, assume that by applying [T-INC] we infer $I, x\colon (\mathsf{lin}?T.S, \mathsf{lin}!T.\overline{S}) \vdash_D x(y).P$. Intuitively, a process type checked by Y cannot tell apart the input process from $\mathbf{0}$ because interaction on x is forbidden by Y; the compatibility condition enforces the type environment Y to do not contain input or output capabilities of x, which are already used in a linear way in I. Once obtained this result, we should be able to prove our algorithm to be semantically complete, in the following sense.

Claim (Completeness). If $I_1 \vdash_D P_1$ then there are a type environment I_2 and a process P_2 s.t. $\mathsf{typeCheck}(I_2, P_2)$ and $Y \models P_1 \cong P_2$ with Y a type environment compatible with both I_1 and I_2.

The idea is to build P_2 by descending the derivation tree for $I_1 \vdash P_1$ and by substituting subtrees of $I_1 \vdash P_1$ with a leaf $I_2 \vdash \mathbf{0}$ by following two rules:

[T-INC] $I_1, x\colon (\mathsf{lin}?T.S, \mathsf{lin}!T.\overline{S}) \vdash x(y).Q$ is exchanged with $I_2 \vdash \mathbf{0}$;

[T-OUTC] $I_1, x\colon (\mathsf{lin}!T.S, \mathsf{lin}?T.\overline{S}) \vdash \overline{x}\langle v\rangle.Q$ is exchanged with $I_2 \vdash \mathbf{0}$;

Ideally, we would let $I_1 = I_2$. Unfortunately, the linear design of the algorithm forbids this option since the call of the type checking function would return in output the linear entries not consumed by [A-INACT]. This approach indeed works if we relax the linearity of type checking an relies on an affine setting where each session type is used at most once. Otherwise, we could prune the linear entries from

$I_1, x\colon (\mathsf{lin}?T.S, \mathsf{lin}!T.\overline{S})$ and let the type environment I_2 to contain all unrestricted typings in I_1. The proof should proceed by co-induction by exploiting the typed bisimulation semantics defined in [4].

As a by-product, this technique could be also useful to detect simple deadlocks generated by erroneous programming of two opposite linear capabilities in a sequential way.

5 Examples

The protocol for the scheduling of a meeting discussed in Section 1 requires the interaction with one or more clients executed in parallel with the service. The bootstrap is due to the interaction with a client process acting as the creator of the poll, defined as process P_2 below. The process, once it has received the channel for the poll, sets the title and the date and then sends the invitation for the poll to a number of recipients by forwarding the channel established to communicate the date proposals. An instance of the protocol is obtained by considering the parallel composition of the service P_1 and the client P_2; we let $\mathsf{string} = \mathsf{un}\,\mathsf{end} = \mathsf{date}$.

$$P_1 = !x(w).(\nu p : (S_1, S_2))\ (\overline{w}\langle p\rangle.p(\mathsf{title}).p(\mathsf{date}).!p(\mathsf{date}))$$

$$P_2 = \overline{x}\langle y\rangle.y(p).(\overline{p}\langle \mathsf{Meeting}\rangle.\overline{p}\langle \mathsf{17March}\rangle.(\overline{z_1}\langle p\rangle \mid \cdots \mid \overline{z_n}\langle p\rangle))$$

$$S_1 = \mathsf{lin}\,?\mathsf{string}.\mathsf{lin}\,?\mathsf{date}.S_3 \qquad S_3 = \mathsf{un}\,?\mathsf{date}.S_3$$

$$S_2 = \mathsf{lin}\,!\mathsf{string}.\mathsf{lin}\,!\mathsf{date}.S_4 \qquad S_4 = \mathsf{un}\,!\mathsf{date}.S_4$$

By passing the (safe) context Γ below to the type checker we obtain that $P_1 \mid P_2$ is accepted. Notice that, due to Lemma 4.9, $P_2 \mid P_1$ is also accepted; we believe this feature to be of practical interest. For the sake of compactness, in the following we will shorten the unrestricted type $\mathsf{un}\,\mathsf{end}$ with end.

$$\Gamma = x : T_x, y : (\mathsf{lin}\,!S_2.\mathsf{end}, \mathsf{lin}\,?S_2.\mathsf{end}), z_1 : \mathsf{lin}\,!S_4.\mathsf{end}, \ldots, z_n : \mathsf{lin}\,!S_4.\mathsf{end}$$

$$T_x = (\mu a.\mathsf{un}?(\mathsf{lin}\,!S_2.\mathsf{end}).a, \mu b.\mathsf{un}!(\mathsf{lin}\,!S_2.\mathsf{end}).b)$$

We present below the most interesting snippets of the execution of $\mathsf{typeCheck}(\ \Gamma, P_1 \mid P_2\ )$.

Typing the (linear) poll delegation. In typing the continuation of P_1, the [A-RES] pattern is matched. Once verified that the type (S_1, S_2) is balanced, the following sub-call is launched by adding to the context the channel type for the poll:

$$\Gamma_1 = \mathsf{check}(\ \Gamma, w : \mathsf{lin}\,!S_2.\mathsf{end}, p : (S_1, S_2)\ ,\ \overline{w}\langle p\rangle.p(\mathsf{title}).p(\mathsf{date}).!p(\mathsf{date})\ ) \tag{1}$$

The call (1) matches the pattern [A-OUT-L] and a call for the continuation is invoked by setting to void the sent end point type S_2.

$$\Gamma_2 = \mathsf{check}(\ \Gamma, w : \mathsf{end}, p : (S_1, \circ)\ ,\ p(\mathsf{title}).p(\mathsf{date}).!p(\mathsf{date})) \tag{2}$$

When receiving the context Γ_2, the pattern [A-OUT-L] requires $\Gamma_2(w)$ to be unrestricted. The context returned in output to the call in (1) is obtained by setting $\Gamma_2(w) = \circ$. When receiving the context Γ_1, the pattern [A-RES] requires $\Gamma_1(p)$ to be unrestricted, and the context returned in output to the caller is obtained by removing the entry for p from Γ_1.

Typing the replicated receiving of the date. In typing the continuation of the process above the pattern [A-IN-L] is matched and the following call is launched by passing as parameter the context $\Gamma' = \Gamma, w : \mathsf{end}, p : S_3, \mathsf{title} : \mathsf{string}, \mathsf{date} : \mathsf{date}$:

$$\Gamma_3 = \mathsf{check}(\ \Gamma'\ ,\ !p(\mathsf{date})\ ) \tag{3}$$

The pattern [A- REPL] is matched and the following call is launched.

$$\Gamma_4 = \mathsf{check}(\ \Gamma',\ p(\mathsf{date})\ ) \tag{4}$$

To succeed in returning the context in output, [A- REPL] requires the context Γ_4 received from the call (4) to be equal to Γ'. This is satisfied; in this way we know that any linear resource has not been used under replication, because that would have implied the presence of a new void typing. Finally the context $\Gamma_3 = \Gamma_4$ is returned by [A- REPL] to the caller.

Typing the (unrestricted) poll delegation. In typing the continuation of the client P_2, pattern [A-OUT-L] is matched and the following call is launched by passing as argument the context $\Gamma_3 = x : T_x, y : (\circ, \mathsf{end}), p : S_4, z_1 : \mathsf{lin}\,!S_4.\mathsf{end}, \ldots, z_n : \mathsf{lin}\,!S_4.\mathsf{end}$:

$$\Gamma_5 = \mathsf{check}(\Gamma_3\,, \overline{z_1}\langle p\rangle \mid \cdots \mid \overline{z_n}\langle p\rangle) \tag{5}$$

The call (5) matches the [A-PAR] pattern and corresponds to the forwarding of the poll to the recipients in order to propose a date. The checking procedure for the first delegation is invoked:

$$\Gamma_6 = \mathsf{check}(\Gamma_3\,, \overline{z_1}\langle p\rangle) \tag{6}$$

The context Γ_6 obtained by setting to void the entry for z_1 in Γ_3 is returned to the caller (5) in order to type the next thread. Lastly context Γ_5 is obtained by setting to void the entries for $z_1, \ldots, z_n$ in Γ_3.

Remark. By setting typings to void at the end of the call for a linear typing we avoid unsound derivations as the one below

$$\Gamma_1, x : \mathsf{lin}!T.\mathsf{un}?T.S \overset{?}{\vdash} \overline{x}\langle v\rangle.P \mid x(y).Q \triangleright \Gamma_2, x : \circ\ .$$

On contrast, we could type check a standard use of pi calculus channels by using the rules for unrestricted channel types of the form $T = (\mu a.\mathsf{un}\,?T'.a, \mu b.\mathsf{un}\,!T'.b)$:

$$\Gamma_1, x : T \vdash \overline{x}\langle y\rangle.P \mid x(y).Q \triangleright \Gamma_2, x : T\ .$$

6 Discussion

We have presented a type checking algorithm for establishing a session-based discipline in (a typed version of) the pi calculus of Milner, Parrow and Walker. Following a recent approach [14] our session types are qualified as linear or unrestricted; a linear session type could evolve to an unrestricted session type. Each session type describes one end of the session; the whole session is described by a type constructor representing the concurrent behavior of the two channel ends [5]. We assess the soundness of the algorithm by showing that type checked processes are accepted by a typing system satisfying subject reduction.

Similarly to other approaches for type checking of linear and session types in the pi calculus [7, 3], we rely on the idea to type a parallel process $P \mid Q$ by ignoring the set of linear identifiers used by P before type checking Q. By reasoning at the type level, we provide for a clean account of the notion of used

identifier by introducing explicit markers for consumed types. On contrast with the cited approaches, this construction let us prove that the algorithm preserves structural congruence, and in turn that re-arranging of parallel processes is possible; we think that this feature is of practical interest.

While the algorithm is not complete, we claim that we are not loosing expressiveness since the algorithm should type checks all interesting processes accepted by the split-based typing system. We are working on a proof of this result which is based on a typed observational theory which permits to contrast the behavior of processes w.r.t. contexts regulated by type checking [4].

Qualified session types are expressive enough to represent linear types for lambda calculus [15] and linear and session types for pi calculus [7, 3]: see [5] for the details. The presented algorithm is therefore a useful tool to type check systems based on the notion of linearity of communications. For instance, the qualified session typing system presented in [13] for a variant of pi calculus relies on the idea of a double binder to represent the two ends of a communication. By projecting a qualified session type S into its dual $\overline{S}$ (cf. [5]) we could easily map this construct in our system and in turn provide a (different) type checking algorithm:

$$[\![(\nu xy\colon S)P]\!] = (\nu x\colon (S,\overline{S}))[\![P[x/y]]\!]$$

It should be noted that the choice of representing computations with a channel type representing the two ends of the communication rules out some process that could be interesting. A process that we are not able to type check is below.

$$!x(y).(\nu a)(\overline{y}\langle a\rangle.a(\mathsf{title}).a(\mathsf{date}).(!a(\mathsf{date}) \mid \overline{a}\langle 22\mathsf{March}\rangle)$$

The process consists in a modified version of the poll service where the service itself proposes a date for the meeting. Both the algorithm and the split-based system do not accept this process because in the (unrestricted) continuation type both capabilities would be needed. While we do not envisage difficulties in introducing subtyping for unrestricted types à la [11], this seems to go in the opposite direction of the idea of channel types. We therefore need to investigate subtyping solutions which take into account the channel type construct.

Lastly, a natural completion of this work would be to deploy an algorithm for type inference. We are convinced that the channel type abstraction leads to a feasible implementation based on constraint techniques (e.g. [8]).

Acknowledgments. This work was carried out during the tenure of an "Alain Bensoussan" Fellowship Programme. This programme is supported by the Marie Curie Co-Funding of Regional, National and International Programmes (COFUND) of the European Commission. The author was also supported by the Comete project, INRIA Saclay-Île de France. I would like to thank the anonymous referees for detailed comments.

References

[1] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino & Luca Padovani (2009): *Foundations of session types*. In: *PPDP*, ACM Press, pp. 219–230. Available at `http://doi.acm.org/10.1145/1599410.1599437`.

[2] Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2009): *Sessions and Session Types: An Overview*. In: *WS-FM*, *Lectures Notes in Computer Science* 6194, Springer, pp. 1–28. Available at `http://dx.doi.org/10.1007/978-3-642-14458-5_1`.

[3] Simon J. Gay & Malcolm J. Hole (2005): *Subtyping for Session Types in the Pi Calculus*. *Acta Informatica* 42(2/3), pp. 191–225.

[4] Marco Giunti (2011): *Typed Observational Equivalence for Sessions*. Submitted.

[5] Marco Giunti & Vasco T. Vasconcelos (2010): *A Linear Account of Session Types in the Pi Calculus*. In: *CONCUR, Lecture Notes in Computer Science* 6269, Springer, pp. 432–446. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_30`.

[6] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP, Lectures Notes in Computer Science* 1381, Springer, pp. 122–138. Available at `http://dx.doi.org/10.1007/BFb0053567`.

[7] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. *ACM Transactions on Programming Languages and Systems* 21(5), pp. 914–947. Available at `http://doi.acm.org/10.1145/330249.330251`.

[8] Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2009): *Typing Component-Based Communication Systems*. In: *FMOODS/FORTE, Lectures Notes in Computer Science* 5522, Springer, pp. 167–181. Available at `http://dx.doi.org/10.1007/978-3-642-02138-1_11`.

[9] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, parts I and II*. *Information and Computation* 100(1), pp. 1–77.

[10] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[11] Benjamin C. Pierce & Davide Sangiorgi (1996): *Typing and Subtyping for Mobile Processes*. *Mathematical Structures in Computer Science* 6(5), pp. 409–453.

[12] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE, Lectures Notes in Computer Science* 817, Springer, pp. 398–413.

[13] Vasco T. Vasconcelos (2009): *Fundamentals of Session Types*. In: *SFM, Lectures Notes in Computer Science* 5569, Springer, pp. 158–186. Available at `http://dx.doi.org/10.1007/978-3-642-01918-0_4`.

[14] Vasco T. Vasconcelos (2011): *Sessions, from types to programming languages*. In Luca Aceto, editor: *The Concurrency Column, Bulletin of the EATCS* 103, pp. 53–73.

[15] David Walker (2005): *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press.

A Appendix

The table in Figure 4 depicts the shape of contexts used in the proof of the case of congruence of parallel processes in Lemma 4.9. The first three columns in the table represent all possible combinations for (an entry of) safe contexts Γ_1, Γ_2 and Γ_3 such that

$$\Gamma_1 \vdash P \triangleright \Gamma_2 \text{ and } \Gamma_2 \vdash Q \triangleright \Gamma_3$$

Given these inputs, the next three columns show the output for the context in the header. Context Γ_4 in the seventh column is the solution of the following linear system:

$$\begin{cases} \Gamma_1 = (\Gamma_2 \triangleright \Gamma_3) \uplus \Gamma_4 \\ \Gamma_4 = (\Gamma_1 \triangleright \Gamma_2) \uplus \Gamma_3 \end{cases}$$

In the last column we have the environment $\nabla_{\Gamma_1} = \nabla = \nabla_{\Gamma_2}$.

Γ_1	Γ_2	Γ_3	$\Gamma_1 \triangleright \Gamma_2$	$\Gamma_2 \triangleright \Gamma_3$	$\Gamma_1 \triangleright \Gamma_3$	Γ_4	∇
$\mathsf{lin}\, p$	$\mathsf{lin}\, p$	$\mathsf{lin}\, p$	$\circ$	$\circ$	$\circ$	$\mathsf{lin}\, p$	$\circ$
$\mathsf{lin}\, p$	$\mathsf{lin}\, p$	$\circ$	$\circ$	$\mathsf{lin}\, p$	$\mathsf{lin}\, p$	$\circ$	$\circ$
$\mathsf{lin}\, p$	$\circ$	$\circ$	$\mathsf{lin}\, p$	$\circ$	$\circ$	$\mathsf{lin}\, p$	$\circ$
$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$	$\mathsf{un}\, p$
$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\mathsf{lin}p_1, \mathsf{lin}p_2)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$
$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\mathsf{lin}p_1, \circ)$	$(\circ, \circ)$
$(\circ, \mathsf{lin}p_1)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$
$(\circ, \mathsf{lin}p_1)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$	$(\circ, \circ)$
$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \mathsf{lin}p_1)$	$(\circ, \circ)$
$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$	$(\mathsf{un}p_1, \mathsf{un}p_2)$
$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$	$(\mathsf{un}p_1, \circ)$
$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$	$(\circ, \mathsf{un}p_2)$
$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$	$(\circ, \circ)$

Figure 4: Contexts in [A-PAR]