Algorithmic type checking for a pi-calculus with name matching and session types

Marco Giunti[☆]

CITI, Faculty of Sciences and Technology, New University of Lisbon

Abstract

We present a type checking algorithm for establishing a session-based discipline in a π-calculus with name matching. We account for analyzing processes exhibiting different behaviours in the branches of the if-then-else by imposing an affine discipline for session types. This permits to obtain type-safety or absence of communication errors while accepting processes of the form $\mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, \mathbf{0}$ that install a session protocol P whenever the test succeeds, and abort otherwise. To this aim we define a type system based on a notion of context split, and we prove that it satisfies subject reduction and type-safety. We implement the type system in a split-free type checking algorithm, and we prove that processes accepted by the algorithm are well-typed. We then show that processes that are typed and do not contain *Wait for* deadlocks –an input and its corresponding output (or vice-versa) are in the same thread instead of in parallel ones– are accepted by the algorithm, thus providing a partial completeness result. We conclude by investigating the expressiveness of the typing system and show that our theory subsumes recent works on linear and session types.

1. Introduction

Session types allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol. Introduced for a dialect of the π-calculus with accept/request primitives [16, 24], the concept has been transferred to different realms, including functional [8, 21, 23, 25, 18] and object-oriented programming [1, 4, 6], and even to operating systems [5]; refer to [3] for a recent overview.

In this paper, we make a step further towards the theory and practice of session types by allowing a more flexible programming style that includes the ability to check the identity of resources, and by developing an algorithmic type system to discern well-typed processes that cannot go wrong. A central feature in session-based systems is the capacity to pass on the processing of a session, which is known as delegation. The following example, which is taken

[☆]A preliminary version appeared in the proceedings of *WWV'11.*

from the original formulation of session types in [16], perfectly illustrates the idea. The design of a FTP server requires the presence of a daemon and of a pool of threads, which communicate by means of a private channel. The daemon is awake by a client request on a public channel to establish a session s. Afterwards, a random thread accepts the daemon's request to establish another session on the private channel; the aim of the session is to delegate the client session s, which will be served by the thread. A simplified code is written in the π-calculus below. Ftpd is the daemon composed with n parallel instances of FtpThread, where b is the private communication channel. The daemon waits on channel *pid* for the client's request, and afterwards sends one end point of a freshly created session k over b, while uses the remaining end point of k to delegate the client session s.

$$\begin{aligned}
\mathrm{Init}(pid) &\stackrel{\text{def}}{=} (\nu b)(\mathrm{Ftpd}[pid, b] \mid \textstyle\bigotimes_n \mathrm{FtpThread}[b]) \\
\mathrm{Ftpd}[pid, b] &\stackrel{\text{def}}{=} pid(s).(\nu k)\bar{b}\langle k\rangle.\bar{k}\langle s\rangle.\mathrm{Ftpd}[pid, b] \\
\mathrm{FtpThread}[b] &\stackrel{\text{def}}{=} b(k).k(s).s(userid, passwd).\mathrm{FtpThread}[b]
\end{aligned}$$

Clearly, the delegation mechanism requires reliable channels in order to be sound. In the system above, this should be ensured by the restriction (νb), which creates a channel dedicated to inter-communication among the daemon and the threads. The general question that we face is: can we rely *only* on this kind of assumptions in order to build safe systems? This could not be always the case, as defended in [11]. For instance, the requirement of using dedicated channels is too strong for many actual distributed systems, which often are based on open networks; also, the privacy of a channel can be disclosed (e.g. the π-calculus scope extrusion mechanism) leading to non-secure restricted channels. More practical approaches must then be taken into account in order to design trustworthy session-based systems.

Motivation. We are interested in the design and analysis of session protocols that rely on the ability to verify the identity of resources. This is relevant for many purposes, which range from the design of protocols using a notion of trust, to transaction systems featuring a roll-back option in case of wrong data. To illustrate, consider a variant of the FTP server where the communication channel among the daemon and the threads must be sent for verification to a certification authority before that it can be used for delegating a session. The system based on the Ftpd daemon below is instantiated with a tuple of (free) delegation channels $B = (b_1, \ldots, b_n)$: the daemon checks the trust of b_1 by first establishing a session h with the authority by means of the channel *cert*, and then by sending b_1 over h. The continuation then waits for the answer of the authority: if b_1 is trusted then it is used to send a fresh session k to FtpThread,

otherwise the certification protocol restarts by verifying the trust of b_2.

$$\mathrm{Ftpd}[pid, B] \stackrel{\text{def}}{=} pid(s).\mathrm{Cert}[s, B]$$

$$\mathrm{Cert}[s, B] \stackrel{\text{def}}{=} (\nu h)\overline{cert}\langle h\rangle.\overline{h}\langle b_1\rangle.h(m).\mathsf{if}\, m = ok\, \mathsf{then}\, (\nu k)\overline{b_1}\langle k\rangle.\overline{k}\langle s\rangle.\mathrm{Ftpd}[pid, B] \;\mathsf{else}\; \mathrm{Cert}[s, (b_2, \ldots, b_n)]$$

We omit the code for the certification authority and note that it crucially relies on a tuple of certified channels $(t_1, \ldots, t_m)$, and on sub-processes of the form $\mathsf{if}\, b_i = t_j\, \mathsf{then}\, P\, \mathsf{else}\, Q$: given $i \in \{1, \ldots, n\}$, b_i is considered safe when it is identical to t_j, for some $j \in \{1, \ldots, m\}$. Our aim is to deploy an algorithmic type system, or *type-checker*, that can accept processes of the form above basing on session types.

Related Work. To the best of our knowledge, the combination of session types and name matching has been studied before in [10], which introduces a session-based type discipline for a π-calculus with accept/request primitives. In that work, session types follow a linear discipline and the two branches of an if-then-else process must behave identically. On contrast, in this paper we build on a standard π-calculus with name matching and rely on an affine discipline for session types to enforce that each end point of a session is used *at most* once. We believe that requiring the two branches to follow the same linear discipline is unnecessarily restrictive: there are indeed many interesting processes showing a different session behaviour after the name matching test that we want to analyze. [13] introduces a session typing system for a conventional π-calculus with booleans. Types can represent one or both end points of the communication, and are qualified as *linear* or *unrestricted*: session types follow a linear discipline and may evolve to unrestricted types. Types are allocated to processes by means of *context split*, which is inherently non-deterministic. Similarly, [14] considers a session typing system for π-calculus where types represent both end points of the communication and are qualified. Both the papers [13] and [14] do not address algorithmic issues. The typing system presented in this paper is based on types representing both ends of the communication and embeds an expressive fragment of [13]. We deploy an algorithmic typing system that does not rely on context split, similarly to previous work for linear lambda calculus [27] and π-calculus with polarized channels [7]. The main idea in [27] is to avoid to split the context into parts before checking a complex expression by passing the entire context as input to the first sub-expression and have it return the unused portion as an output. In the setting of concurrent computations, when typing a parallel process $P \mid Q$, split-free solutions calculate the set of linear identifiers used by P in order to remove it before type checking Q. This approach, previously outlined for linear π-calculus [19], has been implemented in the session system of [7] by representing each channel end point with a distinct identifier. On contrast, our algorithm works on a standard pi calculus and accepts processes that consume different session identifiers in the two branches of the if-then-else.

Contributions. We introduce a typing theory for establishing a session-based discipline in a π-calculus with name matching. To analyze processes that exhibit different behaviours in the two branches of the if-then-else, we establish an affine discipline for session types by requiring that each end point of a session is used *at most* once. We define a typing system based on a notion of context split and we show that well-typed processes cannot go wrong. We implement the typing rules in a split-free algorithm based on functional patterns, and show that the implementation is sound, that is: processes accepted by the algorithm are well-typed. For the reverse direction we obtain a partial completeness result: typed process not containing *Wait for* deadlocks [2] are accepted by the algorithm. We see this result as satisfactory: a fine-grained analysis permits us to detect this class of typed deadlocks, which in turn we reject. We investigate the expressiveness of our calculus and typing system by encoding a fragment of the linear π-calculus with session types [13]; as a by product, we are able to represent systems based on linear types [19] and session types [7].

A preliminary version of this paper appeared in [9], which deploys an algorithmic solution for [13]. The work presented here is significantly different, since we type more processes eventually relying on the ability to verify the identity of resources. To complete the paper, we added the following results: (1) we prove that well-typed processes cannot go wrong (Theorem 3.10), (2) we prove that typed processes not containing *Wait for* deadlocks are accepted by the algorithm (Theorem 4.8), (3) we embed a fragment of [13], study its expressiveness, and provide a typing correspondence result (Theorem 5.3).

Plan of the paper. Section 2 introduces our language, a typed π-calculus with name matching, and its typing system; a few examples are drawn. Section 3 shows that typed processes do not reach errors. Section 4 contains a type checking algorithm that implement the type system in Section 2, and studies its properties. Section 5 investigates the expressiveness of our system. In Section 6 we discuss the limitations of our approach and envision future work, concluding the paper.

2. Typed π-calculus

This section introduces the syntax and the semantics of our typed π-calculus. The formal definition is in Figure 1. We consider types of the form (S_1, S_2) and (E_1, E_2) where each S_i is a type describing the behavior of an end point of a session and each E_i is a type describing the end point of a channel. An end point of a *session* S finishes with the type `end`. A type of the form $!T.S$ describes a channel end able to send at most once a variable of type T and to proceed as prescribed by S, following an affine discipline. Similarly, $?T.S$ describes a channel end able to receive at most once a variable of type T and to continue as S. The type `end` describes an end point of a session on which no further interaction is possible. An end point of the form $?T$ describes a *channel* that could be used in an unrestricted way to receive a variable of type T. Similarly $!T$ describes an end point channel that can used zero or more times to send a

Syntax of typed processes

T	$::=$	Types	P, Q	$::=$	Processes
	(S, S)	session		$\overline{x}\langle y\rangle.P$	output
	(E, E)	channel		$x(y).P$	input
S, R	$::=$	Session end point		$(\nu y\colon T)P$	restriction
	$?T.S$	input		$\mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q$	name matching
	$!T.S$	output		$(P \mid Q)$	composition
	end	termination		$!P$	replication
E	$::=$	Channel end point		$\mathbf{0}$	inaction
	$?T$	input			
	$!T$	output			

Operator for type progression

$$\mathsf{next}(?T.S) = S \qquad \mathsf{next}(!T.S) = S \qquad \mathsf{next}(\mathsf{end}) = \mathsf{end}$$
$$\mathsf{next}((S_1, S_2)) = (\mathsf{next}(S_1), \mathsf{next}(S_2)) \qquad \mathsf{next}((E_1, E_2)) = (E_1, E_2)$$

Rules for structural congruence

$$P \mid Q \equiv Q \mid P \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \mid \mathbf{0} \equiv P \qquad !P \equiv P \mid !P$$
$$(\nu y\colon (A, \overline{A}))P \mid Q \equiv (\nu y\colon (A, \overline{A}))(P \mid Q) \quad (\nu y\colon (A, \overline{A}))\mathbf{0} \equiv \mathbf{0}$$
$$(\nu x\colon (A, \overline{A}))(\nu y\colon (B, \overline{B}))P \equiv (\nu y\colon (B, \overline{B}))(\nu x\colon (A, \overline{A}))P$$

Rules for reduction

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

$$\frac{P \xrightarrow{y} P' \qquad \mathsf{next}(T) = T'}{(\nu y\colon T)P \xrightarrow{\tau} (\nu y\colon T')P'} \qquad \frac{P \xrightarrow{\mu} P' \qquad \mu \neq y}{(\nu y\colon T)P \xrightarrow{\mu} (\nu y\colon T)P'} \qquad \text{[R-ResB],[R-Res]}$$

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

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \frac{P \equiv Q \quad Q \xrightarrow{\mu} Q' \quad Q' \equiv P'}{P \xrightarrow{\mu} P'} \qquad \text{[R-Par],[R-Struct]}$$

Figure 1: Typed π-calculus: syntax and semantics

variable of type T. Types of the form (S_1, S_2) are called *session types*, while types (E_1, E_2) are called *channel types*. We will often use the type $(\mathsf{end}, \mathsf{end})$ to represent sessions that cannot be used in i/o (but can be passed around), and call it *top* for short: $\top \stackrel{\text{def}}{=} (\mathsf{end}, \mathsf{end})$. Basic channel values are represented by means of $(?\top, !\top)$, which we refer to as *bot*: $\bot \stackrel{\text{def}}{=} (?\top, !\top)$. We use the meta-variables A, B to range over session end points S, R and channel end points E.

End point type duality plays a central role ensuring that communication between the two end points of a channel proceeds smoothly. Intuitively, the dual of an output is an input and the dual of input is an output whenever the type expected in input is exactly the type sent in output. In particular if S_2 is dual of S_1, noted $\overline{S}_1 = S_2$, then $?T.S_1$ is dual of $!T.S_2$.

$$\overline{?T.S} = !T.\overline{S} \qquad \overline{!T.S} = ?T.\overline{S} \qquad \overline{?T} = !T \qquad \overline{!T} = ?T \qquad \overline{\mathsf{end}} = \mathsf{end}$$

Types T of the form $(\mathsf{end}, \mathsf{end})$ and (E_1, E_2) are called *terminated*, and denoted with the predicate $\mathsf{term}(T)$. A channel having terminated type cannot be used for session-based communication.

$$\mathsf{term}((\mathsf{end}, \mathsf{end})) \qquad \mathsf{term}((E_1, E_2))$$

Typed processes are ranged over P, Q. We rely on a set of variables, also referred as names, ranged over by $a, b, \ldots, u, v, \ldots x, y, z$. We consider synchronous output and input processes, in the forms $\overline{x}\langle z\rangle.P$ and $x(y).P$. The restricted process $(\nu y\colon T)P$ provides for create a variable y decorated with the type T. The matching process $\mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q$ allows for comparison of variables. The remaining processes are parallel composition, replication, and inaction. The binders for the language appear in parentheses: the variable y is bound in $x(y).P$ and in $(\nu y\colon T)P$. Free and bound variables in processes, noted respectively with $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$, are defined accordingly, and so is alpha conversion, substitution of variable y by variable z in a process P, denoted $P[z/y]$. We follow Barendregt's variable convention, requiring bound variables to be distinct from each other and from free variables in any mathematical context. We will often omit trailing $\mathbf{0}$'s and abbreviate processes $\overline{x}\langle v\rangle.\mathbf{0}$ and $x(y).\mathbf{0}$ with $\overline{x}\langle v\rangle$ and $x(y)$.

We define the semantics of our calculus via a reduction relation, also in in Figure 1, and use structural congruence to rearrange processes. Structural congruence, noted $\equiv$, is the smallest relation on processes including the equivalence induced by alpha-conversion, and the rules in Figure 1. Most rules are standard, while in the rules for the scope of variables we enforce the decoration type to be of a specific form, to ensure soundness. In the first rule on the second line, we allow the scope of x to encompass Q whenever y is bound by a declaration $(\nu y\colon T)$; due to variable convention, variable y cannot be free in Q. The second rule in the same line permits to remove a binding, while the rule in the third line permits to exchange two bindings of the required form. The reduction is the smallest relation $\stackrel{\mu}{\longrightarrow}$ on processes including the rules in the same figure,

where we let μ range over the internal transition τ and variables x, y, z. The aim of the label on the arrow is to represent the evolution of types of restricted variables; this is only for convenience, and has no semantic impact (cf. [19]).

The [R-Com] rule permits to communicate 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 in the input process is replaced by the variable z. We record on the arrow the variable on which the synchronization takes place, that is x; this will be used in rules [R-ResB],[R-Res]. Rule [R-ResB] depicts the case whether the reduction has been originated from a reduction on the channel under restriction; the type of the restriction of the continuation is inferred by means of the next operator, which allows to unfold a session type. Rule [R-Res] applies when the reduction is inferred from a channel not bound by the restriction, and does not change the type of restriction of the continuation. Rule [R-IfT] says that a matching process contrasting two identical variables does an internal transition τ and reaches its continuation. Rule [R-IfF] applies when the two contrasted variables are different. Rule [R-Par] describes the behaviour of parallel processes, and rule [R-Struct] allows for rearrangement of processes by using structural congruence. We will often abuse the notation and write $P \to P'$ to indicate that there is μ such that $P \xrightarrow{\mu} P'$. Similarly, we write $P \Rightarrow P'$ to indicate that a) $P \to \cdots \to P'$ or that b) $P' = P$.

Type system. Type environments or *contexts* Γ are a possibly empty map from variables to types. In an environment $\Gamma, x\colon T$ we assume that x does not occur in Γ; we also assume the variable bindings in Γ to be unordered. We let $\mathrm{dom}(\Gamma)$ be the set of names in Γ. We write $\Gamma\backslash x$ to indicate the context Γ_1 whenever $\Gamma = \Gamma_1, x\colon T$ or whenever $\Gamma_1 = \Gamma$ and $x \notin \mathrm{dom}(\Gamma)$. A context Γ is *terminated*, noted $\mathsf{term}(\Gamma)$, whenever for all $x \in \mathrm{dom}(\Gamma)$ it holds $\mathsf{term}(\Gamma(x))$. Types and contexts can be split; this will be exploited in the typing rule for parallel processes and in the rules for sending a variable. The definition of the *split* relation over types and type environments, noted $\circ$, is in Figure 2. We will often write $\Gamma_1 \circ \Gamma_2$ to indicate that there is Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Essentially, type split permits to separate the two channel ends of a session, and to spawn the use of terminated types in an unbounded manner. To illustrate, suppose that type $(?T.S, \mathsf{end})$ describes the local use of a variable x in process P_1, and that $(\mathsf{end}, !T.R)$ describes the use of x in process P_2. The global use of x in P_1 and P_2, that is the use of x in the parallel process $P_1 \mid P_2$, is depicted by the session type $(?T.S, !T.R)$, which is eventually split into $(?T.S, \mathsf{end})$ and $(\mathsf{end}, !T.R)$. On contrast, a process using a channel x at type $(?T.S, !T.R)$ cannot be put in parallel with a process using x at type $!T.S'$; this would break the affine type discipline for sessions, which says that each end point of a session must be used at most once. We can instead have x used at type $(?T, !T)$ in several threads, being the usage of a channel type unrestricted, or x used both at type $(?T.S, !T.R)$ in one thread and at type $(\mathsf{end}, \mathsf{end})$ in the remainder.

The typing system is in Figure 2. Rule [T-Var] is for typing values and requires the environment pruned from the typed entry to be terminated, so to

Session end point split rules

$$S = S \circ \mathsf{end} \qquad S = \mathsf{end} \circ S$$

Type split rules

$$\frac{R = R_1 \circ R_2 \qquad S = S_1 \circ S_2}{(R, S) = (R_1, S_1) \circ (R_2, S_2)} \qquad (E_1, E_2) = (E_1, E_2) \circ (E_1, E_2)$$

Context split rules

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad T = T_1 \circ T_2}{\Gamma, x\colon T = (\Gamma_1, x\colon T_1) \circ (\Gamma_2, x\colon T_2)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad T = T_2 \circ T_1}{\Gamma, x\colon T = (\Gamma_1, x\colon T_1) \circ (\Gamma_2, x\colon T_2)}$$

Typing rule for values

$$\frac{\mathsf{term}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad \text{[T-Var]}$$

Typing rules for processes

$$\frac{\Gamma, x\colon (S_1, S_2), y\colon T \vdash P}{\Gamma, x\colon (?T.S_1, S_2) \vdash x(y).P} \qquad \frac{\Gamma, x\colon (S_1, S_2), y\colon T \vdash P}{\Gamma, x\colon (S_1, ?T.S_2) \vdash x(y).P}$$

[T-InS-L],[T-InS-R]

$$\frac{\Gamma(x) = (?T, E) \qquad \Gamma, y\colon T \vdash P}{\Gamma \vdash x(y).P} \qquad \frac{\Gamma(x) = (E, ?T) \qquad \Gamma, y\colon T \vdash P}{\Gamma \vdash x(y).P}$$

[T-In-L],[T-In-R]

$$\frac{\Gamma_1 \vdash y\colon T \qquad \Gamma_2, x\colon (S_1, S_2) \vdash P}{\Gamma_1 \circ (\Gamma_2, x\colon (!T.S_1, S_2)) \vdash \overline{x}\langle y\rangle.P} \qquad \frac{\Gamma_1 \vdash y\colon T \qquad \Gamma_2, x\colon (S_1, S_2) \vdash P}{\Gamma_1 \circ (\Gamma_2, x\colon (S_1, !T.S_2)) \vdash \overline{x}\langle y\rangle.P}$$

[T-OutS-L],[T-OutS-R]

$$\frac{\Gamma_1 \vdash y\colon T \qquad \Gamma_2(x) = (!T, E) \qquad \Gamma_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash \overline{x}\langle y\rangle.P} \qquad \text{[T-Out-L]}$$

$$\frac{\Gamma_1 \vdash y\colon T \qquad \Gamma_2(x) = (E, !T) \qquad \Gamma_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash \overline{x}\langle y\rangle.P} \qquad \text{[T-Out-R]}$$

$$\frac{\Gamma(x) = (E_1, E_2) = \Gamma(y) \qquad \Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q} \qquad \text{[T-If]}$$

$$\frac{\Gamma, x\colon (A, \overline{A}) \vdash P}{\Gamma \vdash (\nu x\colon (A, \overline{A}))P} \qquad \frac{\mathsf{term}(\Gamma) \qquad \Gamma \vdash P}{\Gamma \vdash !P} \qquad \text{[T-Res],[T-Repl]}$$

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \Gamma \vdash \mathbf{0} \qquad \text{[T-Par],[T-Inact]}$$

Figure 2: Type system

give a chance to affine resources to be used. The rule for typing processes follow. For input we have four rules [T-InS-L],[T-InS-R], [T-In-L] and [T-In-R]; rules [T-InS-L],[T-InS-R] are used whenever the input channel is a session respectively on the left and on the right, and [T-In-L] and [T-In-R] are performed whenever the input is a channel end-point respectively on the left and on the right. Rule [T-InS-L] permits to type an input channel x by using the end point type $?T.S_1$ on the left of a type $(?T.S_1, S_2)$. If x is typed with $?T.S_1$, we know that the bound variable y is of type T, and we type P under the extra assumption $y\colon T$. Equally important is the fact that the continuation uses channel x at continuation type (S_1, S_2), that is, process $x(y).P$ uses channel x at type $(?T.S_1, S_2)$ whereas P may use the *same* channel this time at type (S_1, S_2). Rule [T-InS-R] permits to type an input x described by a session type $(S_1, ?T.S_2)$ by following the same mechanism. Rule [T-In-L] permits to type an input channel x by using the channel end point type $?T$ on the left of a type $(?T, E)$. The continuation P is typed by using the same type for x, eventually adding the assumption $y\colon T$ for the bound variable. Rule [T-In-R] applies when the channel end point for the input is on the right.

Then four rules for typing an output follow. Symmetrically, we have two rules for typing a session, [T-OutS-L], [T-OutS-R], and two rules for typing an output channel described by a channel type, [T-Out-L],[T-Out-R]. In the typing rules for output we account for sending a variable that can have a channel or a session type; in the latter case we have *delegation* of one or both ends of the session. To illustrate the delegation mechanism we describe the rule for typing an output with the left end point of a channel type, [T-Out-L]; the remaining rules follow a similar schema. Rule [T-Out-L] permits to use an environment Γ such that $\Gamma(x) = (!T, E)$ to send a variable y at type T on x and to continue as P, given that there is a split $\Gamma = \Gamma_1 \circ \Gamma_2$ such that $\Gamma_1 \vdash y\colon T$, and $\Gamma_2 \vdash P$. For instance if $T = (?T'.S_1, \mathsf{end})$ and $\Gamma(y) = (?T'.S_1, !T'.S_2)$ then y is both sent at type T and used at type $(\mathsf{end}, !T'.S_2)$ in the continuation P.

Rule [T-If] permits to type an if-then-else process which compares the identity of two variables, which must have the same channel type; comparing the identity of sessions can indeed lead to gain more information and in turn break the affine discipline, similarly to what happens in i/o types [22]. The rule for restriction [T-Res] permits to type a variable having a type composed by dual end points. This is essential to preserve subject reduction, as we will see in the next section. Rule [T-Par] allow to type two processes put in parallel with a context Γ if there is a split of Γ into Γ_1 and Γ_2 such that P is typed by Γ_1 and Q is typed by Γ_2. This permits to preserve the affine invariant, so that we cannot have two processes using the same end of a session. In the replication rule, [T-Repl], we require the environment to be terminated, so that the process under replication does not contain free sessions. Any context could be used in rule [T-Inact] to type the inert process; as introduced, requiring contexts to be terminated (cf. [13]) rules out interesting processes that show different behaviour in the branches of the if-then-else.

Example 2.1. *We write a derivation for a variant of the FTP daemon using*

certified channels of the introduction. We consider a tuple of free channels $(b_1, \ldots, b_n)$*,* $n > 0$*, and let* $Cert[s, \epsilon] \stackrel{def}{=} \mathbf{0}$*, where* ϵ *is the empty tuple. The code for the daemon is below.*

$$Ftpd[pid, (b_1, \ldots, b_n)] \stackrel{def}{=} !\, pid(s).Cert[s, (b_1, \ldots, b_n)]$$

$$Cert[s, (b_1, \ldots, b_n)] \stackrel{def}{=} (\nu h\colon T_h)\overline{cert}\langle h\rangle.\overline{h}\langle b_1\rangle.h(m).\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_1] \,\mathsf{else}\, Cert[s, (b_2, \ldots, b_n)]$$

$$Use[s, b] \stackrel{def}{=} (\nu k\colon T_k)\overline{b}\langle k\rangle.\overline{k}\langle s\rangle$$

We assign labels of the form T_y *to types, meaning that* y *has type* T_y*, and identify context* Γ *and* Γ'*:*

$$T_k = (?T_s.\mathsf{end}, !T_s.\mathsf{end}) \qquad T'_k = (?T_s.\mathsf{end}, \mathsf{end}) \qquad T''_k = (\mathsf{end}, !T_s.\mathsf{end})$$
$$T_h = (!T_b.?\bot.\mathsf{end}, ?T_b.!\bot.\mathsf{end}) \qquad T'_h = (\mathsf{end}, ?T_b.!\bot.\mathsf{end}) \qquad T''_h = (!T_b.?\bot.\mathsf{end}, \mathsf{end})$$
$$T'''_h = (?\bot.\mathsf{end}, \mathsf{end}) \qquad T_s = (?\bot.\mathsf{end}, \mathsf{end}) \qquad T_b = (?T'_k, !T'_k) \qquad T_c = (?T'_h, !T'_h)$$
$$\Gamma = pid\colon (?T_s, !T_s), cert\colon T_c, b_1\colon T_b, \ldots, b_n\colon T_b, ok\colon \bot \qquad \Gamma' = \Gamma, s\colon T_s, h\colon \top$$

The derivation for process $Use[s, b_i]$ *is below, where* $i \in \{1, \ldots, n\}$*. We omit to label the judgements obtained by using [*T-Var*].*

$$\dfrac{\dfrac{\Gamma', s\colon \top, h\colon \top, k\colon T'_k, m\colon \bot \vdash k\colon T'_k \qquad \dfrac{\Gamma', k\colon \top, m\colon \bot \vdash s\colon T_s \qquad \dfrac{}{\Gamma, s\colon \top, h\colon \top, k\colon \top, m\colon \bot \vdash \mathbf{0}}\,([\text{T-Inact}])}{\Gamma', k\colon T''_k, m\colon \bot \vdash \overline{k}\langle s\rangle}\,([\text{T-OutS-R}])}{\Gamma', k\colon T_k, m\colon \bot \vdash \overline{b_i}\langle k\rangle.\overline{k}\langle s\rangle}\,([\text{T-Out-R}])}{\Gamma', m\colon \bot \vdash Use[s, b_i]}\,([\text{T-Res}])$$

The derivation for the continuation of $Cert[s, b_n]$ *is the following.*

$$\dfrac{\Gamma', m\colon \bot \vdash Use[s, b_n] \qquad \dfrac{}{\Gamma', m\colon \bot \vdash Cert[s, \epsilon]}\,([\text{T-Inact}])}{\Gamma', m\colon \bot \vdash \mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n] \,\mathsf{else}\, Cert[s, \epsilon]}\,([\text{T-If}])$$

We now type process $Cert[s, b_n]$*; we use* $\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n]$ *as a short for* $\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n] \,\mathsf{else}\, Cert[s, \epsilon]$*.*

$$\dfrac{\dfrac{\Gamma', s\colon \top, h\colon T'_h \vdash h\colon T'_h \qquad \dfrac{\Gamma, s\colon \top, h\colon \top \vdash b_n\colon T_b \qquad \dfrac{\Gamma', m\colon \bot \vdash \mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n]}{\Gamma, s\colon T_s, h\colon T'''_h \vdash h(m).\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n]}\,([\text{T-InS-L}])}{\Gamma, s\colon T_s, h\colon T''_h \vdash \overline{h}\langle b_n\rangle.h(m).\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n]}\,([\text{T-OutS-L}])}{\Gamma, s\colon T_s, h\colon T_h \vdash \overline{cert}\langle h\rangle.\overline{h}\langle b_n\rangle.h(m).\mathsf{if}\, m = ok \,\mathsf{then}\, Use[s, b_n]}\,([\text{T-Out-R}])}{\Gamma, s\colon T_s \vdash Cert[s, b_n]}\,([\text{T-Res}])$$

By following the same schema, we can type process $Cert[s, (b_{n-1}, b_n)]$*.*

$$\dfrac{\Gamma, s\colon T_s, h'\colon \top, m'\colon \bot \vdash \mathsf{if}\, m' = ok \,\mathsf{then}\, Use[s, b_{n-1}] \,\mathsf{else}\, Cert[s, b_n]}{\cdots}$$
$$\Gamma, s\colon T_s \vdash Cert[s, (b_{n-1}, b_n)]$$

We conclude with the derivation for the daemon, where the judgement for $Cert[s, (b_1, \ldots, b_n)]$ *is built by chaining the derivations above.*

$$\dfrac{\dfrac{\Gamma, s\colon T_s \vdash Cert[s, (b_1, \ldots, b_n)]}{\Gamma \vdash pid(s).Cert[s, (b_1, \ldots, b_n)]}\ (\lceil\text{T-In-L}\rfloor)}{\Gamma \vdash Ftpd[pid, (b_1, \ldots, b_n)]}\ (\lceil\text{T-Repl}\rfloor)$$

Example 2.2. *We design an e-commerce system to sell items with a rebate option, and outline how we can type it. To ease the notation, in the code below we emphasize the variables used for communication, and assume the remaining variables to have type* $\bot$*. We also send pair of values, noted* $\langle v, w\rangle$*. The selling service waits for requests from clients to establish a session* q*. Afterwards, the service receives from the client two distinct session end points: the* first *endpoint, tha is* a*, is used to receive the chosen item, the coupon for the rebate, and the client's credit card number; the* second *endpoint, that is* b*, is used to roll-back the transaction in case that the coupon is wrong.*

$$\begin{aligned}
\mathit{SellingService} &\stackrel{def}{=} !p(q).q(a).q(b).a(\text{item}).a(\text{coupon}).a(\text{creditCard}).\\
&\qquad \mathsf{if}\ \text{coupon} = \text{rebateCode}\ \mathsf{then}\ \mathit{OkSendRebate}\\
&\qquad \mathsf{else}\ \mathit{RepeatNoRebate}\\
\mathit{OkSendRebate} &\stackrel{def}{=} (\nu r\colon (?\bot.\mathsf{end}, !\bot.\mathsf{end}))(\overline{a}\langle r\rangle \mid \overline{\mathit{server}}\langle \text{rebateCode}, r\rangle)\\
\mathit{RepeatNoRebate} &\stackrel{def}{=} \overline{b}\langle \text{couponInvalid}\rangle.b(\text{item}).b(\text{creditCard})
\end{aligned}$$

The reader will notice that the two branches OkSendRebate and RepeatNoRebate describe two different behaviours for a *and* b*:* a *is used in OkSendRebate to send a freshly created rebate channel* r *(also delegated to a server) and is left unused in RepeatNoRebate, while* b *is used in RepeatNoRebate to re-start the transaction without a rebate option and is left unused in OkSendRebate. We stress that requiring* a *and* b *to have the same session behaviour in both branches would be unnecessarily artificial. We briefly outline the typings of the session variables* a *and* b*, and of server; the type of channel* p *and of its bound variable* q *are defined accordingly.*

$$\mathit{InputRebate} \stackrel{def}{=} (?\bot.\mathsf{end}, \mathsf{end}) \qquad \mathit{OutputRebate} \stackrel{def}{=} (\mathsf{end}, !\bot.\mathsf{end})$$
$$a\colon (?\mathit{Item}.?\mathit{Coupon}.?\mathit{CreditCard}.!\mathit{OutputRebate}.\mathsf{end}, \mathsf{end})$$
$$b\colon (!\mathit{Message}.?\mathit{Item}.?\mathit{CreditCard}.\mathsf{end}, \mathsf{end})$$
$$\mathit{server}\colon (?\langle \mathit{Coupon,InputRebate}\rangle, !\langle \mathit{Coupon,InputRebate}\rangle)$$

3. Typed processes do not go wrong

This section contains the proof of the main result for the typing system: typed processes do not reach errors during the computation. A key property in building towards this result is subject reduction, which we tackle by introducing auxiliary definitions and lemmas.

We start by introduction the notion *termination closure* of a context Γ, noted $\mathscr{T}(\Gamma)$, which is the projection of Γ that is available to all threads.

$$\mathscr{T}(\emptyset) = \emptyset \qquad \mathscr{T}(\Gamma, x\colon (S_1, S_2)) = \mathscr{T}(\Gamma), x\colon (\mathsf{end}, \mathsf{end})$$
$$\mathscr{T}(\Gamma, x\colon (E_1, E_2)) = \mathscr{T}(\Gamma), x\colon (E_1, E_2)$$

A termination closure can be always extracted by means of context split. We will use this property in proving the lemma that is the wedge of subject reduction (Lemma 3.6).

Lemma 3.1. *The following hold.*

1. *If* $\mathsf{term}(\Gamma)$ *then* $\mathscr{T}(\Gamma) = \Gamma$*;*
2. $\Gamma = \Gamma \circ \mathscr{T}(\Gamma)$*;*
3. $\Gamma = \mathscr{T}(\Gamma) \circ \Gamma$.

Proof. By induction on the size of $\mathrm{dom}(\Gamma)$. The proof is straightforward. □

Weakening is a fundamental property of the typing system, and permits to change a type end with any session type.

Lemma 3.2 (Weakening)**.** *The following hold.*

1. *If* $\Gamma, x\colon (\mathsf{end}, S) \vdash P$ *then* $\Gamma, x\colon (R, S) \vdash P$
2. *If* $\Gamma, x\colon (R, \mathsf{end}) \vdash P$ *then* $\Gamma, x\colon (R, S) \vdash P$.

Proof. By induction on the length of the inference. To illustrate, consider (1) and suppose that the derivation $\Gamma, x\colon (\mathsf{end}, S) \vdash \overline{z}\langle w\rangle.P$ terminates with [T-OutS-R]. Thus $\Gamma, x\colon (\mathsf{end}, S) = \Gamma_1 \circ (\Gamma_2, z\colon (S_1, !T.S_2))$ and $\Gamma_1 \vdash w\colon T$ and $\Gamma_2, z\colon (S_1, S_2) \vdash P$. We know $\Gamma_1 = \Gamma', x\colon (\mathsf{end}, S')$, and $\Gamma_2 = \Gamma'', x\colon (\mathsf{end}, S'')$ when $z \neq x$, otherwise we have $z = x$, and $S_1 = \mathsf{end}$, and $x \notin \mathrm{dom}(\Gamma_2)$. By I.H. when $z \neq x$ we have $\Gamma'', x\colon (R, S''), z\colon (S_1, S_2) \vdash P$, otherwise $\Gamma_2, z\colon (R, S_2) \vdash P$. In both cases we conclude by applying [T-OutS-R] and $\Gamma_1 \vdash w\colon T$, obtaining $\Gamma, x\colon (R, S) \vdash \overline{z}\langle w\rangle.P$. Now assume case [T-If] holds: $\Gamma \vdash \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q$ inferred from $\Gamma(x) = (E_1, E_2) = \Gamma(y) =$, $\Gamma \vdash P$ and $\Gamma \vdash Q$. Let $\Gamma = \Gamma_1, z\colon (\mathsf{end}, S)$. We apply the I.H. and infer $\Gamma_1, z\colon (R, S) \vdash P$ and $\Gamma_1, z\colon (R, S) \vdash Q$. Since $z \neq x, z \neq y$ we apply [T-If] and infer the desired result, $\Gamma_1, z\colon (R, S) \vdash \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q$. As further example, consider case [T-Par]. We have $\Gamma, x\colon (\mathsf{end}, S) \vdash P \mid Q$ inferred from $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$ where $\Gamma, x\colon (\mathsf{end}, S) = \Gamma_1 \circ \Gamma_2$. Therefore $\Gamma_1(x) = (\mathsf{end}, S_1)$ and $\Gamma_2(x) = (\mathsf{end}, S_2)$ where $S = S_1 \circ S_2$. We apply the I.H. and infer that $(\Gamma_1 \backslash x), x\colon (R, S_1) \vdash P$. We apply [T-Par] and deduce that $\Gamma, x\colon (R, S) \vdash P \mid Q$, as desired. □

The corollary below generalizes the result to environments; the proof is based on multiple applications of the weakening lemma.

Corollary 3.3. *If* $\Gamma_1 \vdash P$ *and* $\Gamma_1 \circ \Gamma_2$ *is defined then* $\Gamma_1 \circ \Gamma_2 \vdash P$.

Reduction preserves typability for a certain class of type environments, which we refer as *balanced*. Essentially, a balanced environment ensures that in a synchronization the type of the variable sent in output is equal to the type expected in input. This notion is standard in session types (cf. [16]), and has a counter-part in typing systems featuring subtyping (e.g. [22, 7]), where the type expected in input must be a super-type of the type of the value sent in output. The formal definition of the balanced predicate, noted $\mathsf{bal}(T)$, is below. The case whether one channel end has type S while the other channel end has type end depicts a channel used only with one modality (the one of S), which can be input, output, or none. An environment Γ is balanced whenever $\mathsf{bal}(\Gamma(x))$ for all $x \in \mathrm{dom}(\Gamma)$.

$$\mathsf{bal}((S, \overline{S})) \qquad \mathsf{bal}((S, \mathsf{end})) \qquad \mathsf{bal}((\mathsf{end}, S)) \qquad \mathsf{bal}((E, \overline{E}))$$

To understand why subject reduction does not hold for unbalanced contexts, consider the composition of $P \stackrel{\text{def}}{=} a(x).\overline{x}\langle z\rangle$ and $Q \stackrel{\text{def}}{=} (\nu b\colon (?\top.\mathsf{end}, !\top.\mathsf{end}))\ \overline{a}\langle b\rangle.(b(y) \mid \overline{b}\langle w\rangle)$. Process P expects on a a channel to be used in output, while process Q sends over a a session channel that cannot be used in i/o, since its continuation uses both the input and the output end point of the session. The composition $P \mid Q$ reduces to a non-typed process (by applying [R-Com], [R-Struct]): $Q' \stackrel{\text{def}}{=} (\nu b\colon (?\top.\mathsf{end}, !\top.\mathsf{end}))(\overline{b}\langle z\rangle \mid b(y) \mid \overline{b}\langle w\rangle)$. This process cannot be accepted since there are two outputs on the same session and it is considered an *error*, as we will introduce later. However, the composition $P \mid Q$ could be typed by assigning to a the unbalanced type $(?(\mathsf{end}, !\top.\mathsf{end}), !(\top))$, which we disallow. In a subtyped world, the situation would be the following: Q sends b at type $\top$ while P does expect a sub-type of $\top$, or the output capability, which is clearly unsound and in turn rejected. We note that unbalanced contexts can still appear in derivations, for increased flexibility. For instance we may have the typing $x\colon (?\top.\mathsf{end}, !T.!\top.\mathsf{end}) \vdash (\nu w\colon T)\overline{x}\langle w\rangle.(\nu z\colon \top)(\overline{x}\langle z\rangle \mid x(y))$.

An important result says that structural congruence preserve typings, in the following sense.

Lemma 3.4 (Preservation of Structural Congruence)**.** *If $P \equiv Q$ and $\Gamma \vdash P$ then $\Gamma \vdash Q$.*

Proof. By induction on the number of applications of $\equiv$. We draw some example. The first case we tackle is when the last rule applied is $P \mid \mathbf{0} \equiv P$. To see the left to the right direction, take $\Gamma \vdash P \mid \mathbf{0}$, $\Gamma_1 \vdash P$, $\Gamma_2 \vdash \mathbf{0}$, and $\Gamma = \Gamma_1 \circ \Gamma_2$. We apply weakening (Corollary 3.3) and infer $\Gamma \vdash P$. For the right to the left direction, the result follows from $\mathscr{T}(\Gamma) \vdash \mathbf{0}$ and Lemma 3.1(3). Now take case $!P \equiv P \mid !P$. For the left to the right direction assume $\Gamma \vdash !P$ inferred from $\mathsf{term}(\Gamma)$ and $\Gamma \vdash P$. We apply Lemma 3.1(1)-(2) and [T-Par] and infer $\Gamma \vdash P \mid !P$. For the right to the left direction, assume $\Gamma \vdash P \mid !P$ inferred from $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash !P$ and $\Gamma = \Gamma_1 \circ \Gamma_2$. We apply weakening (Corollary 3.3) and infer $\Gamma \vdash !P$. The last case we show is rule $(\nu x\colon (A, \overline{A}))\mathbf{0} \equiv \mathbf{0}$. The left to the right direction follows by application of [T-Inact], while the right to the left direction follows by application of [T-Res]. □

Preservation of typability in substitutions is essential to achieve subject reduction.

Lemma 3.5 (Substitution)**.** *If* $\Gamma_1, x\colon T \vdash P$*,* $\Gamma_2(z) = T$ *and* $\Gamma_1 \circ \Gamma_2$ *is defined, then* $\Gamma_1 \circ \Gamma_2 \vdash P[z/x]$*.*

Proof. By induction on the length of the judgement $\Gamma, x\colon T \vdash P$. The proof, although elaborate, is completely standard. □

The following lemma is the wedge of the proof of subject reduction and permits to identify the environment that types the redex.

Lemma 3.6. *If* $\Gamma \vdash P$ *with* Γ *balanced and* $P \xrightarrow{\mu} P'$*, then*

(Case $\mu = x$**):** *there are* T, S *such that*

1. $\Gamma = \Gamma_1, x\colon (?T.S, \overline{?T.S})$ *and* $\Gamma_1, x\colon (S, \overline{S}) \vdash P'$*, or*
2. $\Gamma = \Gamma_1, x\colon (!T.S, \overline{!T.S})$ *and* $\Gamma_1, x\colon (S, \overline{S}) \vdash P'$*, or*
3. $\Gamma = \Gamma_1, x\colon (?T, \overline{?T})$ *and* $\Gamma \vdash P'$*, or*
4. $\Gamma = \Gamma_1, x\colon (!T, \overline{!T})$ *and* $\Gamma \vdash P'$*;*

(Case $\mu = \tau$**):** $\Gamma \vdash P'$*.*

Proof. By induction on $\Gamma \vdash P$. The interesting case is when [T-Par] is used to type a process that does perform [R-Com]: $\overline{x}\langle z\rangle.P \mid x(y).Q \xrightarrow{x} P \mid Q[z/y]$. We have $\Gamma \vdash \overline{x}\langle z\rangle.P \mid x(y).Q$ inferred from [T-Par], $\Gamma = \Gamma_P \circ \Gamma_Q$, $\Gamma_P \vdash \overline{x}\langle z\rangle.P$, and $\Gamma_Q \vdash x(y).Q$. We have the following sub-cases for $\Gamma(x)$ corresponding to (a) $\Gamma(x) = (?T.S, !T.\overline{S})$ and (b) $\Gamma(x) = (!T.S, ?T.\overline{S})$ and (c) $\Gamma(x) = (?T, !T)$ and (d) $\Gamma(x) = (!T, ?T)$. To illustrate, consider (a). From [T-OutS-r] we infer that $\Gamma_P = \Gamma_P^1 \circ (\Gamma_P^2, x\colon (\mathsf{end}, !T.\overline{S})$, $\Gamma_P^1 \vdash z\colon T$, and $\Gamma_P^2, x\colon (\mathsf{end}, \overline{S}) \vdash P$. From [T-InS-l] we know that $\Gamma_Q = \Gamma_Q^1, x\colon (?B.S, \mathsf{end})$, and $\Gamma_Q^1, x\colon (S, \mathsf{end}), y\colon T \vdash Q$. From $\Gamma_P^1 \vdash z\colon T$ and [T-Var] we know $\Gamma_P^1(z) = T$ and $\mathsf{term}(\Gamma_P^1 \backslash z)$. We can easily see that $\Gamma_P^1 \circ \Gamma_Q^1, x\colon (S, \mathsf{end})$ is defined. We apply substitution (Lemma 3.5) and infer $\Gamma_P^1 \circ \Gamma_Q^1, x\colon (S, \mathsf{end}) \vdash Q[z/y]$. An application of [T-Par] give us: $\Gamma_P^2, x\colon (\mathsf{end}, \overline{S}) \circ \Gamma_P^1 \circ \Gamma_Q^1, x\colon (S, \mathsf{end}) \vdash P \mid Q$. The result then follows by noting that the environment above is equal to $\Gamma \backslash x, x\colon (S, \overline{S})$. Sub-case b) is analogous, while sub-cases c),d) follow a similar schema, but are simpler. As a further example, consider the case whether through [T-Par] we infer $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q$ because of $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$, and whether by means of [R-Par] we infer the reduction $P \mid Q \xrightarrow{\mu} P' \mid Q$ because of $P \xrightarrow{\mu} P'$. We apply the I.H. to $\Gamma_1 \vdash P$ and infer that when $\Gamma' \vdash P'$ with Γ' having one of the shape of the statement. Consider the case $(\mu = x)$. We need to show that the split of Γ' and Γ_2 is defined, so that we can apply [T-Par] and conclude. In sub-cases (1,2) we have $\Gamma_2(x) = (\mathsf{end}, \mathsf{end})$ and in turn $\Gamma' \circ \Gamma_2$ defined; in sub-cases (3,4) we have $\Gamma_2(x) = \Gamma_1(x) = \Gamma'(x)$, as desired. We apply [T-Par] and conclude $\Gamma' \circ \Gamma_2 \vdash P' \mid Q$. Otherwise we may have $\mu = \tau$ and $\Gamma' = \Gamma_1$: we apply [T-Par] and infer $\Gamma_1 \circ \Gamma_2 \vdash P' \mid Q$, and we have done. In case [T-Res] the type of the bound channel in the redex is provided by [R-Res] or [R-ResB]. In case [R-Struct] we use Lemma 3.4. The remaining cases are standard. □

We have all the ingredients to prove that balanced typings are preserved by reduction. We let $\Gamma \mapsto \mu$ be the environment that is identified in Lemma 3.6 to type a redex reached by firing μ, and $\Gamma \mapsto \epsilon$ be equal to Γ.

Lemma 3.7. *Let $\Gamma \vdash P$ with Γ balanced. If $P_0 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_n} P_n$ with $n \geq 0$, then $((\Gamma \mapsto \mu_1)\dots) \mapsto \mu_n \vdash P_n$.*

Proof. We proceed by induction on the number n of reductions. The case $n = 0$ is immediate: $\Gamma \vdash P_0$. Otherwise let $n > 0$ and $P_0 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{n-1}} P_{n-1} \xrightarrow{\mu_n} P_n$. By I.H. we have that $((\Gamma \mapsto \mu_1)\dots) \mapsto \mu_{n-1} \vdash P_{n-1}$. To apply Lemma 3.6, we need to ensure that the environment above is balanced. This can be shown by inspecting the shape of $\Gamma \mapsto \mu$. An application of the lemma give us $(((\Gamma \mapsto \mu_1)\dots) \mapsto \mu_{n-1}) \mapsto \mu_n \vdash P_n$. We note that the resulting environment is balanced, and conclude. □

The subject reduction theorem is an immediate consequence of Lemma 3.7.

Theorem 3.8 (Subject Reduction)**.** *If $\Gamma \vdash P$ with Γ balanced and $P \Rightarrow P'$, then $\Gamma' \vdash P'$ with Γ' balanced.*

We now build towards type-safety, or absence of errors. In our language, the monadic pi calculus, errors can be only due to communication. In particular, we repute an error each process that contains two parallel outputs or two parallel inputs on the same session.

Definition 3.9 (Error)**.** *A closed process P is an error whenever*

1. *$P \equiv (\nu x_1 \colon T_1) \cdots (\nu x_n \colon T_n)(\overline{x_i}\langle v\rangle.P_1 \mid \overline{x_i}\langle w\rangle.P_2 \mid Q)$ for some v, w, P_1, P_2 and $i \in 1, \dots, n$ such that T_i is not terminated;*
2. *$P \equiv (\nu x_1 \colon T_1) \cdots (\nu x_n \colon T_n)(x_i(y).P_1 \mid x_i(y).P_2 \mid Q)$ for some P_1, P_2 and $i \in 1, \dots, n$ such that T_i is not terminated.*

The main result of this section says that well-typed processes cannot go wrong.

Theorem 3.10 (Type safety)**.** *If $\vdash P$ and $P \Rightarrow P'$ then P' is not an error.*

Proof. W.l.o.g., let $P' \equiv (\nu x_1 \colon T_1) \cdots (\nu x_n \colon T_n)(Q_1 \mid Q_2)$, where $\mathrm{fv}(Q_1 \mid Q_2) \subseteq \{x_1, \dots, x_n\}$. It is easy to see that $P \Rightarrow P'$ has been inferred from $P \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} P'$, where $n \geq 0$. We use Lemma 3.7 and infer that $\vdash P'$. We need to prove that (i) if $Q_1 = \overline{x_i}\langle v\rangle.P_1 \mid \overline{x_i}\langle w\rangle.P_2$ then $\mathsf{term}(T_i)$ and (ii) if $Q_1 = x_i(y).P_1 \mid x_i(y).P_2$ then $\mathsf{term}(T_i)$. We show (ii), (i) is analogous. By multiple applications of [T-Res] we infer $x_1 \colon T_1, \cdots, x_n \colon T_n \vdash Q_1 \mid Q_2$, and by two applications of [T-Par] we obtain that $x_1 \colon T_1', \cdots, x_n \colon T_n' \vdash Q_1$ and with $T_j = T_j' \circ T_j''$ for all $j \in 1, \dots, n$. Another application of [T-Par] gives us $x_1 \colon A_1, \cdots, x_n \colon A_n \vdash x_i(y).P_1$ and $x_1 \colon B_1, \cdots, x_n \colon B_n \vdash x_i(y).P_2$ where $T_j' = A_j \circ B_j$ for $j \in 1, \dots, n$. Now the assumption $\mathsf{term}(T_i)$ false leads to a contradiction, because this would imply $T_i = (?T.S, \overline{?T.S})$, for some T and S, and in turn $A_i = (?T.S, R_1)$, $B_i = (?T.S, R_2)$, for some R_1 and R_2, which is not deducible by the split rules in Figure 2. □

The result can be transported to open processes by exploiting the lemma below.

Lemma 3.11. *If* $x_1 \colon (R_1, \overline{R_1}), \ldots, x_n \colon (R_n, \overline{R_n}) \vdash P$ *and* $P \Rightarrow P'$ *then*

1. $\vdash (\nu x_1 \colon (R_1, \overline{R_1})) \cdots (\nu x_n \colon (R_n, \overline{R_n}))P$*;*
2. *there are* $U_1, \ldots, U_n$ *such that* $(\nu x_1 \colon (R_1, \overline{R_1})) \cdots (\nu x_n \colon (R_n, \overline{R_n}))P \Rightarrow (\nu x_1 \colon (U_1, \overline{U_1})) \cdots (\nu x_n \colon (U_n, \overline{U_n}))\ P'$.

Proof. To see 1) it suffices to apply [T-Res] n times. 2) follows by multiple applications of [R-ResB]. □

4. Algorithmic type-checking

In this section we present an algorithm that implements the typing system in Figure 2. The main difficulty to tackle is to avoid the use of type and context split, which is inherently non-deterministic. To mimic the split of context into parts, before checking a process we pass the entire context as input to the first typing call and have it return the unused portion as an output. We still have to deal with non-determinism in typing if-then-else processes. In that case, we compute the *join* of the contexts return by the two branches.

We present typing rules for processes of the form $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ where Γ_1 is a context received in input, P is a process received in input, and Γ_2 is a context produced as output; in the rest of the presentation we let the output be what is on the right of the symbol $\triangleright$. An important difference among the type system presented in this section and the one defined in Figure 2 is that the algorithmic rules work up-to balanced contexts: that is, whenever we write $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ we assume that the input process Γ_1 is balanced; we will then show that the return process Γ_2 is balanced as well. For this very reason, rules are deterministic: given an input formed by a balanced context and a process, exactly zero or one rule does match. The algorithmic rules are a declarative presentation of the *patterns* of a function with signature $\mathsf{check}(\mathsf{g} : \mathsf{context}, \mathsf{p} : \mathsf{process}) : \mathsf{context}$: we choose this presentation to ease the notation and to devise clear proofs, and refer the reader to [9] for the details of the implementation of such patterns in *ML*. Note that the top-level call of the function ensures that the context received in input is balanced, otherwise the call immediately aborts.

We start by introducing the rules for variables. Each rule has the form $\Gamma_1 \vdash_{\mathsf{A}} x : T \triangleright \Gamma_2$ where Γ_1, x and T are respectively a context, a variable and a type received in input, and Γ_2 is a context returned in output. In the rules below the output context is obtained by setting each end point type S_i used to

type the variable to type end.

$$\frac{\Gamma = \Gamma_1, x\colon (S_1, S_2), \Gamma_2}{\Gamma \vdash_{\mathsf{A}} x\colon (S_1, \mathsf{end}) \triangleright \Gamma_1, x\colon (\mathsf{end}, S_2), \Gamma_2} \qquad \frac{\Gamma = \Gamma_1, x\colon (S_1, S_2), \Gamma_2}{\Gamma \vdash_{\mathsf{A}} x\colon (\mathsf{end}, S_2) \triangleright \Gamma_1, x\colon (S_1, \mathsf{end}), \Gamma_2} \text{[A-VarS-L],[A-VarS-R]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (S_1, S_2) \triangleright \Gamma_2}{\Gamma \vdash_{\mathsf{A}} x\colon (S_1, S_2) \triangleright \Gamma_1, x\colon \top, \Gamma_2} \qquad \frac{\Gamma = \Gamma_1, x\colon (S_1, S_2) \triangleright \Gamma_2}{\Gamma \vdash_{\mathsf{A}} x\colon \top \triangleright \Gamma} \text{[A-VarS],[A-Var-top]}$$

$$\frac{\Gamma = \Gamma_1, x\colon (E_1, E_2), \Gamma_2}{\Gamma \vdash_{\mathsf{A}} x\colon (E_1, E_2) \triangleright \Gamma} \text{[A-Var]}$$

The rules for typing processes follow below. 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.

$$\Gamma \vdash_{\mathsf{A}} \mathbf{0} \triangleright \Gamma \qquad \frac{\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2 \qquad \Gamma_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3}{\Gamma_1 \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_3} \text{[A-Inact],[A-Par]}$$

Rule [A-Out-L] is used when the entry for the output is the left end point of a type of the form $(!T.S_1, S_2)$. We ignore the right end point S_2 and invoke type checking for the continuation by changing the type of the channel to (S_1, end) through a *context update* operation, noted $+$:

$$\Gamma, x\colon \top + x\colon (S_1, S_2) = \Gamma, x\colon (S_1, S_2)\ .$$

Lastly we change the type of x in the context returned by the call for the continuation by restoring the right end point, that is we return for x the type (end, S_2), which disallows any further use of the left end point in parallel threads. This will will be of help in *reject* deadlocked processes that hold both ends of a session sequentially, as we will show in Section 4.1. Rule [A-Out-L-R] is matched when the output of the session is on the right.

$$\frac{\Gamma_1, x\colon \top \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_2 \qquad \Gamma_2 + x\colon (S_1, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma_3, x\colon (S', S'')}{\Gamma_1, x\colon (!T.S_1, S_2) \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x\colon (\mathsf{end}, S_2)} \text{[A-OutS-L]}$$

$$\frac{\Gamma_1, x\colon \top \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_2 \qquad \Gamma_2 + x\colon (\mathsf{end}, S_2) \vdash_{\mathsf{A}} P \triangleright \Gamma_3, x\colon (S', S'')}{\Gamma_1, x\colon (S_1, !T.S_2) \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_3, x\colon (S_1, \mathsf{end})} \text{[A-OutS-R]}$$

For sending a variable on an channel having a termination end point on the left in rule [A-Out-L] we require the sent variable to be typed by the same context received in input. The context obtained by the typing for the variable is then

used to call the checking function for the continuation process. Rule [A-Out-r], which applies when the termination end point is on the right, describes the same mechanism.

$$\frac{\Gamma_1(x) = (!T, ?T) \qquad \Gamma_1 \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \vdash_{\mathsf{A}} P \triangleright \Gamma_3}{\Gamma_1 \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_3} \qquad \text{[A-Out-l]}$$

$$\frac{\Gamma_1(x) = (?T, !T) \qquad \Gamma_1 \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_2 \qquad \Gamma_2 \vdash_{\mathsf{A}} P \triangleright \Gamma_3}{\Gamma_1 \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_3} \qquad \text{[A-Out-r]}$$

To type an input process we require the expected type to agree with the type of the input channel. We have four rules, [A-InS-l],[A-InS-r],[A-In-l] and [A-In-r], that correspond respectively to the cases whether the end point for the input is a session on the left and on the right, and whether the end point is a termination on the left and on the right. In all rules we check that the type for the sound variable is balanced, otherwise we reject the process. We illustrate rule [A-InS-l]. In this case the type of the input channel is of the form $(?T.S_1, S_2)$, and we invoke type-checking of the continuation by changing the type of x to (S_1, end) and by adding the typing $y\colon T$ for the variable bound by the input. The context returned is pruned from the bound variable, noted $\Gamma\backslash y$, and the type of x is restored to (end, S_2), so that x can be used in parallel threads.

$$\frac{\mathsf{bal}(T) \qquad \Gamma_1, x\colon (S_1, \mathsf{end}), y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (S', S'')}{\Gamma_1, x\colon (?T.S_1, S_2) \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2\backslash y, x\colon (\mathsf{end}, S_2)} \qquad \text{[A-InS-l]}$$

$$\frac{\mathsf{bal}(T) \qquad \Gamma_1, x\colon (\mathsf{end}, S_2), y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (S', S'')}{\Gamma_1, x\colon (S_1, ?T.S_2) \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2\backslash y, x\colon (S_1, \mathsf{end})} \qquad \text{[A-InS-r]}$$

$$\frac{\mathsf{bal}(T) \qquad \Gamma_1(x) = (?T, !T) \qquad \Gamma_1, y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma_2}{\Gamma_1 \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2\backslash y} \qquad \text{[A-In-l]}$$

$$\frac{\mathsf{bal}(T) \qquad \Gamma_1(x) = (!T, ?T) \qquad \Gamma_1, y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma_2}{\Gamma_1 \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2\backslash y} \qquad \text{[A-In-r]}$$

The rule for typing a matching process, [A-If], requires the types of the contrasted variables to be terminated and equal, as in [T-If]. To resolve the nondeterminism of the two branches we build a context to be return by relying on a *join* partial commutative operation over types and contexts with the same domain, noted $\otimes$. The formal definition is below, where we use infix notation and write $\Gamma_1 \otimes \Gamma_2$ to indicate the context $\otimes(\Gamma_1, \Gamma_2)$, whenever defined.

$$S \otimes \mathsf{end} = \mathsf{end} \qquad S \otimes S = S$$
$$(R_1, R_2) \otimes (S_1, S_2) = (R_1 \otimes S_1, R_2 \otimes S_2) \qquad (E_1, E_2) \otimes (E_1, E_2) = (E_1, E_2)$$
$$\emptyset \otimes \emptyset = \emptyset \qquad (\Gamma_1, x\colon T_1) \otimes (\Gamma_2, x\colon T_2) = \Gamma_1 \otimes \Gamma_2, x\colon T_1 \otimes T_2$$

This permits to type processes that do not exhibit the same behaviour in the

two branches, as for instance the process $\mathsf{if}\, x = y \,\mathsf{then}\, P \,\mathsf{else}\, \mathbf{0}$.

$$\frac{\Gamma_1(x) = (E_1, E_2) = \Gamma_1(y) \qquad \Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2 \qquad \Gamma_1 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3}{\Gamma_1 \vdash_{\mathsf{A}} \mathsf{if}\, x = y \,\mathsf{then}\, P \,\mathsf{else}\, Q \triangleright \Gamma_2 \otimes \Gamma_3} \quad \text{[A-If]}$$

The rules for type a process generating a new channel, [A-Res], and replication, [A- Repl], are below. The rule for restriction requires the decoration type to be of the form $(S, \overline{S})$ or $(E, \overline{E})$, as in [T-Res]. 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 of a type in the range of the output context would indicate that an end point of a session has been used or delegated: this must clearly be forbidden under replication.

$$\frac{\Gamma_1, y\colon (A, \overline{A}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2}{\Gamma_1 \vdash_{\mathsf{A}} (\nu y\colon (A, \overline{A}))P \triangleright \Gamma_2 \backslash y} \qquad \frac{\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2 \quad \Gamma_2 = \Gamma_1}{\Gamma_1 \vdash_{\mathsf{A}} !P \triangleright \Gamma_2} \quad \text{[A-Res],[A- Repl]}$$

Example 4.1. *We show an algorithmic derivation for a service that allows to clients to establish a session with the condition that the bootstrap must occur on a specific channel t, which is assume to be trusted. The code is below:*

$$\mathit{Service} \stackrel{def}{=} (\nu s\colon T_s)a(x).\mathsf{if}\, x = t \,\mathsf{then}\, \overline{x}\langle s\rangle.s(y) \,\mathsf{else}\, \overline{\mathit{ack}}\langle \mathit{error}\rangle$$

We identify the following types and contexts, where we let T_s be the type of s, and T_x be the type of both x and t.

$$T_s = (?\top.\mathsf{end}, !\top.\mathsf{end}) \qquad T_x = (?(\mathsf{end}, !\top.\mathsf{end}), !(\mathsf{end}, !\top.\mathsf{end}))$$
$$\Gamma = a\colon (?T_x, !T_x), t\colon T_x, \mathit{ack}\colon (?\bot.\mathsf{end}, !\bot.\mathsf{end}), \mathit{error}\colon \bot$$
$$\Gamma' = a\colon (?T_x, !T_x), t\colon T_x, \mathit{ack}\colon (?\bot.\mathsf{end}, \mathsf{end}), \mathit{error}\colon \bot$$
$$\Gamma'' = \Gamma, s\colon (?\top.\mathsf{end}, \mathsf{end}), x\colon T_x \qquad \Gamma''' = \Gamma, s\colon \top, x\colon T_x$$

The derivation for the then branch is the following.

$$\dfrac{\dfrac{}{\Gamma, s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} s\colon (\mathsf{end}, !\top.\mathsf{end}) \triangleright \Gamma''}\,(\text{[A-VarS-R]}) \qquad \dfrac{\dfrac{}{\Gamma, s\colon \top, x\colon T_x, y\colon \top \vdash_{\mathsf{A}} \mathbf{0} \triangleright \Gamma, s\colon \top, x\colon T_x, y\colon \top}\,(\text{[A-Inact]})}{\Gamma'' \vdash_{\mathsf{A}} s(y) \triangleright \Gamma, s\colon \top, x\colon T_x}\,(\text{[A-InS-L]})}{\Gamma, s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \overline{x}\langle s\rangle.s(y) \triangleright \Gamma, s\colon \top, x\colon T_x}\,(\text{[A-Out-L]})$$

The derivation for the else branch is below.

$$\dfrac{\dfrac{}{\Gamma', s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \mathit{error}\colon \bot \triangleright \Gamma', s\colon T_s, x\colon T_x}\,(\text{[A-Var]}) \qquad \dfrac{}{\Gamma', s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \mathbf{0} \triangleright \Gamma', s\colon T_s, x\colon T_x}\,(\text{[A-Inact]})}{\Gamma, s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \overline{\mathit{ack}}\langle \mathit{error}\rangle \triangleright \Gamma', s\colon T_s, x\colon T_x}\,(\text{[A-Out-L-R]})$$

Incidentally, we note that $(\Gamma, s\colon \top, x\colon T_x) \otimes (\Gamma', s\colon T_s, x\colon T_x) = \Gamma', s\colon \top, x\colon T_x$.
We conclude with the derivation for the service.

$$\dfrac{\dfrac{\dfrac{\Gamma, s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \overline{x}\langle s\rangle.s(y) \triangleright \Gamma, s\colon \top, x\colon T_x \qquad \Gamma, s\colon T_s, x\colon T_x \vdash_{\mathsf{A}} \overline{\mathit{ack}}\langle \mathit{error}\rangle \triangleright \Gamma', s\colon T_s, x\colon T_x}{\Gamma, s\colon T_s, x\colon T_x \vdash \mathsf{if}\, x = t \,\mathsf{then}\, \overline{x}\langle s\rangle.s(y) \,\mathsf{else}\, \overline{\mathit{ack}}\langle \mathit{error}\rangle \triangleright \Gamma', s\colon \top, x\colon T_x}\,(\text{[A-If]})}{\Gamma, s\colon T_s \vdash a(x).\mathsf{if}\, x = t \,\mathsf{then}\, \overline{x}\langle s\rangle.s(y) \,\mathsf{else}\, \overline{\mathit{ack}}\langle \mathit{error}\rangle \triangleright \Gamma', s\colon \top}\,(\text{[A-In-L]})}{\Gamma \vdash \mathit{Service} \triangleright \Gamma'}\,(\text{[A-Res]})$$

4.1. Properties of the algorithmic system

The first part of this section is devoted to establishing the soundness of the algorithm, that is: processes accepted by the algorithm are typed by the system in Figure 2. In the second part we show a result of partial completeness: if a process is typable and does not contain a *Wait for* deadlock then it is accepted by the algorithm.

Soundness. To tackle the soundness result, we project the pattern rules presented in Section 4 into the typing system of Figure 2. We introduce preliminary lemmas and definitions that will be useful to prove the main result of this section. The first property says that the algorithmic system preserves the balancing of contexts, in the following sense.

Lemma 4.2. *If* Γ_1 *is balanced and* $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ *then* $\mathrm{dom}(\Gamma_2) = \mathrm{dom}(\Gamma_1)$ *and* Γ_2 *balanced.*

Proof. By induction on $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$, eventually relying on a similar result for values. To illustrate, take [A-OutS-L] and let $\Gamma_1, x\colon (!T.S, \mathsf{end}) \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_3$ be inferred from $\Gamma_1, x : \top \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_2$ and $\Gamma_2 + x\colon (S, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma_3$. Assume $\Gamma_1, x\colon (!T.S, \mathsf{end})$ balanced. From the top type balanced we infer $\Gamma_1, x : \top$ balanced. It's easy to see that this implies $\mathrm{dom}(\Gamma_2) = \mathrm{dom}(\Gamma_1, x : \top)$ with Γ_2 balanced, and $\Gamma_2(x) = \top$. We can apply induction on $\Gamma_2 + x\colon (S, \mathsf{end}) = (\Gamma_2 \backslash x), x : (S, \mathsf{end})$ and infer that $\mathrm{dom}(\Gamma_2 + x\colon (S, \mathsf{end})) = \mathrm{dom}(\Gamma_3)$ with Γ_3 balanced. The result then follows by transitivity. As a further example, consider case [A-Par] and assume that $\Gamma_1 \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_3$ has been inferred from $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ and $\Gamma_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3$. The I.H. says that $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ with Γ_2 balanced. We apply the I.H. to $\Gamma_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3$ and obtain $\mathrm{dom}(\Gamma_2) = \mathrm{dom}(\Gamma_3)$ with Γ_3 balanced. □

We then establish a result similar to Lemma 3.2.

Lemma 4.3 (Algorithmic Weakening)**.** *Let* Γ_1 *be balanced.*

1. *If* $\Gamma_1, x\colon (S, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (S', \mathsf{end})$ *then* $\Gamma_1, x\colon (S, \overline{S}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (S', \overline{S})$*;*
2. *If* $\Gamma_1, x\colon (\mathsf{end}, S) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (\mathsf{end}, S')$ *then* $\Gamma_1, x\colon (\overline{S}, S) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (\overline{S}, S')$.

Proof. By induction on the length of the inference, eventually relying on a similar result for values. To see an example of 1), consider rule [A-InS-L]: $\Gamma_1, x\colon (?T.S, \mathsf{end}) \vdash_{\mathsf{A}} x(y).P \triangleright (\Gamma_2 \backslash y), x : (\mathsf{end}, \mathsf{end})$ inferred from $(*)\, \Gamma_1, x\colon (S, \mathsf{end})$, $y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x : (S_1, \mathsf{end})$. We apply [A-InS-L] to $(*)$ and infer the desired result, $\Gamma_1, x\colon (?T.S, \overline{?T.S}) \vdash_{\mathsf{A}} x(y).P \triangleright (\Gamma_2 \backslash y), x : (\mathsf{end}, \overline{?T.S})$. As further example, consider [A-Par]: $\Gamma_1, x\colon (S, \mathsf{end}) \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_2, x\colon (S_2, \mathsf{end})$ inferred from $(**)\, \Gamma_1, x\colon (S, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma'$ and $(***)\, \Gamma' \vdash_{\mathsf{A}} Q \triangleright \Gamma_2, x\colon (S_1, \mathsf{end})$. We easily see that $\Gamma' = \Gamma'', x\colon (S_1, \mathsf{end})$. We apply the I.H. to $(**)$ and infer $\Gamma_1, x\colon (S, \overline{S}) \vdash_{\mathsf{A}} P \triangleright \Gamma'', x\colon (S_1, \overline{S})$. Next we apply the I.H. to $(***)$ and obtain $\Gamma'', x\colon (S_1, \overline{S}) \vdash_{\mathsf{A}} Q \triangleright \Gamma_2, x\colon (S_2, \overline{S})$. An application of [A-Par] gives us the desired result, $\Gamma_1, x\colon (S, \overline{S}) \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_2, x\colon (S_2, \overline{S})$. □

By repeated applications of the lemma above we obtain the following useful result.

Corollary 4.4. *Let* $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ *with* Γ_1 *balanced. If* $\Gamma_1 \circ \Gamma_3$ *is defined and balanced, then* $\Gamma_1 \circ \Gamma_3 \vdash_{\mathsf{A}} P \triangleright \Gamma_2 \circ \Gamma_3$.

Next, we note that in a type derivation $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$, the session end points contained in Γ_2 have not been used to type P (otherwise they would have been set to end). We identify the environment that is sufficient to type P by means of the notion of *used closure* defined below. Given a balanced judgement $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ we let the used closure of 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:

$$S \triangleright S = \mathsf{end} \qquad S \triangleright \mathsf{end} = S$$
$$(S_1, S_2) \triangleright (S', S'') = (S_1 \triangleright S', S_2 \triangleright S'') \qquad (E_1, E_2) \triangleright (E_1, E_2) = (E_1, E_2)$$

Lemma 4.5. *If* $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ *with* Γ_1 *balanced then* $\Gamma_1 \triangleright \Gamma_2$ *is defined.*

Proof. By induction on $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$. In the output cases we use a similar result for values. As an example, consider case [A-InS-L]. We have $\Gamma_1, x\colon (?T.S_1, S_2) \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2, x\colon (\mathsf{end}, S_2)$ inferred from $\Gamma_1, x\colon (S_1, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon (S', S'')$. By I.H. we have $\Gamma_1, x\colon (S_1, \mathsf{end}) \triangleright \Gamma_2, x\colon (S', S'')$ defined. From this we infer $\Gamma_1 \triangleright \Gamma_2$ defined. We conclude by noting that $(?T.S_1, S_2) \triangleright (\mathsf{end}, S_2) = (\mathsf{end}, S_2)$. Consider now case [A-If] and assume $\Gamma_1 \vdash_{\mathsf{A}} \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q \triangleright \Gamma_2 \otimes \Gamma_3$ be inferred from $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ and $\Gamma_1 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3$ where $\Gamma_1(x) = (E_1, E_2) = \Gamma_1(y)$. By I.H. $\Gamma_1 \triangleright \Gamma_2$ and $\Gamma_1 \triangleright \Gamma_3$ are defined. Let $\Gamma = \Gamma_2 \otimes \Gamma_3$. We need to show that $\Gamma_1 \triangleright \Gamma$ is defined. Whenever $\Gamma_1 = \emptyset$ this is true by hypothesis. Otherwise take $\Gamma_1 = \Gamma', z : T$. We proceed by case analysis and show that the I.H. imply that $T \triangleright \Gamma(z)$ is defined. Whenever $T = (E_1, E_2)$ we have $\Gamma_2(z) = T = \Gamma_3(z)$, and in turn $\Gamma(z) = T$, $T \triangleright T = T$, and the result follows. Otherwise let $T = (R_1, R_2)$, $\Gamma_2(z) = (S_1, S_2)$, $\Gamma_3(z) = (U_1, U_2)$, $\Gamma(z) = (V_1, V_2)$. For each R_i, $i = 1, 2$, we have that one of the following cases holds, where we let A be of the form S with $S \neq \mathsf{end}$: (a) $R_i = A \wedge S_i = A \wedge U_i = A \wedge V_i = A$, or (b) $R_i = A \wedge S_i = \mathsf{end} \wedge U_i = A \wedge V_i = \mathsf{end}$, or (c) $R_i = A \wedge S_i = A \wedge U_i = \mathsf{end} \wedge V_i = \mathsf{end}$, or (d) $R_i = A \wedge S_i = \mathsf{end} \wedge U_i = \mathsf{end} \wedge V_i = \mathsf{end}$. In all cases we have $R_i \triangleright V_i$ defined and in turn $T \triangleright (V_1, V_2)$ defined, as desired. □

An used closure generated by the algorithmic system is sufficient to type a process with the system $\vdash$, as we show below. The following lemma is the wedge of the proof of soundness (cf. Theorem 4.7).

Lemma 4.6. *Let* Γ_1 *be balanced.*

1. *If* $\Gamma_1 \vdash_{\mathsf{A}} z\colon T \triangleright \Gamma_2$ *then* $\Gamma_1 \triangleright \Gamma_2 \vdash z\colon T$*;*
2. *If* $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ *then* $\Gamma_1 \triangleright \Gamma_2 \vdash P$.

Proof. For the first item, we proceed by a case analysis and obtain that there are Γ', T' and T'' such that $\Gamma_1 = \Gamma', z\colon T'$, $\Gamma_2 = \Gamma', z\colon T''$, and $T' \triangleright T'' = T$. We conclude by $\mathsf{un}(\Gamma' \triangleright \Gamma')$ and [T-Var]. For the second item, we proceed by induction on the length of the derivation. We prove the most interesting cases. Assume case [A-Par] holds and let $\Gamma_1 \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_3$ be inferred from $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ and $\Gamma_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3$. By I.H. we have $\Gamma_1 \triangleright \Gamma_2 \vdash P$ and $\Gamma_2 \triangleright \Gamma_3 \vdash Q$. To conclude by applying [T-Par] we need to show that $(\Gamma_1 \triangleright \Gamma_2) \circ (\Gamma_2 \triangleright \Gamma_3)$ is defined and equal to $\Gamma_1 \triangleright \Gamma_3$. If $\Gamma_1 = \emptyset$ we are done by applying the first rule for context split. Otherwise assume $x \in \mathrm{dom}(\Gamma_1)$. We exploit Lemma 4.5 in order to infer the types $\Gamma_2(x)$ and $\Gamma_3(x)$. Whenever $\Gamma_1(x) = (E_1, E_2)$ we have $\Gamma_2(x) = \Gamma_1(x) = \Gamma_3(x)$, because of Lemma 4.5. We then conclude by applying the split rule for termination types in Figure 2. Now assume $\Gamma_1(x) = (S, \mathsf{end})$. Lemma 4.5 implies that we have two cases for $\Gamma_2(x)$ corresponding to (i) (S, end) and (ii) $\top$. In case (i) we have $(\Gamma_1 \triangleright \Gamma_2)(x) = \top$ and two sub-cases for $\Gamma_3(x)$ corresponding to (i.a) (S, end) and (i.b) $\top$. In case (i.a) we have $(\Gamma_2 \triangleright \Gamma_3)(x) = \top$ and in case (i.b) we have $(\Gamma_2 \triangleright \Gamma_3)(x) = (S, \mathsf{end})$. In both sub-cases we have $\top \circ ((\Gamma_2 \triangleright \Gamma_3)(x)) = (\Gamma_1 \triangleright \Gamma_3)(x)$. In case (ii) we have $(\Gamma_1 \triangleright \Gamma_2)(x) = (S, \mathsf{end})$ and $\Gamma_3(x) = \top$, and in turn $(\Gamma_2 \triangleright \Gamma_3)(x) = \top$. From this we infer $(S, \mathsf{end}) \circ \top = (S, \mathsf{end}) = \Gamma_1(x) \triangleright \Gamma_3(x)$. The cases $\Gamma_1(x) = (\mathsf{end}, S)$ and $\Gamma_1(x) = (S_1, S_2)$ are similar. Consider now case [A-If], and let $\Gamma_1 \vdash_{\mathsf{A}} \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q \triangleright \Gamma$ be inferred from $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$ and $\Gamma_1 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3$ where $\Gamma_1(x) = (E_1, E_2) = \Gamma_1(y)$ and $\Gamma = \Gamma_2 \otimes \Gamma_3$. We need to show that $\Gamma_1 \triangleright \Gamma \vdash \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q$. By I.H we have $\Gamma_1 \triangleright \Gamma_2 \vdash P$ and $\Gamma_1 \triangleright \Gamma_3 \vdash Q$. We claim that there are Γ' and Γ'' such that $\Gamma_1 \triangleright \Gamma = (\Gamma_1 \triangleright \Gamma_2) \circ \Gamma'$ and $\Gamma_1 \triangleright \Gamma = (\Gamma_1 \triangleright \Gamma_3) \circ \Gamma''$. The result then follows by apply weakening, Corollary 3.3, followed by [T-If]. To justify the claim, take $\Gamma_1 \triangleright \Gamma_2$. This environment could differ from $\Gamma_1 \triangleright \Gamma$ in a number of end points in the range of Γ_2 and Γ that are equal to S in the former and to end in the latter; this is deduced by Lemma 4.5, and by definition of $\otimes$. Therefore such end points are equal to end in $\Gamma_1 \triangleright \Gamma_2$ and to S in $\Gamma_1 \triangleright \Gamma$. We then build Γ' as follows: $\Gamma_1 \triangleright \Gamma_2(x) = (S_1, S_2)$ and $\Gamma_1 \triangleright \Gamma(x) = (R_1, R_2)$ imply $\Gamma'(x) = (U_1, U_2)$ where $S_i = R_i \circ U_i$ for each $i = 1, 2$. The case $\Gamma_1 \triangleright \Gamma_3$ is analogous. □

By relying on the lemma above we establish the main result of this section.

Theorem 4.7 (Soundness)**.** *If* $\Gamma \vdash_{\mathsf{A}} P \triangleright \Gamma'$ *with* Γ *balanced then* $\Gamma \vdash P$.

Proof. Let $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2$. By Lemma 4.6 we have $\Gamma_1 \triangleright \Gamma_2 \vdash P$. Contexts Γ_1 and $\Gamma_1 \triangleright \Gamma_2$ could differ in a number of end points that are of the form S in the range of the former and have been set to end in the range of the latter, because they are not used. We build a context Γ_3 such that $\Gamma_1 = (\Gamma_1 \triangleright \Gamma_2) \circ \Gamma_3$ as follows: $\Gamma_1(x) = (S_1, S_2)$ and $\Gamma_1 \triangleright \Gamma_2(x) = (R_1, R_2)$ implies $\Gamma_3(x) = (U_1, U_2)$ with $S_i = R_i \circ U_i$, for $i = 1, 2$. We apply weakening (Corollary 3.3) to $\Gamma_1 \triangleright \Gamma_2 \vdash P$ and conclude $\Gamma_1 \vdash P$, as desired. □

Completeness for deadlock-free processes. We now show that we accept an important class of typed processes, or processes not containing *Wait for* dead-

[T-OutS-L]	$\mathsf{dead}(\Gamma, x\colon (S_1, S_2) \vdash \overline{x}\langle y\rangle.P)$	$\Gamma, x\colon (S_1, \mathsf{end}) \vdash \overline{x}\langle y\rangle.P \ \rightarrow$ $\mathsf{dead}(\Gamma' \vdash P)$
[T-OutS-R]	$\mathsf{dead}(\Gamma, x\colon (S_1, S_2) \vdash \overline{x}\langle y\rangle.P)$	$\Gamma, x\colon (\mathsf{end}, S_2) \vdash \overline{x}\langle y\rangle.P \ \rightarrow$ $\mathsf{dead}(\Gamma' \vdash P)$
[T-Out-L]	$\mathsf{dead}(\Gamma, x\colon (E_1, E_2) \vdash \overline{x}\langle y\rangle.P)$	$\mathsf{dead}(\Gamma' \vdash P)$
[T-Out-R]	$\mathsf{dead}(\Gamma, x\colon (E_1, E_2) \vdash \overline{x}\langle y\rangle.P)$	$\mathsf{dead}(\Gamma' \vdash P)$
[T-InS-L]	$\mathsf{dead}(\Gamma, x\colon (S_1, S_2) \vdash x(y).P)$	$\Gamma, x\colon (S_1, \mathsf{end}) \vdash x(y).P \ \rightarrow$ $\mathsf{dead}(\Gamma' \vdash P)$
[T-InS-R]	$\mathsf{dead}(\Gamma, x\colon (S_1, S_2) \vdash x(y).P)$	$\Gamma, x\colon (\mathsf{end}, S_2) \vdash x(y).P \ \rightarrow$ $\mathsf{dead}(\Gamma' \vdash P)$
[T-In-L]	$\mathsf{dead}(\Gamma, x\colon (E_1, E_2) \vdash x(y).P)$	$\mathsf{dead}(\Gamma' \vdash P)$
[T-In-R]	$\mathsf{dead}(\Gamma, x\colon (E_1, E_2) \vdash x(y).P)$	$\mathsf{dead}(\Gamma' \vdash P)$
[T-Repl]	$\mathsf{dead}(\Gamma \vdash !P)$	$\mathsf{dead}(\Gamma \vdash P)$
[T-Res]	$\mathsf{dead}(\Gamma \vdash (\nu x\colon T)P)$	$\mathsf{dead}(\Gamma, x\colon T \vdash P)$
[T-If]	$\mathsf{dead}(\Gamma \vdash \mathsf{if}\, x = y\, \mathsf{then}\, P\, \mathsf{else}\, Q)$	$\mathsf{dead}(\Gamma \vdash P) \ \vee \ \mathsf{dead}(\Gamma \vdash Q)$
[T-Par]	$\mathsf{dead}(\Gamma_1 \circ \Gamma_2 \vdash P \mid Q)$	$\mathsf{dead}(\Gamma_1 \vdash P) \ \vee \ \mathsf{dead}(\Gamma_2 \vdash Q)$

Table 1: Deadlocked predicate

locks [2][1]: an input and its corresponding output (or vice-versa) are in the same thread instead of in parallel ones. We rely on the help of types to try to infer this kind of errors made by the programmer in coding sessions, and design a type-checking algorithm that reject these processes. Moreover, we are studying a procedure to disentangle such form of deadlocks; we will discuss further this point in the conclusions (cf. Section 6).

We start by introducing the notion of *Wait for* deadlock. Given a typing derivation $\Gamma \vdash P$, we let the *deadlocked* predicate, noted $\mathsf{dead}(\Gamma \vdash P)$, be defined inductively on the structure of proof trees as in Table 1. The definition formalizes an easy concept: a process is deadlocked whenever it contains a prefix x that cannot be typed by using *only* one end point of a session type: that is, if $x(y).P$ or $\overline{x}\langle y\rangle.P$ require x to have a type of the form (S_1, S_2) with both S_1 and S_2 distinct from the type end, then they are deadlocked. The first column of Table 1 contains the label of the last rule applied in $\Gamma \vdash P$, the second column contains the predicate $\mathsf{dead}(\Gamma \vdash P)$, and the third column contains the condition to be satisfied, stated as a standard logic formula. In the third column of the first and second line, if there exists a derivation $\Gamma_1 \vdash \overline{x}\langle y\rangle.P$, for the identified Γ_1, then $\Gamma' \vdash P$ is the judgement appearing as premise in that derivation. Similarly, in the third column of the fifth and sixth line, $\Gamma' \vdash P$ is the premise of the judgement $\Gamma_1 \vdash x(y).P$ appearing in the left of the implication. In contrast, in the third column of the third, fourth, seventh and eighth line the judgement $\Gamma' \vdash P$ is the premise of $\Gamma, x\colon (E_1, E_2) \vdash R$, that is the proof tree appearing as

[1]This is one of the fourth conditions identified in the paper for a deadlock to occur. In the pi-calculus setting, the *Wait for* condition is sufficient to block a single thread.

subject of the deadlocked predicate in the second column, for the identified R.

We refine the class of typed processes accepted by the algorithm by introducing the notion of *strongly balanced* type and context. A balanced type (A_1, A_2) is strongly balanced whenever all types $T_1, \ldots T_n$ occurring as input argument in A_1 and A_2 are balanced. Strongly balanced contexts contain in their range only strongly balanced types. We note that an input process that waits for an unbalanced variable is useless since it cannot receive such variable from a balanced process, which sends in output only free or bound balanced variables.

The following theorem establishes our partial completeness result. The strong balanced assumption is essential to obtain a typing correspondence in the algorithm, which rejects non-balanced bound input variables and contexts.

Theorem 4.8 (Completeness)**.** *If* $\Gamma \vdash P$ *with* Γ *strongly balanced and not* $\mathsf{dead}(\Gamma \vdash P)$*, then* $\Gamma \vdash_{\mathsf{A}} P \triangleright \Gamma'$*, for some* Γ'*.*

Proof. By induction on $\Gamma \vdash P$. Take case [T-InS-L] and assume that $\Gamma, x\colon (?T.S, R) \vdash x(y).P$ is inferred from $\Gamma, x\colon (S, R), y\colon T \vdash P$. We use the logical equivalence $\neg(A_1 \to A_2) = A_1 \wedge \neg A_2$ and reformulate the hypothesis as: $\Gamma, x\colon (?T.S, \mathsf{end}) \vdash P$ and not $\mathsf{dead}(\Gamma, x\colon (S, \mathsf{end}), y\colon T \vdash P)$. From $\Gamma, x\colon (?T.S, R)$ strongly balanced we obtain $\Gamma, x\colon (?T.S, \mathsf{end}), y\colon T$ strongly balanced, since the types occurring as input argument in T are contained in $?T.S$ as well. We apply the I.H. and infer $\Gamma_1, x\colon (S, \mathsf{end}), y\colon T \vdash_{\mathsf{A}} P \triangleright \Gamma', y\colon T'$. We conclude by applying [A-InS-L]: $\Gamma, x\colon (?T.S, R) \vdash x(y).P \triangleright \Gamma' \backslash x, x\colon (\mathsf{end}, R)$. Case [T-InS-R] is similar, while cases [T-In-L] and [T-In-R] follow directly by induction. Consider case [T-OutS-L] and let $\Gamma \vdash \overline{x}\langle y\rangle.P$ with $\Gamma = \Gamma_1 \circ (\Gamma_2, x\colon (!T.S, R)$ be inferred from $\Gamma_1 \vdash y\colon T$ and $\Gamma_2, x\colon (S, R) \vdash P$. From [T-Var] we have $\Gamma_1(x) = \top$, and by hypothesis $\Gamma_1 \circ (\Gamma_2, x\colon (!T.S, \mathsf{end}) \vdash \overline{x}\langle y\rangle.P$ and not $\mathsf{dead}(\Gamma_2, x\colon (S, \mathsf{end}) \vdash P)$. We apply the I.H. and infer $\Gamma_2, x\colon (S, \mathsf{end}) \vdash_{\mathsf{A}} P \triangleright \Gamma_3$. We find Γ', Γ'' such that $\Gamma_1 \circ (\Gamma_2, x\colon (!T.S, \mathsf{end}) \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma'$, and $\Gamma' = \Gamma_2, x\colon (S, \mathsf{end}) \circ \Gamma''$, and we weaken (Corollary 4.4) the judgement to $\Gamma' \vdash_{\mathsf{A}} P \triangleright \Gamma_3 \circ \Gamma''$. We conclude by applying [A-OutS-L]: $\Gamma \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright (\Gamma_3 \circ \Gamma'')\backslash x, x\colon (\mathsf{end}, R)$. Case [T-OutS-R] is analogous, while cases [T-Out-L] and [T-OutS-R] follow directly from induction. The remaining cases are a direct consequence of the I.H. As an example, consider case [T-Par] and let $\Gamma \vdash P \mid Q$ be inferred from $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$ where $\Gamma = \Gamma_1 \circ \Gamma_2$. Since $\mathsf{dead}(\Gamma \vdash P \mid Q)$ is false, we know that $\mathsf{dead}(\Gamma_1 \vdash P)$ and $\mathsf{dead}(\Gamma_2 \vdash Q)$ are false as well. We also know that Γ_1 and Γ_2 are strongly balanced; this can be shown by induction on the size of $\mathrm{dom}(\Gamma)$. We apply the I.H. and obtain $\Gamma_1 \vdash_{\mathsf{A}} P \triangleright \Gamma'$ and $\Gamma_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma''$. We apply weakening (Corollary 4.4) and obtain $\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{A}} P \triangleright \Gamma' \circ \Gamma_2$ and $\Gamma_2 \circ \Gamma' \vdash_{\mathsf{A}} Q \triangleright \Gamma'' \circ \Gamma'$. We note that $\Gamma' \circ \Gamma_2$ and $\Gamma_2 \circ \Gamma'$ are equal and balanced; this follows from the definition of context split and from Lemma 4.2. We apply [A-Par] and conclude: $\Gamma \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma'' \circ \Gamma'$. □

Remark 4.9. *A key property of the type system in Section 2 is that structural congruence preserve typings (Lemma 3.4). By gluing Theorem 4.7, Lemma 3.4, and Theorem 4.8 we have a similar result for the algorithmic system: if* Γ *balanced and* $\Gamma \vdash_{\mathsf{A}} P \triangleright \Gamma'$ *(which in turn implies* Γ *strongly balanced), and* $P \equiv Q$*,*

and not $\mathsf{dead}(\Gamma \vdash Q)$*, then* $\Gamma \vdash_{\mathsf{A}} Q \triangleright \Gamma''$*. We conjecture that the result holds in general; the proof we envision is based on the one contained in [9]. We leave this for future work.*

5. Expressiveness

Having defined a typed theory, and implemented it in a type checking algorithm, it remains to investigate the expressiveness of the theory itself. In this section we identify a fragment of the linear π-calculus of [13] that can be encoded in the typed π-calculus of Section 2 while preserving both typings and reductions. The system in [13] does feature (a) linear types that evolve to unrestricted types and (b) recursive types. The fragment of our interest does not permit the input/output use of channels typed with (b) (while those channels can be passed around), and do not feature linear recursive types of (b) (while limited unrestricted recursion is allowed). We are interested in this fragment because it corresponds with the image of the encoding of variants of π-calculus that feature session [7] and linear [19] types, as we discuss at the end of the section. However, we do not see difficulties in extending the system in Section 2 by considering recursive types, while the combination of a) with the analysis of the if-then-else processes of our interest appears to be challenging. We will return on this point in the conclusions (cf. Section 6).

Source language. The syntax of the source language and its typing rules are in Figure 3. The differences with respect the syntax of processes in Figure 1 are: (1) restriction is not decorated and (2) processes can send, receive and test boolean constants. The syntax of types L follow, where an end point is *qualified* as linear (lin) or unrestricted (un). Recursive types are limited to the form $\mu a.\,\mathsf{un}?L.U$ and $\mu a.\,\mathsf{un}!L.U$, where a is a type variable, that is they are unrestricted. End point type duality is as expected:

$$\overline{\mathsf{lin}\,?L.N} = \mathsf{lin}\,!L.\overline{N} \qquad \overline{\mathsf{lin}\,!L.N} = \mathsf{lin}\,?L.\overline{N} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{\mu a.\,\mathsf{un}?L.U} = \mu a.\,\overline{\mathsf{un}?L.U} \qquad \overline{\mu a.\,\mathsf{un}!L.U} = \mu a.\,\overline{\mathsf{un}!L.U} \qquad \overline{a} = a$$

We let the term predicate to hold for unrestricted and boolean types: $\mathsf{term}(\mathsf{bool})$, $\mathsf{term}(N_1)$ and $\mathsf{term}((N_1, N_2))$ whenever N_i is not of the form $\mathsf{lin}\,?L.N$ or $\mathsf{lin}\,!L.N$. We use the same *balanced* predicate introduced in Section 2. We also use the predicate un: $\mathsf{un}(U)$ and $\mathsf{un}((U_1, U_2))$. We use Δ to range over contexts assigning types of the form L to variables x.

In the second part of Figure 3 we outline the (left) rules for split entries having linear and mixed types; the rules for booleans and unrestricted types are those of channel types (E_1, E_2) in Figure 2. The typing rules for processes are below in the same figure. Rule [Tl-Inact] requires the environment to be terminated, so to enforce the linear discipline. Note also conditions $(*)$ and $(**)$ in typing input and output channels, so that (1) we do not permit linear types to evolve to unrestricted types, and (2) we exclude unsound unrestricted

Syntax of typed processes

L	$::=$	Types	K	$::=$	Processes
	(N_1, N_2)	pair		$\overline{x}\langle v\rangle.K$	output
	N	single		$x(y).K$	input
	bool	boolean		$(\nu x)K$	restriction
N	$::=$	End point		$\mathsf{if}\, v\, \mathsf{then}\, K_1\, \mathsf{else}\, K_2$	conditional
	$\mathsf{lin}\,?T.S$	input		$(K_1 \mid K_2)$	composition
	$\mathsf{lin}\,!T.S$	output		$!K$	replication
	end	termination		$\mathbf{0}$	inaction
	U	unrestricted	v	$::=$	Values
U	$::=$	Recursion		$\mathsf{true}, \mathsf{false}$	constant
	$\mu a.\, \mathsf{un}?L.U$	input		x, y	variable
	$\mu a.\, \mathsf{un}!L.U$	output			
	a	type variable			

Context split rules

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = N \text{ or } (N_1, N_2)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon (N_1, N_2) = (\Gamma_1, x\colon N_1) \circ (\Gamma_2, x\colon N_2)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon (N_1, U_2) = (\Gamma_1, x\colon (N_1, U_2)) \circ (\Gamma_2, x\colon U_2)}$$

Typing rules

$$\frac{\mathsf{term}(\Delta)}{\Delta \vdash \mathbf{0}} \qquad \frac{\mathsf{bal}(N) \qquad \Delta, x\colon (N, \overline{N}) \vdash K}{\Delta \vdash (\nu x)K} \qquad [\textsc{Tl-Inact}],[\textsc{Tl-Res}]$$

$$\frac{\Delta_1 \vdash v\colon \mathsf{bool} \qquad \Delta_2 \vdash K_1 \qquad \Delta_2 \vdash K_2}{\Delta_1 \circ \Delta_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, K_1\, \mathsf{else}\, K_2} \qquad [\textsc{Tl-If}]$$

$$\frac{\Delta, x\colon N, y\colon L \vdash K \qquad (*)}{\Delta, x\colon q?L.N \vdash x(y).K} \qquad \frac{\Delta_1 \vdash v\colon L \qquad \Delta_2, x\colon N \vdash H \qquad (**)}{\Delta_1 \circ (\Delta_2, x\colon q\,!L.N) \vdash \overline{x}\langle v\rangle.K} \qquad [\textsc{Tl-In}],[\textsc{Tl-Out}]$$

$$\frac{\Delta, x\colon (N, N'), y\colon L \vdash K \qquad (*)}{\Delta, x\colon (q?L.N, N') \vdash x(y).K} \qquad \frac{\Delta_1 \vdash v\colon L \qquad \Delta_2, x\colon (N, N') \vdash K \qquad (**)}{\Delta_1 \circ (\Delta_2, x\colon (q\,!L.N, N')) \vdash \overline{x}\langle v\rangle.K} \qquad [\textsc{Tl-InC}],[\textsc{Tl-OutC}]$$

$$(*)\ \ q = \mathsf{lin} \Rightarrow \neg\mathsf{un}(N) \text{ and } \ q = \mathsf{un} \Rightarrow q?L.N = N$$

$$(**)\ \ q = \mathsf{lin} \Rightarrow \neg\mathsf{un}(N) \text{ and } q = \mathsf{un} \Rightarrow q!L.N = N \wedge v \neq x$$

Figure 3: Source language: linear π-calculus

Encoding of types

$$\begin{array}{rcl@{\qquad}rcl}
[\![\,\mathsf{lin}\,?L.N\,]\!] & \stackrel{\text{def}}{=} & ?(\!(L)\!).[\![\,N\,]\!] & [\![\,\mathsf{lin}\,!L.N\,]\!] & \stackrel{\text{def}}{=} & !(\!(L)\!).[\![\,N\,]\!] \\
[\![\,\mu a.\,\mathsf{un}?L.U\,]\!] & \stackrel{\text{def}}{=} & \mathsf{end} & [\![\,\mu a.\,\mathsf{un}!L.U\,]\!] & \stackrel{\text{def}}{=} & \mathsf{end} \\
[\![\,\mathsf{end}\,]\!] & \stackrel{\text{def}}{=} & \mathsf{end} & (\!(\mathsf{bool})\!) & \stackrel{\text{def}}{=} & \bot \\
(\!((N_1,N_2))\!) & \stackrel{\text{def}}{=} & ([\![\,N_1\,]\!],[\![\,N_2\,]\!])\ (*) & (\!(N)\!) & \stackrel{\text{def}}{=} & ([\![\,N\,]\!],\mathsf{end})\ (*) \\
(\!((U_1,U_2))\!) & \stackrel{\text{def}}{=} & (?(\!(L_1)\!),!(\!(L_2)\!))\ (**) & (\!((U_2,U_1))\!) & \stackrel{\text{def}}{=} & (!(\!(L_2)\!),?(\!(L_1)\!))\ (**) \\
(\!(U)\!) & \stackrel{\text{def}}{=} & (\!((U,\overline{U}))\!) & & & \\
\end{array}$$

$$(*)\ N \neq U \qquad (**)\ U_1 = ?L_1.U_1 \text{ and } U_2 = !L_2.U_2$$

Encoding of contexts

$$(\!(\emptyset)\!) \stackrel{\text{def}}{=} \{true\colon \bot, false\colon \bot\} \qquad (\!(\Gamma, x\colon L)\!) \stackrel{\text{def}}{=} (\!(\Gamma)\!), x\colon (\!(L)\!)$$

Figure 4: Encoding of linear types and contexts

recursive types of the form $U_1 = \mathsf{un}\,?L.U_2$ and $U_1 = \mathsf{un}\,!L.U_2$ with $U_1 \neq U_2$, and processes of the form $\overline{x}\langle x\rangle.P$.

We have the following theorem, which says that balanced typings are preserved during the computation.

Theorem 5.1 (Subject reduction [13])**.** *Let Δ be balanced and assume $\Delta \vdash K$. If $K \Rightarrow K'$ then $\Delta' \vdash K'$ with Δ' balanced.*

Encoding. We encode the source language in the typed π-calculus of Section 2 by assuming the existence of two distinct *reserved* variables *true* and *false* that can only occur free in the syntax of processes P. Figure 4 introduces the encoding $(\!(\cdot)\!)$ from qualifier-preserving contexts Δ to contexts Γ, where $\Delta, x\colon (N_1, N_2)$ does preserve qualifiers whenever N_1 and N_2 are both linear or both unrestricted, and when Δ does. Contexts that do not preserve qualifiers are unbalanced and are not interesting since they do not guarantee subject reduction. Type bool is mapped into $\bot$, which we remind is a label for the termination $(!(\mathsf{end},\mathsf{end}), ?(\mathsf{end},\mathsf{end}))$. The encoding relies on the auxiliary encoding $[\![\,\cdot\,]\!]$ from linear end point types N to end point types S. The auxiliary encoding disregards unrestricted types occurring at the end of a linear end point L by setting the continuation to end. This is sound since we know that in the source language the channels described by these types are not used in i/o. We consider *indexed* processes K_i and P_i such that each restricted variable has an unique natural index, and the remaining variables have index 0; we let $\mathsf{idx}(K_i)$ be the set of the indexes greater than 0. We will use positive indexes to retrieve the type of restricted variables, following De Bruijn. We define the encoding $[\![\cdot]\!]_\phi$ from processes K_i to processes P_i with the following clauses and let the remaining

ones be homomorphic, where ϕ is a type decoration function from indexes i to types T, w range over x_i and true , false and $[\![\cdot]\!]$ does project the boolean constants true and false into the reserved keywords *true* and *false*: $[\![\mathsf{true}]\!] = true$, $[\![\mathsf{false}]\!] = false$, and $[\![x_i]\!] = x_i$.

$$
\begin{array}{rcll}
[\![\overline{x_i}\langle w\rangle.K_i]\!]_\phi & \stackrel{\text{def}}{=} & \overline{x_i}\langle [\![w]\!]\rangle.[\![K_i]\!]_\phi & \\
[\![x_i(y_0).K_j]\!]_\phi & \stackrel{\text{def}}{=} & x_i(y_0).[\![K_i]\!]_\phi & \\
[\![(\nu x_i)K_j]\!]_\phi & \stackrel{\text{def}}{=} & (\nu x_i\colon (\!(L)\!))[\![K_j]\!]_\phi & \phi(i) = L \\
[\![\mathsf{if}\, w\, \mathsf{then}\, K_i\, \mathsf{else}\, K_j]\!]_\phi & \stackrel{\text{def}}{=} & \mathsf{if}\, [\![w]\!] = true\, \mathsf{then}\, [\![K_i]\!]_\phi\, \mathsf{else}\, [\![K_j]\!]_\phi &
\end{array}
$$

Example 5.2. *We map a typed derivation $\Delta \vdash K_i$ into a derivation $\Gamma \vdash P$ of Figure 2. We represent unrestricted recursive types of the form $\mu a.\mathsf{un}?L.U$ such that $U = \mu a.\mathsf{un}?L.U$ with $*?L$, and types of the form $\mu a.\mathsf{un}!L.U$ such that $U = \mu a.\mathsf{un}!L.U$ with $*!L$; we avoid to write index zero. Let $N \stackrel{def}{=} \mathsf{lin}!\mathsf{bool}.\mathsf{end}$, $\Delta \stackrel{def}{=} z\colon (*?N, *!N), u\colon (*?\mathsf{bool}, *!\mathsf{bool})$, and $K_1 \stackrel{def}{=} z(j).\overline{j}\langle\mathsf{true}\rangle \mid (\nu y_1)(\overline{z}\langle y_1\rangle.y_1(w).\mathsf{if}\, w\, \mathsf{then}\, \overline{u}\langle w\rangle$. We note that there is a derivation tree $\Delta \vdash K_1$ such that y_1 has assigned $(N, \overline{N})$. We thus let $\phi \stackrel{def}{=} 1 \mapsto (N, \overline{N})$, and have $(\!((N, \overline{N})\!)) = (!\bot.\mathsf{end}, ?\bot.\mathsf{end})$, $(\!(\Delta)\!) = true\colon \bot, false\colon \bot, z\colon (?(!\bot.\mathsf{end}, \mathsf{end}), !(!\bot.\mathsf{end}, \mathsf{end})), u\colon (?\bot, !\bot)$, and $[\![K_1]\!]_\phi = z(j).\overline{j}\langle true\rangle \mid (\nu y_1\colon (!\bot.\mathsf{end}, ?\bot.\mathsf{end}))\ \overline{z}\langle y_1\rangle.y_1(w).\mathsf{if}\, w = true\, \mathsf{then}\, \overline{u}\langle w\rangle$. We conclude that $(\!(\Delta)\!) \vdash [\![K_1]\!]_\phi$. We also note that $K_1 \Rightarrow \overline{u}\langle\mathsf{true}\rangle$, and $[\![K_1]\!]_\phi \Rightarrow \overline{u}\langle true\rangle$.*

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

Theorem 5.3 (Linear-π to π correspondence)**.** *Let $\Delta \vdash K_i$. Then there is a decoration function ϕ such that $\mathrm{dom}(\phi) = \mathsf{idx}(K_i)$ and*

1. $(\!(\Delta)\!) \vdash [\![K_i]\!]_\phi$
2. $K_i \to K_i'$ *implies* $[\![K_i]\!]_\phi \to [\![K_i']\!]_{\phi'}$ *for some* ϕ' *s.t.* $\mathrm{dom}(\phi') = \mathrm{dom}(\phi)$.

Proof. Given the derivation tree $\Delta \vdash K_i$, when $\mathrm{dom}(\phi) \neq \emptyset$ we build the decoration function ϕ by recording all typings for the indexed restricted variables in a list of pairs of the form $[(1, L_1) :: \cdots :: (n, L_n)]$, where $n \geq 1$. The result (1) then follows by proceeding by induction on the length of the inference $\Delta \vdash K_i$. We illustrate the most interesting cases, where we avoid indexes when unnecessary. In case [TL-RES] we have $\Delta \vdash (\nu x_i)K_i$ inferred from $\Delta, x_i\colon (N, \overline{N}) \vdash K_i$, and we know that $\phi(i) = (N, \overline{N})$. We apply [T-RES] to the I.H and infer the desired result, $(\!(\Delta)\!) \vdash (\nu x_i\colon (\!(\!(N, \overline{N})\!)\!))K_i$. As further example, consider case [TL-INC] and let $\Delta, x\colon (q?L.N, N') \vdash x(y).K_i$ be inferred from $\Delta, x\colon (N, N'), y\colon L \vdash K_i$, with the conditions on the qualifiers. Let $q = \mathsf{lin}$. Since $\Delta, x\colon (q?L.N, N')$ does preserve qualifiers, we have that $N' \neq U'$, for some U'. We use condition $(*)$ and infer that $N \neq U$ as well. We can apply the I.H. and infer that $(\!(\Delta, x\colon (N, N'), y\colon L)\!) \vdash [\![K_i]\!]_\phi$. A simple case-analysis on the encoding let us infer the desired result by applying

[T-InS-L]: $(\!(\Delta, x\colon (\mathsf{lin}?L.N, N')\,)\!) \vdash x(y).[\![K_i]\!]_\phi$. Let now $q = \mathsf{un}$. The hypotheses on qualifiers let us infer that $N' = U'$ and $N = \mathsf{un}?L.N$. We apply the I.H. and infer $(\!(\Delta, x\colon (N, N'), y\colon L)\,)\!) \vdash [\![K_i]\!]_\phi$. An application of [T-In-L] gives us the desired result, $(\!(\Delta, x\colon (\mathsf{un}?L.N, N')\,)\!) \vdash x(y).[\![K_i]\!]_\phi$. As last example, take [Tl-If]: $\Delta_1 \circ \Delta_2 \vdash \mathsf{if}\, w\, \mathsf{then}\, K_i\, \mathsf{else}\, K_j$ inferred from $\Delta_1 \vdash w\colon \mathsf{bool}$, $\Delta_2 \vdash K_i$, and $\Delta_2 \vdash K_j$. First, note that Δ_1 and Δ_2 may have different domain, because of linear types contained in Δ_2 and not in Δ_1. Consider now w, which can be a boolean constant or a variable x such that $\Delta_1(x) = \mathsf{bool}$ and $\Gamma_2(x) = \mathsf{bool}$, because of context split of booleans; the case $w = x_i$ with $i > 0$ is ruled out since booleans do not appear under restriction. In both cases we infer $(\!(\Delta_1)\!) \vdash [\![w]\!]\colon \bot$. Moreover, we can weaken the judgement by adding typings $u\colon \top, \ldots, z\colon \top$, so to make the domain of $(\!(\Delta_1)\!)$ equal to that of $(\!(\Delta_2)\!)$: $(\!(\Delta_1)\!), u\colon \top, \ldots, z\colon \top \vdash [\![w]\!]\colon \bot$. By I.H. we have $(\!(\Delta_2)\!) \vdash [\![K_i]\!]_\phi$, and $(\!(\Delta_2)\!) \vdash [\![K_j]\!]_\phi$. Note that (i) $(\!(\!(\Delta_1)\!), u\colon \top, \ldots, z\colon \top) \circ (\!(\!(\Delta_2)\!)) = (\!(\Delta_1 \circ \Delta_2)\!)$, and (ii) $(\!(\Delta_1 \circ \Delta_2)\!) = (\!(\Delta_2)\!)$, and (iii) $(\!(\Delta_2)\!)(true) = \bot$, as in all environments generated by the encoding. We apply [T-If] and infer the desired result, $(\!(\Delta_1 \circ \Delta_2)\!) \vdash [\![\mathsf{if}\, w\, \mathsf{then}\, K_i\, \mathsf{else}\, K_j]\!]_\phi$.

To prove (2), we label linear π-calculus reductions occurring on restricted channels with positive indexes (rather than with τ), and the remaining ones as in Figure 1. We then show a stronger result: (a) $K_i \xrightarrow{\mu} K_i'$ implies $[\![K_i]\!]_\phi \xrightarrow{\mu} [\![K_i']\!]_\phi$, and (b) $K_i \xrightarrow{h} K_i'$ implies $[\![K_i]\!]_\phi \xrightarrow{\tau} [\![K_j']\!]_{\phi \mapsto h}$, where when $h > 0$ and $\phi(h) = L$ we set $(\phi \mapsto h)(h) = \mathsf{next}(L)$ (cf. Section 3). The proof is by induction on the length of the inference. Most cases are straightforward, while the matching cases follow from the axioms $\mathsf{if}\, true = true\, \mathsf{then}\, P\, \mathsf{else}\, Q \xrightarrow{\tau} P$ and $\mathsf{if}\, false = true\, \mathsf{then}\, P\, \mathsf{else}\, Q \xrightarrow{\tau} Q$. The interesting case is restriction: $(\nu x_i)K_j \xrightarrow{i} (\nu x_i)K_j'$ inferred from $K_j \xrightarrow{x_i} K_j'$. We apply [R-ResB] to the I.H. $[\![K_j]\!]_\phi \xrightarrow{x_i} [\![K_j']\!]_\phi$, and infer $(\nu x_i\colon (\!(L)\!))[\![K_j]\!]_\phi \xrightarrow{\mu} (\nu x_i\colon \mathsf{next}((\!(L)\!)))[\![K_j']\!]_\phi$, where we let $\phi(i) = L$. This is the expected result, as $(\phi \mapsto h)(i) = \mathsf{next}(L)$, and $(\!(\mathsf{next}(L)\,)\!) = \mathsf{next}((\!(L)\!))$. □

Cross-encodings. The identified fragment is interesting because it permits to encode several variants of typed π-calculi. The paper [13] introduces an encoding of the branch-select free fragment of the π-calculus with polarities presented in [7], and an encoding of the linear π-calculus introduced in [19]. Both encodings project session and linear types into types L, and map polarized and standard π-calculus processes into processes K. The paper establishes a typing and reduction correspondence for both systems. This shows that [13] subsumes those systems; more precisely, the fragment presented in this section does suffice to encode both the recursive-free fragment of [7] and [19]. Since we encode types L and processes K in our framework by having a correspondence result (Theorem 5.3), this shows that we subsume the core of the polarised π-calculus in [7], and the linear π-calculus in [19] as well.

6. Conclusions

We introduce an affine discipline for session types in a π-calculus with name matching, and implement the typing rules in in a type checking algorithm based on functional patterns. The algorithm is sound, that is processes accepted by the algorithm are well-typed, although not complete: we identify the class of typed processes rejected by the algorithm as *Wait for* deadlocks [2].

Rejecting such processes makes sense since they are potentially dangerous, e.g. they could represent a financial transaction that gets stuck in a critical part, but is a draconian choice. A more flexible approach would permit to find a fix for the typed process and to propose to the programmer two choices: keep the original process or run a new one obtained by deadlock resolution.

In ongoing work [12], we are adding this feature to the algorithm by devising a process transformation of *Wait for* deadlocks that is guided by session types. The main idea is to detect such deadlocks during type-checking by exploiting the concept behind the `dead` predicate, and to install a forwarder [17] in the deadlocked point in order to disentangle the blocked channel, while preserving type-safety and a correspondence among the original and the resolved process.

To the best of our knowledge, the idea of typing if-then-else processes that do not exhibit the same behaviour in the branches by imposing an affine discipline is a new contribution in the session type literature. Since behavioural type disciplines tend to impose uniform typings for both branches of an if-then-else, we believe that our proposal is a step towards more flexible type systems that can be of interest for the programming languages community.

A limitation of our work is that session types and channel types do not mix. While most session systems in the literature follow the same approach (cf. [16, 7]), recently there have been proposals to let linear identifiers to progress to unrestricted identifiers [26, 13]. We note that the combination of this feature with the ability to accept the if-then-else processes of our interest is challenging, and eventually leads to a much more complex definition of the algorithm. While we think that this would be feasible, we preferred to keep the presentation compact. We also believe that our approach leads to manageable algorithms for type inference, for instance by following the constraint-based techniques in [20]. We leave this as future work. As a further improvement, we plan to introduce recursive types in the typing theory, and extend the algorithm accordingly, following the ideas in [13]. Last, we are exploring a semantic characterization of deadlocks based on typed behavioral equivalences [15]. We believe that this would complete the results in Section 4.1.

Acknowledgements. Work partially supported by the project PTDC/EIA-CCO/117513/2010 Liveness, Statically. I would like to thank Antonio Ravara, which gave me great advices on the structure the paper, and the anonymous reviewers for their constructive criticism.

References

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

[2] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[3] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In *WS-FM*, volume 6194 of *LNCS*, pages 1–28. Springer, 2009.

[4] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.

[5] Manuel Fähndrich et al. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190. ACM, 2006.

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

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

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

[9] Marco Giunti. A type checking algorithm for qualified session types. In *WWV*, volume 61 of *EPTCS*, pages 96–114, 2011.

[10] Marco Giunti, Kohei Honda, Vasco T. Vasconcelos, and Nobuko Yoshida. Session-based type discipline for pi calculus with matching. In *PLACES*, 2009.

[11] Marco Giunti, Catuscia Palamidessi, and Frank D. Valencia. Hide and new in the pi-calculus. In *EXPRESS/SOS*, volume 89 of *EPTCS*, pages 65–79, 2012.

[12] Marco Giunti and Antonio Ravara. Towards static deadlock resolution in the pi calculus. In *TGC*, 2013. To appear.

[13] Marco Giunti and Vasco T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR*, volume 6269 of *LNCS*, pages 432–446. Springer, 2010.

[14] Marco Giunti and Vasco T. Vasconcelos. Linearity, session types and the pi-calculus. *Mathematical Structures in Computer Science*, 2013. In press.

[15] Matthew Hennessy. *A Distributed Pi-calculus*. Cambridge University Press, 2007.

[16] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[17] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[18] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in haskell. In *PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010.

[19] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[20] Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Typing component-based communication systems. In *FMOODS/FORTE*, volume 5522 of *LNCS*, pages 167–181. Springer, 2009.

[21] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[22] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[23] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *1st ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2008.

[24] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[25] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.

[26] Vasco T. Vasconcelos. Sessions, from types to programming languages. In *The Concurrency Column*, pages 53–73. *Bulletin of the EATCS* 103, 2011.

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