Information and Computation 217 (2012) 52–70

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Fundamentals of session types

Vasco T. Vasconcelos

LaSIGE, Faculty of Sciences, University of Lisbon, Portugal

ARTICLE INFO

Article history:
Received 10 February 2012
Revised 7 May 2012
Available online 23 May 2012

ABSTRACT

We present a reconstruction of session types in a linear pi calculus where types are qualified as linear or unrestricted. Linearly qualified communication channels are guaranteed to occur in exactly one thread, possibly multiple times; unrestricted (or shared) channels may appear in an unbounded number of threads. In our language each channel is characterized by two distinct variables, one used for reading, the other for writing; scope restriction binds together two variables, thus establishing the correspondence between the two ends of the same channel. This mechanism allows a precise control of resources via a conventional linear type system. Furthermore, the uniform treatment of linear and shared channels leads to a surprisingly simply theory which, in addition, extends typability when compared to traditional systems for session types. We build the language gradually, starting from simple input/output, then adding recursive types, replication and finally choice. We also present an algorithmic type checking system.

1. Introduction

In complex concurrent interactions partners often exchange a large number of messages as part of a pre-established scheme. The nature and order of these messages constitute a natural candidate for structuring the interactions themselves. It is in this context that session types make their contribution by allowing a concise description of the continuous interactions among partners in a concurrent computation.

Central to the theory of session types is the distinction between linear and shared (or unrestricted) communication channels: linear channels are supposed to be known to two interacting parties alone, shared channels can be shared by zero or more partners. Session types were first introduced in a variant of the pi calculus [9], featuring bound output and a syntactic distinction between linear and shared channels. Later, together with a new notion of subtyping, the theory was adapted to a conventional pi calculus with free output [4]. Yet, all the hitherto formulations of the calculus syntactically distinguish two classes of channels—linear and shared—and the type theory stratifies types in two distinct categories—linear and shared—leading to the duplication of syntactic concepts, reduction rules and typing rules.

This paper introduces a reconstruction of session types based on the ideas of a linear type system for the lambda calculus [22]. Rather than using two distinct syntactic categories for linear/shared types, we qualify pre-types with a lin/un annotation. This simple move allows in turn to abolish the syntactic distinction between linear and shared channels. Instead we work with undifferentiated channels, leaving the linear/shared characterization for the type system. The benefit is an extremely simple theory, with no concept/rule duplication, that, somewhat unexpectedly, extends typability by allowing channels governed by a linear type to become shared while still allowing interaction.

The previous version of this paper appeared as lecture notes for a summer school [19]; we have kept the gradual introduction of the various concepts usually associated to the pi calculus and to session types, thus motivating the dependencies between the various concepts involved. We start by studying a language with input, output, parallel composition, and scope

E-mail address: vv@di.fc.ul.pt.

0890-5401/$ – see front matter
http://dx.doi.org/10.1016/j.ic.2012.05.002

$P ::=$		Processes:
	$\bar{x}\,v.P$	output
	$x(x).P$	input
	$P \mid P$	parallel composition
	if v then P else P	conditional
	$\mathbf{0}$	inaction
	$(\nu xx)P$	scope restriction
$v ::=$		Values:
	x	variable
	true | false	boolean values

Fig. 1. Syntax of processes.

restriction (Sections 2 and 3). Even though the required syntactic and operational semantics machinery are in place, the particular form of types does not allow to type useful unrestricted channels—recursive types provide for such a facility (Section 4). Up to this point the language does not allow to describe unbounded computation—we introduce replication for the effect (Section 5). We then incorporate choice in the form of branching and selection (Section 6), and prove the soundness of the type system with respect to the operational semantics (Section 7). The last step in the development of our language introduces an algorithmic type checking system and proves its correctness with respect to the type system introduced in Sections 3 to 6 (Section 8). The closing section discusses related work and concludes the paper.

2. The pi calculus

Fig. 1 presents the syntax of our language. There is one base set only: variables. When writing processes, any lower case roman-letter except u and v represents a variable. Depending on the context we also use the expression "channel end" to mean a variable.

In interactive behavior, variables come in pairs, called *co-variables*. The best way to understand co-variables is to think of them as representing the two ends of a communication channel—some parties write on the first end, others read from the second. In order to communicate, threads do not need to share variables; since a channel is represented as a pair of co-variables, each thread may hold one variable allowing it to read or to write on the channel. This mechanism allows a precise control of resources via a rather conventional linear type system.

The constructors of the language are those of the pi calculus with boolean values, except for a small difference in scope restriction. The output process $\bar{x}\,v.P$ writes value v on channel x and continues as P. Conversely, the input process $y(z).P$ reads from channel y a value it uses to replace the bound variable z before continuing with the execution of process P. The parallel composition $P \mid Q$ allows processes P and Q to proceed concurrently. The conditional process if v then P else Q executes P or Q depending on the boolean value v. The terminated process, or inaction, is denoted by $\mathbf{0}$. The particular form of scope restriction $(\nu xy)P$ is the novelty with respect to the pi calculus—not only it simultaneously hides (or binds) two variables, but it also establishes x and y as two co-variables, allowing communication to happen in process P, between a thread writing on x and another thread reading from y (or vice versa). It should be stressed that $(\nu xy)P$ is not a short form for $(\nu x)(\nu y)P$; instead it binds two co-variables together.

In our language parenthesis represent bindings—variable y occurs *bound* in $x(y).P$ and in $(\nu xy)P$; variable x occurs bound in $(\nu xy)P$. A variable that occurs in a non-bound position within a process is said to be *free*. The set of free variables in a process P, denoted by $\mathrm{fv}(P)$, is defined accordingly, and so is alpha-conversion, as well as the capture-free substitution of variable x by value v in process P, denoted by $P[v/x]$. Notice that substitution is not a total function; it is not defined, e.g., for $(\overline{y}\,\mathsf{false})[\mathsf{true}/y]$. When writing $P[v/x]$ we assume that the substitution operation involved is defined. We work up to alpha-conversion and follow Barendregt's variable convention, whereby all variables in binding occurrences in any mathematical context are pairwise distinct and distinct from the free variables.

To evaluate processes we use a small step operational semantics. As usual in the pi calculus, we factor out a structural congruence relation on processes allowing the syntactic rearrangement of these, thus contributing for a more concise presentation of the reduction relation.

Structural congruence is the smallest congruence relation on processes that satisfies the axioms in Fig. 2. The axioms are standard in pi calculus. The first three say that parallel composition is commutative, associative and has the terminated process $\mathbf{0}$ for neutral. The first axiom on the second line is called scope extrusion, and allows the scope of a ν-binder to extend to a new process Q or to retract from this, as needed. Notice that the proviso "x, y not free in Q" is redundant in face of the variable convention, for x occurring bound in $(\nu xy)P$ cannot occur free in Q. The last two axioms allow to collect unused restrictions and to exchange the order of bindings.

The *operational semantics* is also defined in Fig. 2, as a binary relation on processes. In rule [R-Com], a process willing to send a value v on variable x, in parallel with another process ready to receive on variable y, engages in communication only if x, y are two co-variables, that is if the two processes are underneath a restriction (νxy). In that case, both prefixes are consumed and v replaces the bound variable z in the receiving party. The binding (νxy) persists, in order to potentiate

Structural congruence, $P \equiv P$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P$$
$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q) \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu wx)(\nu yz)P \equiv (\nu yz)(\nu wx)P$$

Reduction rules, $P \to P$

$$(\nu xy)(\bar{x}\, v.P \mid y(z).Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{[R-Com]}$$

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

$$\frac{P \to Q}{(\nu xy)P \to (\nu xy)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \text{[R-Res] [R-Par]}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \qquad \text{[R-Struct]}$$

Fig. 2. Operational semantics.

$q ::=$		Qualifiers:
	lin	linear
	un	unrestricted
$p ::=$		Pretypes:
	$?T.T$	receive
	$!T.T$	send
$T ::=$		Types:
	bool	boolean
	end	termination
	$q\ p$	qualified pretype
$\Gamma ::=$		Contexts:
	$\emptyset$	empty context
	$\Gamma, x : T$	assumption

Fig. 3. The syntax of types.

further interactions in the resulting process. Process R collects all other threads that may share x and y. A direct consequence of this rule is that communication cannot happen on free variables for there is no way to tell what the co-variables are.

Rules [R-IfT] and [R-IfF] replace a conditional process with the then branch or with the else branch, depending on the value of the condition. Rules [R-Res] and [R-Par] allow reduction to happen underneath scope restriction and parallel composition, respectively. Finally, rule [R-Struct] incorporates structural congruence in the reduction relation.

To lighten the syntax in examples, we omit the trailing "**.0**" in processes. As an example we have:

$$(\nu x_1x_2)\big(\overline{x_1}\,\mathsf{true}.x_1(y).\bar{a}\,y \mid x_2(z).\overline{x_2}\,z\big) \to (\nu x_1x_2)\big(x_1(y).\bar{a}\,y \mid \overline{x_2}\,\mathsf{true}\big) \to \bar{a}\,\mathsf{true}$$

If the process above is well behaved according to our semantics, we would not like to consider as well formed the processes below.

$$(\nu x_1x_2)\big(\overline{x_1}\,\mathsf{true} \mid x_2(y).\overline{y}\,\mathsf{false}\big) \qquad \times$$
$$(\nu x_1x_2)\mathsf{if}\ x_1\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0} \qquad \times$$
$$\bar{x}\,\mathsf{true} \mid x(y) \qquad \times$$

In the first, substitution is not defined, as discussed above, in the second the conditional process tests a channel end rather than a boolean value, and in the last the two threads are both trying to write and to read on the same channel end. The type system introduced in the next section rules out such processes.

3. Typing

The syntax of types is described in Fig. 3. We have bool, the type of the boolean values, end, used to type a channel end on which no further interaction is possible, and lin/un annotated pretypes. Pretypes $!T.U$ and $?T.U$ describe channel ends ready to send or to receive a value of type T and then continuing their interaction as prescribed by type U.

Intuitively, linearly qualified types describe channel ends that occur in exactly *one thread*, a thread being any process not comprising parallel composition. The unrestricted qualifier indicates that the value can occur in multiple threads. In this way, a type $\mathsf{lin}\,!T.U$ represents a channel that can be used once for sending a value of type T, and $\mathsf{un}\,!T.U$ a channel that

$$\overline{q\,?T.U} = q\,!T.\overline{U} \qquad \overline{q\,!T.U} = q\,?T.\overline{U} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

Fig. 4. The dual function on types.

can be used from multiple threads to send values of type T. In either case, type U describes the subsequent behavior of the channel. Typing contexts, also introduced in Fig. 3, gather type information on variables. We require the various variables that appear in a context to be pairwise distinct, and treat contexts up to the exchange of entries, not distinguishing, e.g., $x : \mathsf{bool}, y : \mathsf{end}$ from $y : \mathsf{end}, x : \mathsf{bool}$.

To lighten the syntax in examples involving types, we adopt a few extra abbreviations: we omit all linear type qualifiers and only annotate unrestricted types, and we omit the trailing ".end" in types. Further, in examples involving communication we assume that co-variables are annotated with subscripts 1 and 2, for example (x_1, x_2) and (y_1, y_2), even if not explicitly under a ν binding. We also use x (subscripted or not) for a variable of an arbitrarily qualified type, a for a variable of an unrestricted type and c for a variable of a linear type. Under these assumptions, the first process is well formed, whereas the last one is not.

$$\overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{false} \qquad \checkmark$$

$$\overline{c}\,\mathsf{true} \mid \overline{c}\,\mathsf{false} \qquad \times$$

Type duality plays a central role in the theory, ensuring that communication on co-variables proceeds smoothly. Intuitively, the dual of output is input and the dual of input is output. In particular if U is dual of T, then $q?S.U$ is dual of $q!S.T$. Type end is dual of itself; duality is not defined for the bool type. The definition is in Fig. 4.

Based on duality, we would like to accept the first two processes, but not the last two.

$$\overline{x_1}\,\mathsf{true} \mid x_2(z) \qquad \checkmark$$

$$\overline{c_1}\,\mathsf{true}.c_1(w) \mid c_2(z).\overline{c_2}\,\mathsf{false} \qquad \checkmark$$

$$\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{false} \qquad \times$$

$$\overline{c_1}\,\mathsf{true}.c_1(w) \mid c_2(z).c_2(t) \qquad \times$$

One might expect duality to affect the parameter of the sent and the received type, e.g., $\overline{q\,?T.U} = q\,!\overline{T}.\overline{U}$. That would be unsound as the example below shows. Suppose that we would like to type process

$$\overline{x_1}\,y_2 \mid x_2(z).\bar{z}\,\mathsf{true} \mid \overline{y_1}\,\mathsf{false} \qquad \times$$

at context $x_1 : !(!\mathsf{bool}), x_2 : ?(?\mathsf{bool}), y_1 : !\mathsf{bool}, y_2 : !\mathsf{bool}$, where the type of argument y_2 in the send operation on x_1 is dual to that of the parameter z in the receive operation on x_2, that is, !bool is dual to ?bool. It should be easy to see that the process reduces to an illegal process, where y_1 and y_2 cannot interact.

$$\overline{y_2}\,\mathsf{true} \mid \overline{y_1}\,\mathsf{false} \qquad \times$$

The dual function is not total: it is not defined on bool, nor on any type "terminating" in bool, such as ?bool.bool. Had we incorporated other base types in our language (integers for example), duality would not be defined on them as well. Duality is a function defined on session types only: input, output, and the terminated session end. Imagine that we set $\overline{\mathsf{bool}} = \mathsf{bool}$; we would be able to type the process

$$(\nu xy)\mathsf{if}\ x\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0} \qquad \times$$

or any process reducing to it.

Our type system maintains the following invariants.

- References to linear channel ends occur in exactly one thread;
- Co-variables have dual types.

The first invariant is maintained via a context split operation which relies on a $\mathsf{un}(T)$ predicate, both introduced below. The second invariant is managed by the typing rule for scope restriction also described below.

For each qualifier q we define a predicate $q(T)$ and its extension to contexts $q(\Gamma)$ as follows.

- $\mathsf{un}(T)$ if and only if $T = \mathsf{bool}$ or $T = \mathsf{end}$ or $T = \mathsf{un}\,p$.
- $\mathsf{lin}(T)$ if and only if true.
- $q(\Gamma)$ if and only if $(x : T) \in \Gamma$ implies $q(T)$.

We maintain the linearity invariant through the standard linear context split operation. When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in

Context split, $\Gamma = \Gamma \circ \Gamma$

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad \mathsf{un}(T)}{\Gamma, x:T = (\Gamma_1, x:T) \circ (\Gamma_2, x:T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x:\mathsf{lin}\, p = (\Gamma_1, x:\mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x:\mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x:\mathsf{lin}\, p)}$$

Context update, $\Gamma + x:T = \Gamma$

$$\frac{x:U \notin \Gamma}{\Gamma + x:T = \Gamma, x:T} \qquad \frac{\mathsf{un}(T)}{(\Gamma, x:T) + x:T = (\Gamma, x:T)}$$

Fig. 5. Context split and context update.

Typing rules for values, $\Gamma \vdash v : T$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \quad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \quad \frac{\mathsf{un}(\Gamma)}{\Gamma, x:T \vdash x : T} \qquad \text{[T-True] [T-False] [T-Var]}$$

Typing rules for processes, $\Gamma \vdash P$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \quad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{[T-Inact] [T-Par]}$$

$$\frac{\Gamma, x:T, y:\overline{T} \vdash P}{\Gamma \vdash (\nu xy)P} \quad \frac{\Gamma_1 \vdash v : \mathsf{bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q} \qquad \text{[T-Res] [T-If]}$$

$$\frac{\Gamma_1 \vdash x : q?T.U \quad (\Gamma_2 + x:U), y:T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P} \qquad \text{[T-In]}$$

$$\frac{\Gamma_1 \vdash x : q!T.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 + x:U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \bar{x}\, v.P} \qquad \text{[T-Out]}$$

Fig. 6. Typing rules.

two and passing a different part to each process. In this way, if x is a linear variable then the process $\bar{x}\,\mathsf{true} \mid \bar{x}\,\mathsf{true}$ is not typable, since x can only occur in one of the parts, allowing to type one but not both processes. Fig. 5 defines the context split relation $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that in the third rule, x is not in Γ_2 since it is not in Γ, hence not in $\Gamma_1 \circ \Gamma_2$, and similarly for the last rule and Γ_1. We often write $\Gamma_1 \circ \Gamma_2$ to denote a context Γ such that $(\Gamma, \Gamma_1, \Gamma_2)$ is in the context split relation.

We also need an operation to update a context with the new type for a variable used for input or output. The rules are in Fig. 5 and require linear variables not to be in the context, and unrestricted variables to have their types unchanged.

Equipped with the notions of context split, context update and type duality we are ready to introduce the typing rules. We distinguish the typing rules for values with judgments of the form $\Gamma \vdash v : T$, from those for processes with judgments $\Gamma \vdash P$. The rules are in Fig. 6.

We want to make sure that linear variables are not discarded without being used; the base cases of the type system check that there is no linear variable in the context. In particular, in rules [T-Var], [T-False] and [T-True] for values and [T-Inact] for processes, we check that Γ is unrestricted. Notice that this does not preclude type T itself from being linear in rule [T-Var]. The typing rules for values are conventional—boolean values have type bool, variables have the type prescribed by the context.

Rule [T-Par] uses context splitting to partition linear variables between the two processes: the incoming context is split into Γ_1 and Γ_2, and we use the former to type check process P and the latter to type check process Q, so that each process will have access to all unrestricted channels but only to a disjoint part of the linear ones. For rule [T-Res] we add to the context two extra hypotheses for the newly introduced variables, one at some type T, the other at a dual type $\overline{T}$. The rule captures the essence of co-variables: they must have dual types.

Similarly to parallel composition, rule [T-If] for the conditional process splits the incoming context in two parts, one to type the value to be tested, the other to type the two branches, P and Q. Context Γ_1 is used to type the condition; context Γ_2 is used for the two branches since only one of P or Q will be executed (contrary to parallel composition). Given that type bool is unrestricted, Γ_1 must be unrestricted as well (this follows from a simple analysis of the various typing rules for values). Then Γ_1 contains all unrestricted types in the incoming context and Γ_2 is the incoming context itself (this and other properties of context split are the object of Lemma 7.1), meaning that each branch has access to all variables in the incoming context.

Similarly to the rule for the conditional process, rule [T-In] splits the context into two parts: one to type check variable x, the other to type check continuation P. If x is of type $q?T.U$, we know that the bound variable y is of type T, and we type check P under the extra assumption $y : T$. Equally important is the fact that the continuation uses variable x at continuation type U, that is, process $x(y).P$ uses variable x at type $q?T.U$ whereas P may use the *same* variable this time at type U. If x is a linear variable then it is certainly not in Γ_2 because it is in Γ_1. If, on the other hand, x is unrestricted then context update is only defined when U is equal to $q?T.U$, which will become possible with recursive types, introduced in Section 4.

New syntactic forms

$T ::=$	$\ldots$	Types:
	a	type variable
	$\mu a.T$	recursive type

New duality rules, $\overline{T} = T$

$$\overline{\mu a.T} = \mu a.\overline{T} \qquad \overline{a} = a$$

Fig. 7. Recursive types.

The rule for sending a value, [T-Out], splits the context in three parts, one to check x, another to check v and the last to check continuation P. Similarly to the rule for reception, the continuation process uses variable x at the continuation type, that is, $\overline{x}\,v.P$ uses x at type $q!T.U$, whereas P uses the same variable at type U.

There are many interesting pi calculus processes that our type system fails to check, including $\overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{true}$. In order to type this process we seek a context associating an unrestricted type to x, as in $x : \mathsf{un}!\mathsf{bool}.T$. Then the third premise of rule [T-Out] reads $(x : \mathsf{un}!\mathsf{bool}.T) + (x : T)$ which cannot be fulfilled by any type T built from the syntax in Fig. 3. Clearly, so far, we are dealing with a language of linear channels only.

Unlike other languages equipped with linear type systems, our type system offers no guarantee of progress. If fact processes can deadlock quite easily, it suffices to create two threads that read and write in the "wrong" order.

$$\overline{x_1}\,\mathsf{true}.\overline{y_1}\,\mathsf{false} \mid y_2(x).x_2(w) \qquad \checkmark$$

Even though one finds processes prefixed at any of the four linear variables, and the types are dual, the order by which the two threads order these prefixes is not conducting to reduction. An even more crafty process, uses channel passing to end up with a cycle including a single thread.

$$\overline{x_1}\,y_1 \mid x_2(z).\overline{z}\,\mathsf{true}.y_2(w) \qquad \checkmark$$

4. Recursive types

The typing rule for output processes (rule [T-Out] in Fig. 6) does not allow to type check a process $\overline{x}\,v.P$ with x unrestricted, for it requires the continuation U of type $\mathsf{un}!T.U$ to be equal to $\mathsf{un}!T.U$ itself. We would like to consider as a type the solution to the equation $U = \mathsf{un}!T.U$. Such a type may take the form of a regular infinite tree (a tree composed of finitely many distinct subtrees), for which a finite, μ, notation is introduced. Our type is then (the regular tree associated with) $\mu a.\mathsf{un}!T.a$.

Fig. 7 includes *recursive types* in the syntax of types, where we rely on one more base set, that of *type variables*. Recursive types are required to be *contractive*, i.e., containing no subexpression of the form $\mu a_1 \ldots \mu a_n.a_1$. The μ operator is a binder, giving rise, in the standard way, to notions of bound and free variables and alpha-equivalence. We denote by $T[U/a]$ the capture-avoiding substitution of a by U in T.

When moving to recursive types we use a notion of type equality based on regular infinite trees rather than the syntactic equality used in the previous section. To decide whether two types are equal we compare the infinite unfolding of the two types, a property known to be decidable. The formal definition, which we omit, is co-inductive. This allows us never to consider, in any mathematical context, a type $\mu a.T$ explicitly (or a for that matter). Instead, we pick another type in the same equivalence class, namely $T[\mu a.T/a]$. If the result of the process turns out to start with a μ, we repeat the procedure. Unfolding is bound to terminate due to contractiveness. For example, types $\mu a.!\mathsf{bool}.?\mathsf{bool}.a$ and $!\mathsf{bool}.\mu b.?\mathsf{bool}.!\mathsf{bool}.b$ are equivalent. In other words, we take an equi-recursive view of types [14].

The dual function, extended in Fig. 7 to the new type constructs, descends a μ-type and leaves type variables unchanged. To check that a given type T is dual of another type U, we first build the type $\overline{T}$ and then use the definition above to determine whether $\overline{T}$ is equal to U. For example, to show that $\mu a.?\mathsf{bool}.!\mathsf{bool}.a$ is dual of $!\mathsf{bool}.\mu b.?\mathsf{bool}.!\mathsf{bool}.b$, we build $\overline{\mu a.?\mathsf{bool}.!\mathsf{bool}.a} = \mu a.!\mathsf{bool}.?\mathsf{bool}.a$, and then show that $\mu a.!\mathsf{bool}.?\mathsf{bool}.a = !\mathsf{bool}.\mu b.?\mathsf{bool}.!\mathsf{bool}.b$.

The new type constructors are not qualified, instead $\mu a.T$ takes the qualifier of the enclosed type T. Contractivity ensures that types can be interpreted as regular infinite trees; it also ensures that we can always find out what the qualifier of a type is. Given that types $\mu a.T$ and $T[\mu a.T/T]$ can be used interchangeably, we do not have to touch the definitions of the lin and un predicates. For example, in order to determine whether type $\mu a.\mu b.!\mathsf{bool}.a$ is unrestricted, we take another type in the same equivalence class that does not start with a μ, for example, $!\mathsf{bool}.\mu a.!\mathsf{bool}.a$. Equipped with the equi-recursive notion of types, typing rules (in Fig. 6) remain unchanged.

Consider the type $\mathsf{un}?(!\mathsf{bool}).T$ of an unrestricted channel that receives a linear channel end capable of outputting a boolean value. The following sequent is easy to establish,

$$x_2 : \mathsf{un}?(!\mathsf{bool}).T \vdash x_2(z).\overline{z}\,\mathsf{true} \mid x_2(w).\overline{w}\,\mathsf{false} \qquad \checkmark$$

but only for an appropriate type T. Rule [T-In] dictates that it must be equivalent to $\mathsf{un}?(!\mathsf{bool}).T$, so that T can be, e.g., of the form $\mu a.\mathsf{un}?(!\mathsf{bool}).a$. This form of types is so common that we introduce a short form for them, simply writing $*?(!\mathsf{bool})$.

Our language does not include tuple passing as a primitive construct, rather it can only send or receive a single value at a time. Fortunately, tuple passing is easy to encode. To send a pair of values u, v of types T, U over a linear channel x, we just send the values, one at a time; no interference is possible due to the linear nature of the carrier channel.

$$\bar{x}\langle u, v\rangle.P \text{ abbreviates } \bar{x}\,u.\bar{x}\,v.P$$

However, if the tuple is to be passed on an unrestricted channel, then we must protect the two (separate) receive operations from unwanted interference, creating a new $?T.?U$ channel to carry the values. The standard encoding for the binary sending and receiving operations are as follow.

$$\overline{x_1}\langle u, v\rangle.P \text{ abbreviates } (\nu y_1 y_2)\overline{x_1}\,y_2.\overline{y_1}\,u.\overline{y_1}\,v.P$$

$$x_2(w, t).P \text{ abbreviates } x_2(z).z(w).z(t).P$$

The encodings are typable in our language, if we choose variable y_1 of appropriate linear type, $!T.!U$, and dually for y_2. Variable x_1 is then of type $*!(?T.?U)$, and dually for x_2. We abbreviate the type of channel that sends a pair of values of types T and U to $*!\langle T, U\rangle$, and dually for a channel that receives a pair of values, $*?\langle T, U\rangle$.

Here is another example on passing linear tuples on unrestricted channels. Suppose that we own a channel of type $!\mathsf{bool}.!\mathsf{bool}.?\mathsf{bool}$ and want to delegate the writing part (the initial $!\mathsf{bool}.!\mathsf{bool}$ part) to another process, but intend to locally perform the read operation (the final part $?\mathsf{bool}$). If we simply "pass" the channel, then we cannot further use it, unless we provide a means to get it back. Below is a process that writes two boolean values on a given channel z and then returns the channel (on a given channel w).

$$p_1(z, w).\bar{z}\,\mathsf{true}.\bar{z}\,\mathsf{true}.\overline{w}\,z \qquad \checkmark$$

A process that calls p_1 in order to write two boolean values on a given channel c, and then reads from the channel again, can be written as

$$\overline{p_2}\,\langle c, x_1\rangle.x_2(z).z(y) \qquad \checkmark$$

where p_1 is typed at $*?\langle!\mathsf{bool}.!\mathsf{bool}.?\mathsf{bool}, !(?\mathsf{bool})\rangle$.

New to this work, a channel can be created with a linear type and evolve, after some communication, into a channel with an unrestricted type and still be used for communication. For example, type $T = !\mathsf{bool}.*?\mathsf{bool}$ describes a channel that behaves linearly in the first interaction and unrestricted thereafter. Suppose that x_1 is of type T and x_2 of type $\overline{T}$.

$$\overline{x_1}\,\mathsf{true}.\big(x_1(y) \mid x_1(z)\big) \mid x_2(x).(\overline{x_2}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{false} \mid \overline{x_2}\,\mathsf{true}) \qquad \checkmark$$

$$\overline{x_1}\,\mathsf{true}.x_1(y).x_1(y) \mid x_2(z) \qquad \checkmark$$

$$\overline{x_1}\,\mathsf{true}.x_1(y) \mid x_2(y).\overline{x_2}\,\mathsf{true} \mid x_2(w).\overline{x_2}\,\mathsf{true} \qquad \times$$

So now we know that a traditional pi calculus channel that can be used an unbounded number of times for outputting boolean values is of type $*!\mathsf{bool}$, that is, $\mu a.!\mathsf{bool}.a$. Conversely, a channel that can be used for reading an unbounded number of boolean values is of type $*!\mathsf{bool}$, i.e., $\mu b.?\mathsf{bool}.b$. What about a channel that can be used both for reading and for writing? There is no such thing in this theory; the channel is represented by a pair of co-variables, one to read, the other to write. If a given process needs to gain access to the read and the write capability of a channel, then both ends must be passed, possibly using the encoding for pairs above.

$$a_1 : *!\langle!\mathsf{bool}, ?\mathsf{bool}\rangle, a_2 : *?\langle!\mathsf{bool}, ?\mathsf{bool}\rangle \vdash a_2(y_1, y_2).\big(\overline{y_1}\,\mathsf{false} \mid y_2(z)\big) \mid (\nu x_1 x_2)\,\overline{a_1}\,\langle x_1, x_2\rangle \qquad \checkmark$$

5. Replication

Up until now our language is strongly normalizing—each reduction step strictly decreases the number of symbols that compose the processes involved. To provide for unbounded behavior we introduce a special form of receptor that remains after reduction, called *replication*. The details are in Fig. 8.

Rather than introducing a new process constructor we annotate input processes with the lin/un qualifiers used in types. We then have $\mathsf{lin}\,x(y).P$ and $\mathsf{un}\,x(y).P$. The input process we have seen so far, $x(y).P$, is now taken as an abbreviation for $\mathsf{lin}\,x(y).P$ (we stick to our convention of omitting the lin qualifier). Processes of the form $\mathsf{un}\,x(y).P$ are shared (following the intuition of the un qualifier), and thus survive reduction so that they can be used by multiple clients. The reduction rule for linear input is that of Fig. 2; all we have done was to add the lin qualifier and to rename it to [R-LinCom], in order to stress the similarity with the unrestricted case. The rule for replicated processes, [R-UnCom], is similar to [R-LinCom] in all respects except that the replicated process $\mathsf{un}\,x(y).P$ persists in the resulting process.

New syntactic forms

$$\begin{array}{lll} P ::= & \ldots & \text{Processes:} \\ & q x(x).P & \text{input} \end{array}$$

New reduction rules, $P \to P$

$$(\nu xy)(\bar{x}\, v.P \mid \mathsf{lin}\, y(z).Q \mid R) \to (\nu xy)(P \mid Q\,[v/z] \mid R) \qquad \text{[R-LinCom]}$$

$$(\nu xy)(\bar{x}\, v.P \mid \mathsf{un}\, y(z).Q \mid R) \to (\nu xy)(P \mid Q\,[v/z] \mid \mathsf{un}\, y(z).Q \mid R) \qquad \text{[R-UnCom]}$$

New typing rules, $\Gamma \vdash P$

$$\frac{q_1(\Gamma_1 \circ \Gamma_2) \qquad \Gamma_1 \vdash x : q_2\, ?T.U \qquad (\Gamma_2 + x : U),\, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash q_1 x(y).P} \qquad \text{[T-In]}$$

Fig. 8. Replication.

With the introduction of replication we can mention a third invariant of our type system.

- Unrestricted input processes may not contain (free) linear variables.

To check the invariant we make use of the $q(\Gamma)$ predicate introduced in Section 3.

The previous typing rule for the input process, [T-In] in Fig. 6, is adapted to take into consideration the new lin/un qualifier. The new rule, [T-In] in Fig. 8, when applied to a linear process, becomes the rule with the same name in Fig. 6, since $\mathsf{lin}(\Gamma)$ is true for all typing contexts Γ (see Section 3). When in presence of a replicated process, the rule requires the process to be typable under an unrestricted context. Qualifiers q_1 and q_2 are not necessarily equal; in particular if q_1 is un then so is q_2, but $q_1 = \mathsf{lin}$ tells us nothing about q_2. To understand what would happen if we relax this restriction, consider the following process

$$\mathsf{un}\, x_2(z).\bar{c}\,\mathsf{true} \mid \overline{x_1}\,\mathsf{true} \mid \overline{x_1}\,\mathsf{false} \qquad \times$$

where we would like c to be typed at lin!bool. The process reduces in two steps to $\mathsf{un}\, x_2(z).\bar{c}\,\mathsf{true} \mid \bar{c}\,\mathsf{true} \mid \bar{c}\,\mathsf{true}$, invalid given the sought linearity for channel c. Instead, procedures that use linear values must receive them arguments, thus allowing the type system to check possible value duplications. If we pass channel c as parameter,

$$\mathsf{un}\, x_2(z).\bar{z}\,\mathsf{true} \qquad \checkmark$$

then the procedure can no longer be used by process $\overline{x_1}\, c \mid \overline{x_1}\, c$, because rule [T-Par] precludes splitting any context in two parts both containing a channel c of a linear type.

Replication, as firstly introduced in the pi calculus (by Milner [12]) takes a more general form, $!P$, standing for $P \mid P \mid \cdots$. The more general form of replication can be simulated by the following process,

$$(\nu x_1 x_2)\big(\overline{x_1}\, x_1 \mid \mathsf{un}\, x_2(y).(P \mid \overline{x_1}\, y)\big)$$

where x_1, x_2 and y do not occur free in P. An admissible rule for the new construct is quite intuitive.

$$\frac{\mathsf{un}(\Gamma) \qquad \Gamma \vdash P}{\Gamma \vdash\, !P}$$

Notice that the encoding uses no primitive value; the types for channel x_1, x_2 cannot however be written with our "$*$" abbreviation, instead we choose $\mu a.\mathsf{un}!a.a$ for x_1 and $\mu b.\mathsf{un}?b.b$ for x_2. We selected for our language a more controlled, lazily evaluated, form of replication, suitable to be incorporated in programming languages, as for example, in Pict [15].

6. Choice

Choice allows processes to offer a fixed range of alternatives and clients to select among the variety offered. We extend the syntax of our language with support for offering alternatives, called *branching*, and to choose among the alternatives, called *selection*. The details are in Fig. 9, where we add to our repertoire another base set—*labels*. Lower case letters l and m are used to denote labels.

A process of the form $x \lhd l.P$ selects one of the options offered by a process prefixed at the co-variable. Conversely, a process $x \rhd \{l_i : P_i\}_{i \in I}$ offers a range of options, each labelled with a different label in the set $\{l_i\}_{i \in I}$, for I some index set. Such a process handles a selection at label l_j by executing process P_j, if $j \in I$. The operational semantics is extended with rule [R-Case]. The rule follows the pattern of [R-Com] in Fig. 2 (or [R-LinCom] in Fig. 8): the two processes engaging in reduction must be underneath a prefix that puts the two co-variables in correspondence. The selecting party continues with process P, the branching party with the body of the selected choice, P_j.

Types for the new constructors are $\oplus\{l_i : T_i\}_{i \in I}$ and $\&\{l_i : T_i\}_{i \in I}$, representing channels ready to select or to offer l_i options. In either case type T_j describes the continuation once label l_j has been chosen. The new type structures are

New syntactic forms

$$\begin{array}{lll} P ::= & \ldots & \text{Processes:} \\ & x \lhd l.P & \text{selection} \\ & x \rhd \{l_i : P_i\}_{i\in I} & \text{branching} \\ p ::= & \ldots & \text{Pretypes:} \\ & \oplus\{l_i : T_i\}_{i\in I} & \text{select} \\ & \&\{l_i : T_i\}_{i\in I} & \text{branch} \end{array}$$

New reduction rules, $P \to P$

$$\frac{j \in I}{(\nu xy)(x \lhd l_j.P \mid y \rhd \{l_i : Q_i\}_{i\in I} \mid R) \to (\nu xy)(P \mid Q_j \mid R)} \quad \text{[R-Case]}$$

New duality rules, $\overline{T} = T$

$$\overline{q\oplus\{l_i : T_i\}_{i\in I}} = q\&\{l_i : \overline{T_i}\}_{i\in I} \qquad \overline{q\&\{l_i : T_i\}_{i\in I}} = q\oplus\{l_i : \overline{T_i}\}_{i\in I}$$

New typing rules, $\Gamma \vdash P$

$$\frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i\in I} \qquad \Gamma_2 + x : T_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \rhd \{l_i : P_i\}_{i\in I}} \quad \text{[T-Branch]}$$

$$\frac{\Gamma_2 \vdash x : q\oplus\{l_i : T_i\}_{i\in I} \qquad \Gamma_2 + x : T_j \vdash P \qquad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l_j.P} \quad \text{[T-Sel]}$$

Fig. 9. Choice.

interpreted as non-ordered records; we do not distinguish $\&\{l : T, m : U\}$ from $\&\{m : U, l : T\}$. The two new pretypes are dual to each other as described in Fig. 9.

To type check a branching process prefixed by x at type $\&\{l_i : T_i\}_{i\in I}$, rule [T-Branch] checks each of the possible continuations P_i at $x : T_i$. We use the exact same Γ_2 in all cases for only one of the P_i will be executed, similarly to rule for the conditional process, [T-If] in Fig. 6. If rule [T-Branch] introduces an external choice type $\&\{l_i : T_i\}_{i\in I}$, rule [T-Sel] eliminates the dual, internal choice type $\oplus\{l_i : T_i\}_{i\in I}$. To type check a process selecting label l_j on channel x at type $\oplus\{l_i : T_i\}_{i\in I}$, we have to type check the continuation process at the correspondent type $x : T_j$. In both cases, and similarly to the rules for output and input in Fig. 6, context update $\Gamma + x : T$ must be defined.

Below are some examples. The first two illustrate the case when the selected label l is in the corresponding branching process. The third requires a type of a peculiar form. Given that x_1 occurs in three different threads, its type must be unrestricted. Since we have l- and m-labeled messages, we know that the type T for x_1 must be of the form $\mathsf{un}\oplus\{l : T_1, m : T_2\}$. Looking at the context update operation in both rules [T-Sel] and [T-Branch], we realize that both T_1 and T_2 must be equal to T, hence T must be equal to $\mu a.\mathsf{un}\oplus\{l : a, m : a\}$. Similarly, the type for variable x_2 must be equal to $\mu b.\mathsf{un}\&\{l : b, m : b\}$. Following the short form proposed in Section 3 for unrestricted input and output types, these two choice types can be abbreviated to $*\oplus\{l, m\}$ and $*\&\{l, m\}$, respectively. Unrestricted choice types are not known in the literature of session types. They are however present in a variant of the pi calculus where choice and output (and branch and input) form an atomic operation [3,21]. The last three cases represent obvious violations to the expectations of the two threads involved.

$$x_1 \lhd l \mid x_2 \rhd \{l : \mathbf{0}\} \quad \checkmark$$
$$x_1 \lhd l \mid x_2 \rhd \{l : \mathbf{0}, m : \mathbf{0}\} \quad \checkmark$$
$$x_1 \lhd l \mid x_1 \lhd m \mid x_1 \lhd m \mid x_2 \rhd \{l : \mathbf{0}, m : \mathbf{0}\} \quad \checkmark$$
$$\overline{x_1}\ \mathsf{true} \mid x_2 \rhd \{l : \mathbf{0}\} \quad \times$$
$$x_1 \lhd l \mid x_2(z) \quad \times$$
$$x_1 \lhd l \mid x_2 \rhd \{m : \mathbf{0}\} \quad \times$$

For a more concrete example, imagine a data structure mapping elements from some type key to a type value. Among its various operations one finds *put* and *get*. To *put* key k and its associated value v on a x_1-map one writes:

$$x_1 \lhd put.\overline{x_1}\,k.\overline{x_1}\,v$$

To get a value from a map one sends a key and expects a value back, but only if the key is in the data structure. If not then one should be notified of the fact. We use labels *some* and *none* to annotate the result of the *get* operation. Further, in case the key is in the map, we expect a value as well. Here is a client that runs process P if the key is in the map, and runs Q otherwise.

$$x_1 \lhd get.\overline{x_1}\,k.x_1 \rhd \{some\colon x_1(y).P, none\colon Q\}$$

The type of the map, as seen from the side of the client, that is the type of variable x_1, is as follows.

$$\oplus\{put\colon \mathsf{!key.!value.end},\ get\colon \mathsf{!key.\&}\{some\colon \mathsf{?value.end},\ none\colon \mathsf{end}\}\}$$

We take the opportunity to discuss *session initiation* [4]. Looking at the type above, it should be obvious that the map can only be used once, either to read or to write. Useful maps are to be used multiple times, possibly by different clients. As such maps must answer on shared channels. A shared channel, known to all clients, is used to establish individual sessions as follows. Each client creates a channel, x_1x_2, passes one end, x_2, to the map server, and retains the other end, x_1, for interaction. The code for a writing client is then as follows.

$$(\nu x_1x_2)(\overline{map_1}\, x_2 \mid x_1 \lhd put.\, \overline{x_1}\, k.\, \overline{x_2}\, v)$$

The map server is a replicated process that receives a linear channel end on which it conducts the session.

$$\mathsf{un}\ map_2(y).y \rhd \{get\colon y(k)\ldots, put\colon y(k).y(v)\ldots\}$$

If we denote by T the type of variable x_1 above, then the type for the map is $*!T$ for the client (variable map_1) and $*?T$ for the server (variable map_2), as expected.

For an example with a recursive linear type, consider an iterator of boolean values—a process that offers operations *hasNext* and *next* repeatedly until *hasNext* returns "no". Further suppose that the iterator accepts requests at x_2, so that clients write at x_1, the other channel end. A client that reads and discards every value from the iterator can be written as follows,

$$\mathsf{un}\ loop_2(y).y \lhd hasNext.y \rhd \{yes\colon y \lhd next.y(z).\overline{loop}\, y,\ no\colon \mathbf{0}\} \mid \overline{loop_1}\, x_2 \qquad \checkmark$$

but not as

$$\mathsf{un}\ loop_2(y).x_2 \lhd hasNext.x_2 \rhd \{yes\colon x_2 \lhd next.x_2(z).\overline{loop}\, y,\ no\colon \mathbf{0}\} \mid \overline{loop_1}\ \mathsf{true} \qquad \times$$

for x_2 is a linear variable, hence cannot occur free underneath replication (cf. rule [T-In] in Fig. 8). Clearly, the communication pattern of the iterator, as seen by the client at variable x_2, is of the form

$$\oplus\{hasNext\colon \&\{no\colon \mathsf{end},\ yes\colon \oplus\{next\colon \mathsf{!bool.}\oplus\{hasNext\colon \&\{\ldots\}\}\}\}\}$$

which can be written in finite form as follows.

$$\mu a.\oplus\{hasNext\colon \&\{no\colon \mathsf{end},\ yes\colon \oplus\{next\colon \mathsf{!bool.}a\}\}\}$$

Notice that the type in the equation above is equivalent to the following,

$$\oplus\{hasNext\colon \mu b.\&\{no\colon \mathsf{end},\ yes\colon \oplus\{next\colon \mathsf{!bool.}\oplus\{hasNext\colon b\}\}\}\}$$

and that the two types can never be made syntactically equal by finite expansion alone. Yet we would not like to distinguish them, for they have the same infinite expansion; this is another reason to use an equi-recursive view of types.

The pi calculus is known by its flexibility to describe computational idioms. While in possession of branching and recursive types, we can get away without primitive boolean values altogether; if fact we do not need any primitive type. If we fix two variables t_1, t_2 for the truth value true and f_1, f_2 for false, by taking advantage of the encoding of generic replication, $!P$, introduced at the end of Section 5, and by introducing the following abbreviations,

True abbreviates $!(t_1 \lhd true)$

False abbreviates $!(f_1 \lhd false)$

$\mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q$ abbreviates $x \rhd \{true\colon P, false\colon Q\}$

then we can easily see that

$$True \mid False \mid \mathsf{if}\ t_2\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow\!\!\!\!\!\rightarrow True \mid False \mid P$$

and similarly for the false case. Milner [13] introduced an alternative encoding that does not rely on choice. In a reduction similar to the above, a residual, inert, process (not structurally equivalent to $\mathbf{0}$) for the false case is left in the contractum, which must be "removed" via a process equivalence, which we manage to avoid in our proposal. Milner's encoding would nevertheless be typable in our system.

7. Main results

This section looks at the guarantees offered by typable processes.

Many examples of ill-formed processes are presented in the previous sections; we now try to systematize them. Our ill formed processes fall in three categories. For boolean values we have conditional processes whose value in the test is neither true nor false. For the communication primitives we have two threads sharing a variable but using it with distinct interaction patterns (input, output, select or branch), and two threads each possessing a co-variable, but using them in non-dual patterns.

$$\mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q \qquad \times$$

$$\overline{a}\,\mathsf{true} \mid a(z) \qquad \times$$

$$(\nu xy)\big(\overline{x}\,\mathsf{true} \mid y \rhd \{l_i : P_i\}_{i\in I}\big) \qquad \times$$

We say that processes of the form $\overline{x}\,v.P$, $qx(y).P$, $x \lhd l.P$, and $x \rhd \{l_i : P_i\}_{i\in I}$ are *prefixed at variable x*. We call *redexes* to processes of the form $\overline{x}\,v.P \mid qy(z).Q$ and $x \lhd l_j.P \mid y \rhd \{l_i : P_i\}_{i\in I}$ with $j \in I$. We then say that a process is *well formed* if, for each of its structural congruent processes of the form $(\nu x_1 y_1)\ldots(\nu x_n y_n)(P \mid Q \mid R)$ with $n \geqslant 0$, the following conditions hold.

- If P is of the form $\mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, then v is either true or false, and
- if P and Q are processes prefixed at the same variable, then they are of the same nature (input, output, branch, selection), and
- if P is prefixed at x_1 and Q is prefixed at y_1 then $P \mid Q$ is a redex.

Typable processes are not necessarily well formed. Process $\mathsf{if}\ x\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$ is typable under context $x : \mathsf{bool}$, yet we consider it an error for x is not a boolean value. But if P is closed (hence typable under the empty context, by strengthening, Lemma 7.3) and x is bound by a (νxy) binder, then rule [T-Res] introduces two dual types in the context, $x : T$, $y : \overline{T}$, where T is necessarily different from bool, for duality would not be defined otherwise. For the second case $\overline{x}\,v.\mathbf{0}$ and $x \lhd l.P$ are not typable under any context, and similarly for $\overline{x_1}\,v.\mathbf{0}$ and $y_1 \lhd l.P$ since the scope restriction $(\nu x_1 y_1)$ requires x_1 and y_1 to be used in dual mode.

The main result of our system says that typable closed processes do not reduce to ill formed processes.

Theorem 7.1 *(Main result). If $\vdash P$ and P reduces to Q in zero or more steps, then Q is well formed.*

As usual this result follows from two other results: type preservation and type safety.

Theorem 7.2 *(Preservation). If $\Gamma \vdash P$ and $P \to Q$ then $\Gamma \vdash Q$.*

Theorem 7.3 *(Safety). If $\vdash P$ then P is well formed.*

The proof of the main theorem follows by induction on the length of reduction. For the base case we use type safety; for the induction step we use preservation. The rest of this section is dedicated to the proofs of Theorems 7.2 and 7.3.

We start by introducing some basic properties of context split. Let $\mathrm{dom}(\Gamma)$ denote the set of variables x such that $x : T$ is in Γ, and let $\mathcal{U}(\Gamma)$ denote the typing context containing exactly the entries $x : T$ in Γ such that $\mathsf{un}(T)$, and similarly for $\mathcal{L}(\Gamma)$ and the lin predicate.

Lemma 7.1 *(Properties of context split). Let $\Gamma = \Gamma_1 \circ \Gamma_2$.*

1. $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.
2. *If $x : \mathsf{lin}\ p \in \Gamma$ then either $x : \mathsf{lin}\ p \in \Gamma_1$ and $x \notin \mathrm{dom}\,\Gamma_2$, or $x : \mathsf{lin}\ p \in \Gamma_2$ and $x \notin \mathrm{dom}\,\Gamma_1$.*
3. $\Gamma = \Gamma_2 \circ \Gamma_1$.
4. *If $\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

Proof. A straightforward induction on the structure of context Γ. □

From the above properties many other facts can be derived, including $\Gamma = \Gamma_1, \mathcal{L}(\Gamma_2)$, $\Gamma = \Gamma \circ \mathcal{U}(\Gamma)$, and $\Gamma = \Gamma_2$ when $\mathsf{un}(\Gamma_1)$.

We now present two basic properties of our type system: weakening and strengthening. *Weakening* allows introducing new unrestricted entries in a typing context. The result becomes useful in situations where we need context entries for variables not free in the process. Obviously the result does not hold for linear types; for example, $\vdash \mathbf{0}$ but $x : \mathsf{lin}\ ?\mathsf{bool} \not\vdash \mathbf{0}$.

Lemma 7.2 *(Unrestricted weakening). If* $\Gamma \vdash P$ *and* $\mathsf{un}(T)$ *then* $\Gamma, x : T \vdash P$.

Proof. The proof follows by induction on the structure of the derivation. We need to establish a similar result for values, whose proof is a simple case analysis on the three value typing rules. The hypothesis $\mathsf{un}(\Gamma)$ in rule [T-Inact] establishes the base case; the others follow by a straightforward induction. □

Strengthening allows to remove extraneous entries from the context, but only when the variable does not occur free in the process. We use the result when we need to remove context entries for variables not free in the process, usually introduced by a context split operation on an unrestricted type, as for example, when showing that if $\Gamma \vdash (\nu xy)(P \mid Q)$ and $x \notin \mathrm{fv}(Q)$ and x is typed at an unrestricted type, then $\Gamma \vdash (\nu xy)P \mid Q$. Clearly strengthening applies to entries $x : T$ where x is not free in the process and T is unrestricted. If x is free in the process, then an entry for x is certainly required in the typing context: $\not\vdash x(y)$; if on the other hand T is linear then x must be free in the process: $x : \mathsf{lin?bool} \not\vdash \mathbf{0}$.

Lemma 7.3 *(Strengthening). Let* $\Gamma \vdash P$ *and* $x \notin \mathrm{fv}(P)$.

1. $x : \mathsf{lin}\ p \notin \Gamma$.
2. *If* $\Gamma = \Gamma', x : T$ *then* $\Gamma' \vdash P$.

Proof. The proof is by induction on the structure of the derivation. Again we have to establish a similar result for values. The hypothesis $\mathsf{un}(\Gamma)$ in rule [T-Inact] establishes the base case; the other cases follow by a straightforward induction. □

The next lemma states that structural equivalent processes can be typed under the same contexts, and is used in the proof of type preservation.

Lemma 7.4 *(Preservation for* $\equiv$*). If* $\Gamma \vdash P$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q$.

Proof. The proof is by a simple analysis of derivations for each member of each axiom. We use weakening and strengthening (Lemmas 7.2 and 7.3), and check both directions of each axiom.

The most elaborate case is scope restriction. To show that, if $\Gamma \vdash (\nu xy)P \mid Q$ then $\Gamma \vdash (\nu xy)(P \mid Q)$, we start by building the only derivation for $\Gamma \vdash (\nu xy)P \mid Q$, to conclude that $\Gamma = \Gamma_1 \circ \Gamma_2$, $\Gamma_1, x : T, y : \overline{T} \vdash P$, and $\Gamma_2 \vdash Q$. To build a derivation for the conclusion we start with $\Gamma_2 \vdash Q$ and distinguish two cases. If T is linear, then $(\Gamma_1, x : T, y : \overline{T}) \circ \Gamma_2 = \Gamma_1 \circ \Gamma_2, x : T, y : \overline{T}$; otherwise use weakening to conclude that $\Gamma_2, x : T, y : \overline{T} \vdash Q$ and $\Gamma_1 \circ \Gamma_2, x : T, y : \overline{T} = (\Gamma_1, x : T, y : \overline{T}) \circ (\Gamma_2, x : T, y : \overline{T})$. In either case complete the proof with rules [T-Res] and [T-Par].

In the reverse direction, to show that if $\Gamma \vdash (\nu xy)(P \mid Q)$ then $\Gamma \vdash (\nu xy)P \mid Q$, we start by building the only derivation for $\Gamma \vdash (\nu xy)(P \mid Q)$ to conclude that $\Gamma, x : T, y : \overline{T} = \Gamma_1 \circ \Gamma_2$, $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. To build a derivation for the conclusion we distinguish two cases. If T is linear, then $x : T$ is either in Γ_1 or in Γ_2, but not in both (properties of context split). Given that $x \notin \mathrm{fv}(Q)$, strengthening gives us that $x : T \notin \Gamma_2$, hence $x : T \in \Gamma_1$, and similarly for y and $\overline{T}$. Hence $\Gamma_1 = \Gamma_1', x : T, y : \overline{T}$. If, on the other hand T is unrestricted, we know that $\Gamma_1 = \Gamma_1', x : T, y : \overline{T}$ and $\Gamma_2 = \Gamma_2', x : T, y : \overline{T}$, and we apply strengthening to obtain $\Gamma_2' \vdash Q$. In either case we conclude the proof with rule [T-Res] and [T-Par]. □

Inversion of the value typing relation is a simple result that we use often in the proofs of the substitution lemma (below) and type preservation. Even though we could establish a similar result for processes, we do that 'on the fly' within proofs, when required.

Lemma 7.5 *(Inversion of the value typing relation).*

1. *If* $\Gamma \vdash \mathsf{true} : T$ *then* $T = \mathsf{bool}$ *and* $\mathsf{un}(\Gamma)$.
2. *If* $\Gamma \vdash \mathsf{false} : T$ *then* $T = \mathsf{bool}$ *and* $\mathsf{un}(\Gamma)$.
3. *If* $\Gamma \vdash x : T$ *then* $\Gamma = \Gamma_1, x : T$ *and* $\mathsf{un}(\Gamma_1)$.

Proof. A simple analysis of the typing axioms involved. □

The *substitution lemma* plays a central role in proof of type preservation (Theorem 7.2). The result is not applicable when $x = v$ and $\mathsf{un}(T)$ since there is no Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$ given that $x : T \in \Gamma_1$ but $x : U \notin \Gamma_2$, for all type U.

Lemma 7.6 *(Substitution). If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *then* $\Gamma \vdash P[v/x]$.

Proof. The proof is by induction on the structure of P and uses the properties of context split strengthening and weakening (Lemmas 7.1, 7.2 and 7.3). This is the most elaborate proof in this section.

For the base case (process $\mathbf{0}$), we know that $\mathsf{un}(\Gamma_2)$ and $\mathsf{un}(T)$. Given $\mathsf{un}(T)$, by inversion of the value typing relation, we know that $\mathsf{un}(\Gamma_1)$. From the basic properties of context split we obtain $\mathsf{un}(\Gamma)$, hence $\Gamma \vdash \mathbf{0} = \mathbf{0}[v/x]$.

For each inductive case there are a few cases to consider. For parallel composition we distinguish $\mathsf{un}(T)$ and $\mathsf{lin}(T)$; for the conditional we distinguish processes of the form $\mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q$ from $\mathsf{if}\ u\ \mathsf{then}\ P\ \mathsf{else}\ Q$ with $u \neq x$; for the output process we distinguish four cases: $\overline{x}u.P$ with $\mathsf{un}(T)$ and with $\mathsf{lin}(T)$, and $\overline{z}u.P$ $(z \neq z)$ with $\mathsf{un}(T)$ and with $\mathsf{lin}(T)$; and similarly for selection and branching. Finally, for input processes we distinguish six cases: $\mathsf{un}\, x(y).P$, $\mathsf{un}\, z(y).P$ $(z \neq x)$, $\mathsf{lin}\, x(y).P$ with $\mathsf{un}(T)$ and with $\mathsf{lin}(T)$, and $\mathsf{lin}\, z(y).P$ $(z \neq x)$ with $\mathsf{un}(T)$ and with $\mathsf{lin}(T)$.

All the inductive cases follow the same pattern; we detail one: $\mathsf{lin}\, x(y).p$ and $\mathsf{lin}(T)$. From $\Gamma_2, x : T \vdash \mathsf{lin}\, x(y).p$ and rule [T-In], we know that $\Gamma_2 = \Gamma_2' \circ \Gamma_2''$ and $\Gamma_2', x : T \vdash x : T$ and $T = \mathsf{lin!}T_1.T_2$ and $\Gamma_2'' + x : T_2 \vdash P$. Inversion of the value typing relation gives us that $\mathsf{un}(\Gamma_2')$ and the properties of context split that $\Gamma_2'' = \Gamma_2$. Since x is not in Γ_2 we have that $\Gamma_2'' + x : T_2 = \Gamma_2, x : T_2$, hence $(\Gamma_2, y : T_1), x : T_2 \vdash P$. Inversion of the value typing relation also gives us that $\Gamma_1 = \Gamma_1', x : T$ and $\mathsf{un}(\Gamma_1')$.

We now distinguish four cases depending on the linear/unrestricted nature of T_1 and T_2. One extreme is when both types are linear. Rule [T-Var] gives us that $\Gamma_1', v : T_2 \vdash v : T_2$ and we have that $\Gamma' = (\Gamma_1', v : T_2) \circ (\Gamma_2, y : T_1) = \Gamma_1' \circ \Gamma_2, v : T_2, y : T_1$. The induction hypothesis is $\Gamma' \vdash P[v/x]$. Since v is not in $\Gamma_1' \circ \Gamma_2$, we have $\Gamma_1' \circ \Gamma_2 + v : T_2, y : T_1 \vdash P[v/x]$. It should be easy to see that $\Gamma = \Gamma_1 \circ (\Gamma_1' \circ \Gamma_2)$, hence rule [T-In] gives $\Gamma \vdash \mathsf{lin}\, v(y).P[v/x] = (\mathsf{lin}\, x(y).P)[v/x]$ as needed.

The other extreme is when both types are unrestricted. In this case, since $v : T \in \Gamma_1$ we know that $v : T \in \Gamma_2$ as well. Rule [T-Var] gives $\Gamma_1, y : T_1 \vdash v : T$ and we have $\Gamma' = (\Gamma_1, y : T_1) \circ (\Gamma_2, y : T_1) = \Gamma, y : T_1$. The induction hypothesis is $\Gamma' \vdash P[v/x]$, hence $(\Gamma + v : T), y : T_1 \vdash P[v/x]$. Basic facts on context split give us that $\Gamma = \Gamma_1 \circ \Gamma$, hence rule [T-In] gives $\Gamma \vdash \mathsf{lin}\, v(y).P[v/x] = (\mathsf{lin}\, x(y).P)[v/x]$ as needed. The remaining two cases—lin/un and un/lin—are similar. □

We are finally in a position to prove the main results, type preservation and type safety.

Proof of Theorem 7.2. The proof is by induction on the reduction derivation, and uses weakening and substitution (Lemmas 7.2 and 7.6). The inductive cases are straightforward; we use Lemma 7.4 in case [R-Struct].

The most interesting cases are when the derivation of the reduction step ends with rule [R-LinCom] or [R-UnCom]. We sketch the first. Suppose that [T-Res] introduces $x : q!T.U, y : q?T.\overline{U}$. Building the only tree for the hypothesis, we know that $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \circ \Gamma_4$ where $\Gamma_3 \vdash R$. At this point we distinguish two cases depending on the nature of qualifier q. If linear then we have $\Gamma_1, x : U \vdash P$ and $\Gamma_2, z : T, y : \overline{U} \vdash Q$ and $\Gamma_4 \vdash v : T$. From $\Gamma_4 \vdash v : T$ and $\Gamma_2, z : T, y : \overline{U} \vdash Q$ we use the substitution lemma to obtain $\Gamma_4 \circ \Gamma_2, y : \overline{U} \vdash Q[v/z]$. We then conclude the proof with rules [T-Par], [T-Par], [T-Res].

If q is unrestricted, we know that $(\Gamma_1, x : q!T.U) + x : U \vdash P$, and $(\Gamma_2, y : q?T.\overline{U}, z : T) + y : \overline{U} \vdash Q$ and $\Gamma_4, x : q!T.U \vdash v : T$. The first context split operation is defined only when $q!T.U$ is U, and the second when $q?T.\overline{U}$ is $\overline{U}$. Then we use weakening four times: to go from $\Gamma_1, x : U \vdash P$ to $\Gamma_1, x : U, y : \overline{U} \vdash P$, from $\Gamma_2, z : T, y : \overline{U} \vdash Q$ to $\Gamma_2, z : T, x : U, y : \overline{U} \vdash Q$, from $\Gamma_3 \vdash R$ to $\Gamma_3, x : U, y : \overline{U} \vdash R$, and from $\Gamma_4, x : U \vdash v : T$ to $\Gamma_4, x : U, y : \overline{U} \vdash v : T$. Using substitution, we conclude the proof as in the case of q linear. □

Proof of Theorem 7.3. The proof is by contradiction. We build the only derivation for $\vdash (\nu x_1 y_1) \ldots (\nu x_n y_n)(P \mid Q \mid R)$ to obtain that $x_1 : T_1, y_1 : \overline{T_1}, \ldots x_n : T_n,\ y_n : \overline{T_n} = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ and $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. For each case in the definition of well-formed process, a simple analysis of the hypothesis shows that the process is not typable.

For example, suppose P is $x_1(z).P'$ and Q is $x_1 \lhd l.Q'$. In order to have both $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$, we know that T_1 is unrestricted and that $x_1 : T_1$ is both in Γ_1 and in Γ_2. But rule [T-In] requires T_1 to be of the form $q?U.V$, whereas [T-Sel] asks for a type of the form $\oplus\{l_i : T_i\}_{i \in I}$.

For an example of a redex, suppose that P is $x_1(z).P'$ and Q is $y_1 \lhd l.Q'$. Rule [T-In] requires T_1 to be of the form $q?U.V$, whereas [T-Sel] asks for a type $\overline{T}_1$ of the form $\oplus\{l_i : T_i\}_{i \in I}$, which cannot possibly be fulfilled. □

8. Algorithmic type checking

The typing rules provided in the previous sections give a concise specification of what we understand by well-formed programs. They cannot however be implemented directly for two main reasons. One is the difficulty of implementing the non-deterministic splitting operation, $\Gamma = \Gamma_1 \circ \Gamma_2$, for we must guess how to split an incoming context Γ in two parts. The other is the problem of guessing the types to include in the context when in presence of scope restriction (rule [T-Res] in Fig. 6).

To solve the first problem, we restructure the type checking rules to avoid having to guess context splitting. To address the second difficulty we seek the help of programmers by requiring explicit annotations in the scope restriction constructor. We now write $(\nu xy : T)P$, where x is supposed to be typed at T and y at type $\overline{T}$ in process P. Changes are in Fig. 10. We assume that type equivalence is decidable, and use letter L to denote a set of variables.

The central idea of the new type checking system is that, rather than splitting the input context into two (or three) parts before checking a complex process, we pass the entire context to the first subprocess (or value) and have it return the unused part. This output is then passed to the second subprocess, which in turn returns the unused portion of the context, and so on. The output of the last subprocess is then the output of the process under consideration. Sequents are now of

New syntactic forms

$$P ::= \ldots \qquad \text{Processes:}$$
$$(\nu xy : T)P \qquad \text{annotated scope restriction}$$

Context difference, $\Gamma \div L = \Gamma$

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div L = \Gamma_2, x : T \qquad \mathsf{un}(T)}{\Gamma_1 \div (L, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div L = \Gamma_2 \qquad x \notin \mathrm{dom}(\Gamma_2)}{\Gamma_1 \div (L, x) = \Gamma_2}$$

Typing rules for values, $\Gamma \vdash v : T; \Gamma$

$$\Gamma \vdash \mathsf{true} : \mathsf{bool}; \Gamma \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool}; \Gamma \qquad \text{[A-True] [A-False]}$$

$$\frac{\mathsf{un}(T)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2)} \qquad \Gamma_1, x : \mathsf{lin}\, p, \Gamma_2 \vdash x : \mathsf{lin}\, p; (\Gamma_1, \Gamma_2) \qquad \text{[A-UnVar] [A-LinVar]}$$

Typing rules for processes, $\Gamma \vdash P : \Gamma; L$

$$\Gamma \vdash \mathbf{0} : \Gamma; \emptyset \qquad \frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \qquad \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P \mid Q : \Gamma_3; L_2} \qquad \text{[A-Inact] [A-Par]}$$

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L}{\Gamma_1 \vdash (\nu xy : T)P : \Gamma_2 \div \{x, y\}; L \setminus \{x, y\}} \qquad \text{[A-Res]}$$

$$\frac{\Gamma_1 \vdash v : q\,\mathsf{bool}; \Gamma_2 \qquad \Gamma_2 \vdash P : \Gamma_3; L \qquad \Gamma_2 \vdash Q : \Gamma_3; L}{\Gamma_1 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Gamma_3; L} \qquad \text{[A-If]}$$

$$\frac{\Gamma_1 \vdash x : q!T.U; \Gamma_2 \qquad \Gamma_2 \vdash v : T; \Gamma_3 \qquad \Gamma_3 + x : U \vdash P : \Gamma_4; L}{\Gamma_1 \vdash \overline{x}\, v.P : \Gamma_4; L \cup (\mathsf{if}\ q = \mathsf{lin}\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \emptyset)} \qquad \text{[A-Out]}$$

$$\frac{\Gamma_1 \vdash x : q_2?T.U; \Gamma_2 \qquad (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L \qquad q_1 = \mathsf{un} \Rightarrow L = \emptyset}{\Gamma_1 \vdash q_1 x(y).P : \Gamma_3 \div \{y\}; L \setminus \{y\} \cup (\mathsf{if}\ q_2 = \mathsf{lin}\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \emptyset)} \qquad \text{[A-In]}$$

$$\frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i \in I}; \Gamma_2 \qquad \Gamma_2 + x : T_i \vdash P_i : \Gamma_3; L \qquad \forall i \in I}{\Gamma_1 \vdash x \rhd \{l_i : P_i\}_{i \in I} : \Gamma_3; L \cup (\mathsf{if}\ q = \mathsf{lin}\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \emptyset)} \qquad \text{[A-Branch]}$$

$$\frac{\Gamma_2 \vdash x : q\oplus\{l_i : T_i\}_{i \in I}; \Gamma_2 \qquad \Gamma_2 + x : T_j \vdash P : \Gamma_3; L \qquad j \in I}{\Gamma_1 \vdash x \lhd l_j.P : \Gamma_3; L \cup (\mathsf{if}\ q = \mathsf{lin}\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \emptyset)} \qquad \text{[A-Sel]}$$

Fig. 10. Algorithmic type checking.

forms $\Gamma_1 \vdash v : T; \Gamma_2$ for values and $\Gamma_1 \vdash P : \Gamma_2; L$ for processes, with the understanding that Γ_1, v and P form the input to the algorithm and T, Γ_2, and L is the output. Set L collects linear (free) variables in P that occur in subject position, and plays its role in the rule for parallel composition. A variable x occurs in subject position in processes $\overline{x}\, v.P$, $x(y).P$, $x \lhd l.P$ and $x \rhd \{l_i : P_i\}_{i \in I}$.

The base cases for variables and constants allow any context to pass through the judgment, even when containing linear types. Two rules, [A-UnVar] and [A-LinVar], replace the single rule for variables [T-Var] in Fig. 6. The former keeps the entry $x : T$ in the returned context, the latter removes the entry.

The assumptions for unrestricted types are never consumed, as the following example shows.

$$x : *!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x : *!\mathsf{bool}); \emptyset \qquad \checkmark$$

For linear assumptions three things can happen: they may remain (used or not), they may disappear altogether or they may become unrestricted.

$$x : !\mathsf{bool}.!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x : !\mathsf{bool}); \{x\} \qquad \checkmark$$
$$x : !\mathsf{bool} \vdash \mathbf{0} : (x : !\mathsf{bool}); \emptyset \qquad \checkmark$$
$$x : !\mathsf{bool}, y : *!(!\mathsf{bool}) \vdash \overline{y}\, x : \big(y : *!(!\mathsf{bool})\big); \{y\} \qquad \checkmark$$
$$x : !\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x : \mathsf{end}); \{x\} \qquad \checkmark$$

The above examples motivate rule [A-Par]. The output of the first subprocess P cannot be directly passed to the second subprocess Q; a rule of the form

$$\frac{\Gamma_1 \vdash P : \Gamma_2 \qquad \Gamma_2 \vdash Q : \Gamma_3}{\Gamma_1 \vdash P \mid Q : \Gamma_3}$$

would allow to derive

$$x : !\mathsf{bool}.!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{false} : (x : \mathsf{end}) \qquad \times$$
$$x : !\mathsf{bool}, y : *!\mathsf{end} \vdash \overline{x}\,\mathsf{true} \mid \overline{y}\, x : (x : \mathsf{end}, y : *!\mathsf{end}) \qquad \times$$

but we know that $x : !\mathsf{bool}.!\mathsf{bool}. \not\vdash \overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{false}$ and that $x : !\mathsf{bool}, y : *!\mathsf{end} \not\vdash \overline{x}\,\mathsf{true} \mid \overline{y}\,x$. Instead, we collect in set L_1 all linear (free) subjects in process P and use context difference to ensure that they do not remain linear in context Γ_2. The *context difference* operator, $\div$, defined in Fig. 10 both checks that linear variables do not appear in contexts and removes unrestricted variables. In the figure, notation (L, x) denotes the set $L \cup \{x\}$ where $x \notin L$. Notice that this operator is undefined when we try to remove a variable of a linear type from a context. Type checking continues with process Q in a context where the assumptions for the (unrestricted) names in L_2 have been removed.

Using the context difference operation we can quickly check that the algorithm does not succeed on the two processes above. In the first case, we have $x : !\mathsf{bool}.!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x : !\mathsf{bool}), \{x\}$, but $(x : !\mathsf{bool}) \div \{x\}$ is not defined since the type of x is linear. In the second case, we have $x : !\mathsf{bool}, y : *!\mathsf{end} \vdash \overline{x}\,\mathsf{true} : (x : \mathsf{end}, y : *!\mathsf{end}), \{x\}$ and $(x : \mathsf{end}, y : *!\mathsf{end}) \div \{x\} = (y : *!\mathsf{end})$, but the goal $y : *!\mathsf{end} \vdash \overline{y}\,x : \_; \_$ does not succeed, for x is not in the domain of the input context.

Rule [A-Res] ensures that newly introduced linear variables are used to the end. The premise $\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L_2$ introduces variables x and y in the context. If T is linear, then x must be used in P and should not appear in Γ_2 in linear form (it may however still show in unrestricted form). If T is unrestricted, then x certainly appears in Γ_2. The case for y is similar. Unrestricted types for x, y must be deleted from the rule's outgoing context. Using the context difference operator, the outgoing context is $\Gamma_2 \div \{x, y\}$. Because x and y are bound, the rule also removes variables x, y from the set L of (free) variables in subject position.

Rule [A-Out] searches the incoming context Γ_1 for the type of x. Then uses Γ_2, the remaining portion of Γ_1, to type check value v, thus obtaining a type T (which must match the input part of the type for x) and a new context Γ_3. This context is then updated with the new type for x at the continuation type U, and passed to the subprocess P. Similarly to rule [T-Out] in Fig. 6, when $q = \mathsf{lin}$ then x is not in Γ_3 and a new assumption for x is introduced in the context; else when $q = \mathsf{un}$ we must have $q!T.U = U$. The rule outputs a context Γ_4 resulting from type checking the continuation P as well as the set of variables L_4 thus obtained, enriched with subject x if linear.

Rule [A-In] should be easy to understand based on the description of rules [A-Res] and [A-Out]. Similarly to [A-Out] we look in the input context the type of x. We then pass to subprocess P the unused portion of the context together with two new assumptions, for x and for y. In the end, if y remains in the context then it must be unrestricted. Once again, the context difference operator both checks that the type of y is not linear and removes it from the outgoing context. Because y is bound, the rule removes it from the set L of free variables in subject position, and adds subject x if linear (as in rule [A-Out]). The case of replication, $q_1 = \mathsf{un}$, ensures that there are no (free) subjects on linear channels in process P by requiring an empty set L of free subjects.

Each rule in the algorithm is syntax directed. Furthermore all auxiliary functions, including type equality, context membership, context difference, and context restriction are computable. We still need to check that this system is equivalent to the more elegant system introduced in the previous sections.

The algorithmic type system in Fig. 10 is equivalent to the type system introduced gradually in Sections 3 to 6. Notice however that the two type systems talk about different languages, languages that differ in the annotation in the scope restriction constructor. To obtain a non-annotated process from an annotated one, we use function $\mathrm{erase}(P)$ that removes all types from an annotated process P. Function erase is a homomorphism everywhere, except at scope restriction where $\mathrm{erase}((\nu xy : T)P) = (\nu xy)(\mathrm{erase}(P))$. Algorithmic correctness says that if the algorithm succeeds on input (Γ, P) then $\Gamma \vdash P$, but only if the output of the call contains no linear type. For example $x : \mathsf{lin}\, p \vdash \mathbf{0} : (x : \mathsf{lin}\, p); \emptyset$ but we know that $x : \mathsf{lin}\, p \not\vdash \mathbf{0}$.

Theorem 8.1 *(Algorithmic correctness).* $\Gamma_1 \vdash P : \Gamma_2; \_$ *and* $\mathsf{un}(\Gamma_2)$ *if and only if* $\Gamma_1 \vdash \mathrm{erase}(P)$.

The rest of this section is dedicated to the proof of the above theorem. We start with a few properties of the context difference operation. Let $\Gamma \setminus L$ denote context Γ with entries $x : T$ removed, for $x \in L$. Recall that $\mathcal{L}(\Gamma)$ is the typing context containing exactly the entries $x : T$ in Γ such that $\mathsf{lin}(T)$.

Lemma 8.1 *(Properties of context difference). Let* $\Gamma \div L = \Gamma'$.

1. $\Gamma' = \Gamma \setminus L$.
2. $\mathcal{L}(\Gamma) = \mathcal{L}(\Gamma')$.
3. *If* $x : T \in \Gamma$ *and* $x \in L$ *then* $\mathsf{un}(T)$ *and* $x \notin \mathrm{dom}(\Gamma')$.

Proof. A simple induction on the size of set L in the first two cases. We detail the second. When $L = \emptyset$, we have $\Gamma = \Gamma'$, hence $\mathcal{L}(\Gamma) = \mathcal{L}(\Gamma')$. When L is (L', x), by definition we know that $\Gamma \div L'$ is $\Gamma', x : T$ and $\mathsf{un}(T)$, or is Γ' and $x \notin \mathrm{dom}(\Gamma')$. By induction, $\mathcal{L}(\Gamma)$ is $\mathcal{L}(\Gamma', x : T)$ in the former case, and is $\mathcal{L}(\Gamma')$ in the latter. Conclude the case by noting that $\mathcal{L}(\Gamma', x : T) = \mathcal{L}(\Gamma')$ since $\mathsf{un}(T)$.

For the third case, if L is (L', x), we know by definition that $\Gamma \div L'$ is $\Gamma', x : T$ and $\mathsf{un}(T)$, or is Γ' and $x \notin \mathrm{dom}(\Gamma')$, as above. The former case follows directly from the definition; for the latter we know that $x \in \mathrm{dom}(\Gamma)$ and $x \notin \mathrm{dom}(\Gamma')$, hence $x \in \mathrm{dom}(\mathcal{U}(\Gamma))$ since $\mathcal{L}(\Gamma) = \mathcal{L}(\Gamma')$. □

From the above properties, many others can be derived, including, $L \cap \mathrm{dom}(\mathcal{L}(\Gamma)) = \emptyset$, $\Gamma' \setminus L' = (\Gamma \setminus L') \div L$, and $(\Gamma, x : T) \div L = \Gamma', x : T$ when $x \notin L$.

Algorithmic monotonicity relates the output context to the input context in a call to the algorithm and is used extensively in the remaining results.

Lemma 8.2 *(Algorithmic monotonicity).*

1. *If* $\Gamma_1 \vdash v : T; \Gamma_2$ *then* $\Gamma_2 \subseteq \Gamma_1$ *and* $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.
2. *If* $\Gamma_1 \vdash P : \Gamma_2; L$ *then* $L \subseteq \mathrm{dom}(\Gamma_1)$, $\Gamma_2 \setminus L \subseteq \Gamma_1$, *and* $\mathcal{U}(\Gamma_2) \setminus L = \mathcal{U}(\Gamma_1)$.

Proof. The case for values follows by a simple inspection of the four axioms involved. The case for processes follows by induction on the structure of the typing derivation and uses basic set theory as well as the properties of context difference above. For example, if the derivation ends with rule [A-In] we distinguish four cases depending on the un/lin nature of q_1 and q_2. Take the case when both are linear. To show that $\mathrm{dom}(\Gamma_3 \div \{x\}) \subseteq \mathrm{dom}(\Gamma_1)$, we start with the induction hypotheses $\mathrm{dom}(\Gamma_3) \subseteq \mathrm{dom}((\Gamma_2, y : T) + x : U)$ and $\Gamma_2 \subseteq \Gamma_1$. Then we know $\mathrm{dom}(\Gamma_3) \div \{y\} \subseteq \mathrm{dom}((\Gamma_2, y : T) + x : U) \div \{y\} = \mathrm{dom}(\Gamma_2, x : U) = \mathrm{dom}(\Gamma_2) \cup \{x\} \subseteq \mathrm{dom}(\Gamma_1) \cup \{x\} = \mathrm{dom}(\Gamma_1)$. The remaining cases are similar in nature. □

The proof of the main result can be broken in two standard parts, *soundness* and *completeness* of the algorithm with respect to the declarative system. We attack each result in turn, soundness first.

The following lemma, algorithmic linear strengthening, is used when the type system splits an input context Γ in two parts and passes each to a different process (rule [T-Par]). In this case the algorithm passes the whole context to the first process and receives back the unused part. In order to prove soundness we need to show that the first process is algorithmically typable in a context not containing the unused linear entries in Γ, for these are used to type the second process. The proviso that x is not in L is important. Take for T the type $\mu\alpha.!\mathsf{bool}.\alpha$. We have $x : T \vdash \overline{x}\,\mathsf{true} : (x : T); \{x\}$ where the type T of x is invariant, but we know that $\emptyset \nvdash \overline{x}\,\mathsf{true} : \emptyset; \_$.

Lemma 8.3 *(Algorithmic linear strengthening).*

1. *If* $\Gamma_1 \vdash v : T; (\Gamma_2, x : \mathsf{lin}\, p)$, *then* $\Gamma_1 = \Gamma_3, x : \mathsf{lin}\, p$ *and* $\Gamma_3 \vdash v : T; \Gamma_2$.
2. *If* $\Gamma_1 \vdash P : (\Gamma_2, x : \mathsf{lin}\, p); L$ *and* $x \notin L$, *then* $\Gamma_1 = \Gamma_3, x : \mathsf{lin}\, p$ *and* $\Gamma_3 \vdash P : \Gamma_2; L$.

Proof. The case for values follows by a simple inspection of the four axioms involved. The case for processes follows by induction on the structure of the derivation and uses the properties of context difference and monotonicity. We detail two cases.

When the derivation ends with the [A-Par] rule, suppose that $\Gamma_1 \vdash P \mid Q : (\Gamma_3, x : \mathsf{lin}\, p); L_2$. Induction (on Q) tells us that $\Gamma_2 \div L_1 = \Gamma_4, x : \mathsf{lin}\, p$ and $\Gamma_4 \vdash Q : \Gamma_3; L_2$ (1). Since $x \notin L_2$ and $\Gamma_2 \div L_1 = \Gamma_4, x : \mathsf{lin}\, p$ we conclude that $\Gamma_2 = \Gamma_2', x : \mathsf{lin}\, p$, that $\Gamma_4 = \Gamma_2' \div L_1$ (2), and that $x \notin L_1$. Induction (on P this time) tells us that $\Gamma_1 = \Gamma_1', x : \mathsf{lin}\, p$ and that $\Gamma_1' \vdash P : \Gamma_2'; L_1$ (3). The result follows from (1)–(3) and rule [A-Par].

When the derivation ends with rule [A-Out], suppose that $\Gamma_1 \vdash \overline{z}\, v.P : \Gamma_4; L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)$. The case when $x = z$ does not apply. Assume that $\Gamma_4 = \Gamma_4', x : \mathsf{lin}\, p$ and $x \notin L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{ else } \emptyset)$. We know that $x \notin L$ since $x = z$. The induction hypothesis tells us that $\Gamma_3 + x : T = \Gamma_3', z : \mathsf{lin}\, p$ and $\Gamma_3' \vdash P : \Gamma_4, L$. We then know that $z : \mathsf{lin}\, p$ is in Γ_3; let $\Gamma_3 = \Gamma_3'', z : \mathsf{lin}\, p$, hence $\Gamma_3'' + x : U \vdash P : \Gamma_4; L$ (1). Strengthening for values gives $\Gamma_2 = \Gamma_2', z : \mathsf{lin}\, p$ and $\Gamma_2' \vdash v : U; \Gamma_3''$ (2). The same lemma also gives $\Gamma_1 = \Gamma_1, z : \mathsf{lin}\, p$ and $\Gamma_1' \vdash z : T; \Gamma_2'$ (3). The result follows from (1)–(3) and rule [A-Out]. □

We are now in a position to prove the first half of the correctness result. In the result below, dropping the proviso that $\mathcal{U}(\Gamma_2)$ would mean that the type system would not consume all linear variables; something we know not possible. For example, if Γ is the context $x : \mathsf{lin}\, !\mathsf{bool}$, then we have $\Gamma \vdash \mathbf{0} : \Gamma; \emptyset$, but $\Gamma \nvdash \mathbf{0}$.

Lemma 8.4 *(Algorithmic soundness).*

1. *If* $\Gamma_1 \vdash v : T; \Gamma_2$, *then* $\Gamma_3 \vdash v : T$ *and* $\Gamma_1 = \Gamma_2 \circ \Gamma_3$, *for some* Γ_3.
2. *If* $\Gamma_1 \vdash P : \Gamma_2; \_$ *and* $\mathsf{un}(\Gamma_2)$ *then* $\Gamma_1 \vdash \mathrm{erase}(P)$.

Proof. The proof for values follows from a simple analysis of the four axioms involved. The case for processes follows by induction on the structure of derivation of the hypothesis. Cases other than [A-Par] follow by a straightforward induction; we detail [A-Sel]. Suppose that $\Gamma_1 \vdash x \lhd l_j.P : \Gamma_3; L$. By induction we know that $\Gamma_2 + x : T_j \vdash \mathrm{erase}(P)$; soundness for values gives $\Gamma_4 \vdash x : q \oplus \{l_i : T_i\}_{i \in I}$ and $\Gamma_1 = \Gamma_2 \circ \Gamma_4$. We apply rule [T-Sel] to these three results to obtain $\Gamma_1 \vdash x \lhd l_j.\mathrm{erase}(P) = \mathrm{erase}(x \lhd l_j.P)$.

The most interesting case happens when the derivation ends with the [A-Par] rule. We know that $\Gamma_1 \vdash P : \Gamma_2; L_1$ and $\Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2$. Let $\Gamma_2 \div L_1 = \Gamma_2'$; the properties of context difference gives $L_1 \cap \mathrm{dom}(\mathcal{L}(\Gamma_2)) = \emptyset$. Let $\Gamma_1 = \Gamma_1', \mathcal{L}(\Gamma_2)$; applying strengthening to the hypothesis we have $\Gamma_1' \vdash P : \mathcal{U}(\Gamma_2); L_1$. Obviously $\mathsf{un}(\mathcal{U}(\Gamma_2))$; the induction hypothesis yields $\Gamma_1' \vdash \mathrm{erase}(P)$. The same hypothesis also yields $\Gamma_2' \vdash \mathrm{erase}(Q)$. In order to conclude the proof via rule [T-Par] we need to

establish that $\Gamma_1 = \Gamma_1' \circ \Gamma_2'$. Monotonicity yields $\mathcal{U}(\Gamma_2') = \mathcal{U}(\Gamma_1)$; given that $\Gamma_1 = \Gamma_1', \mathcal{L}(\Gamma_2)$ we also know that $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_1')$. As for the linear entries we know, again from $\Gamma_1 = \Gamma_1', \mathcal{L}(\Gamma_2)$, that $\mathcal{L}(\Gamma_1') \cap \mathcal{L}(\Gamma_2) = \emptyset$, and from the properties of context difference that $\mathcal{L}(\Gamma_2) = \mathcal{L}(\Gamma_2')$, hence $\mathcal{L}(\Gamma_1) = \mathcal{L}(\Gamma_1'), \mathcal{L}(\Gamma_2')$. □

The following lemma is used in the proof of completeness, the reverse direction of the correctness theorem.

Lemma 8.5 *(Algorithmic weakening).*

1. *If* $\Gamma_1 \vdash v : T_1; \Gamma_2$ *then* $\Gamma_1, x : T_2 \vdash v : T_1; (\Gamma_2, x : T_2)$.
2. *If* $\Gamma_1 \vdash P : \Gamma_2; L$ *then* $\Gamma_1, x : T \vdash P : (\Gamma_2, x : T); L$ *and* $x \notin L$.

Proof. Once again, the case for values follows from a simple analysis of the axioms involved. The case for processes follows by a straightforward induction on the typing derivation, using the properties of context difference. We detail two cases.

When the derivation ends with rule [A-Res], assume $\Gamma_1 \vdash (\nu xy : T)P : \Gamma_2 \div \{x, y\}; L \setminus \{x, y\}$. Take $z \neq x, y$; by induction we know that $\Gamma_1, x : T, y : \overline{T}, z : U \vdash P : (\Gamma_2, z : U); L$ and $z \notin L$. The properties of context splitting yield $(\Gamma_2, z : U) \div \{x, y\} = \Gamma_1 \div \{x, y\}, z : U$. Conclude the case with rule [A-Res]; that $z \notin L \setminus \{x, y\}$ follows from $z \notin L$ and $z \neq x, y$.

When the derivation ends with rule [A-Par], the induction hypothesis gives $\Gamma_1, x : T \vdash P : (\Gamma_2, x : T); L_1$ with $x \notin L_1$ and $\Gamma_2 \div L_1, x : T \vdash Q : (\Gamma_3, x : T); L_2$ with $x \notin L_2$. The properties of context splitting guarantee that $\Gamma_2 \div L_1, x : T = (\Gamma_2, x : T) \div L_1$, and the result follows by rule [A-Par]. □

We are finally in a position to address the second half of the correctness result and conclude the section.

Lemma 8.6 *(Algorithmic completeness).*

1. *If* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $\Gamma_1 \vdash v : T$ *then* $\Gamma \vdash v : T; \Gamma_2$.
2. *If* $\Gamma_1 \vdash \mathrm{erase}(P)$ *then* $\Gamma_1 \vdash P : \Gamma_2; \_$ *and* $\mathsf{un}(\Gamma_2)$.

Proof. The proof for values follows from a simple analysis of the axioms involved. The case for processes follows by induction on the structure of derivations for the hypothesis. We detail two cases.

When the derivation ends with rule [T-In], we know that $q_1(\Gamma_1 \circ \Gamma_2)$ and $\Gamma_1 \vdash x : q\,?T.U$ and $(\Gamma_2 + x : U), y : T \vdash \mathrm{erase}(P)$. By induction we have $\Gamma_1 \circ \Gamma_2 \vdash x : q\,?T.U; \Gamma_2$ (1) and $(\Gamma_2 + x : U), y : T \vdash P : \Gamma_3; L$ (2) with $\mathsf{un}(\Gamma_3)$. In order to apply [A-In], we still have to show that $q_1 = \mathsf{un} \Rightarrow L = \emptyset$. When q_1 is un, the properties of context splitting tell that $\mathsf{un}(\Gamma_1)$ and $\mathsf{un}(\Gamma_2)$ and $\Gamma_1 = \Gamma_2$. Also monotonicity implies that $\Gamma_2 \setminus L \subseteq \Gamma_1$. Since $\Gamma_1 = \Gamma_2$ we have $\mathrm{dom}(\Gamma_2) \cap L = \emptyset$, but the same properties say that $L \subseteq \mathrm{dom}(\Gamma_1)$, hence $L = \emptyset$ (3). We then apply rule [A-In] to (1)–(3) to obtain the resulting sequent. That $\Gamma_3 \div L$ is unrestricted follows from $\mathsf{un}(\Gamma_3)$.

When the derivation ends with rule [T-Par], by induction we know that $\Gamma_1 \vdash P : \Gamma_3; L_1$ and $\mathsf{un}(\Gamma_3)$. Since $\Gamma_1, \mathcal{L}(\Gamma_2) = \Gamma_1 \circ \Gamma_2$, we weaken the derivation to obtain $\Gamma_1 \circ \Gamma_2 \vdash P : \Gamma_3, \mathcal{L}(\Gamma_2); L_1$ (1). Again by induction we know that $\Gamma_2 \vdash Q : \Gamma_4; L_2$ (2) and $\mathsf{un}(\Gamma_4)$. We now show that $(\Gamma_3, \mathcal{L}(\Gamma_2)) \div L_1 = \Gamma_2$. The properties of context difference tell us that $\mathcal{L}((\Gamma_3, \mathcal{L}(\Gamma_2)) \div L_1) = \mathcal{L}(\Gamma_3, \mathcal{L}(\Gamma_2)) = \mathcal{L}(\Gamma_3), \mathcal{L}(\Gamma_2) = \mathcal{L}(\Gamma_2)$, since $\mathsf{un}(\Gamma_3)$. From the properties of context splitting we know that $\mathcal{U}((\Gamma_3, \mathcal{L}(\Gamma_2)) \div L_1) = \mathcal{U}(\Gamma_1 \circ \Gamma_2) = \mathcal{U}(\Gamma_2)$. We then apply rule [A-Par] to (1)–(2) to obtain the result, $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q : \Gamma_4; L_2$ and $\mathsf{un}(\Gamma_4)$. □

9. Related work and conclusions

Work on session types goes back to Honda, Kubo, Takeuchi, and Vasconcelos, first centering on the type structure [8], then introducing the notion of channel [17], and finally extending the ideas to a more general setting with channel passing [9]. The original work introduces session types, describing chained continuous interactions composed of communication (input and output) and binary choice. The central notion of session types, duality, is also introduced in this work. The subsequent work proposes, at the language level, the concept of *channels* distinct from pi calculus conventional names—channels (linear variables in our terminology) conduct a pattern of interaction between exactly two partners, names (unrestricted variables in this paper) are used by multiple participants to create channels. The language is constructed around a pair of operations, *accept* and *request*, synchronizing on a shared name and establishing a new channel. Channels are endowed with operations to send and receive base values (including names) and to perform choices based on labels, as opposed to the binary choice in [8]. The language in Ref. [9] takes the idea further, allowing channels to be passed on channels—often called *session delegation*—thus including two more operations on channels: to send and to receive a channel.

In Ref. [9], channel passing embodies a technique similar to internal mobility [16] whereby the sender and the receiver must agree on the exact channel being handed over, *prior to communication* itself. Forgoing the variable convention, and using the same variable x to denote the two ends of a channel, the rule for communicating a linear channel y can be written in the conventional pi calculus notation as

$$\overline{x}\,y.P \mid x(y).Q \to P \mid Q$$

where y is both free in $\overline{x}\,y.P$ and bound $x(y).Q$, with the understanding that if the receiving process happens to look like $x(z).Q$ then the bound variable z is renamed to y prior to reduction, *if possible* (that is, if y is not free in Q).

Gay and Hole proposed a variant of [9] by introducing free session passing [4]. Their language is similar to the one in this paper, except for one small detail: it annotates variables with polarities $+$, $-$. Rather than using distinct identifiers x, y that are made co-variables at binding time, $(\nu xy)P$, they use one identifier only, x, that is annotated with polarities (becoming x^+, x^-) and that is bound as a single variable in processes of the form $(\nu x)P$. The new reduction rule for session passing looks as follows.

$$\overline{x^+}\,z.P \mid x^-(y).Q \to P \mid Q[z/y]$$

In either case, our work and that of Gay and Hole, the reason behind the need for syntactically distinguishing the two ends of the same channel comes from free session passing: the same thread may end up possessing the two ends of a channel, as in $\overline{x^+}\,\mathsf{true}.x^-(z)$. After typing $x^-(z)$ we are left with a context where the types for x^+ and x^- are not dual. They will eventually become dual after typing the output process, and should be dual when the derivation reaches scope restriction for x. In other words, duality of channel ends' types is not invariant in typing derivation trees.

Instead, we work with two completely unrelated variables x, y that are made co-variables at binding time only. But there is a fundamental difference between the polarity notation and the co-variable technique used in this paper. In [4], polarity annotated variables are associated to linear channels; unrestricted channels use non-annotated variables. As such, there are two communication rules: for linear channels (on processes of the form $\overline{x^+}\,z.P \mid x^-(y).Q$), and for unrestricted channels (on processes $\overline{x}\,z.P \mid x(y).Q$). We work with co-variables in all cases, using a single communication rule for processes of the form $\overline{x}\,z.P \mid w(y).Q$ where x and w are co-variables. If needed, the distinction between linear and unrestricted channels is made by the type qualifiers associated to variables x and w.

Yoshida and Vasconcelos use the polarity technique to endow the language in [9] with free session passing [23]. All the aforementioned works carefully manage the typing context in order to maintain the invariant where each channel is used exactly in one or two threads, with a technique similar to context splitting.

The technique of binding the two ends of a channel together is due to Gay and Vasconcelos [5], working on a buffered semantics where it makes all the sense to distinguish the two ends of a channel, for each has its own queue for incoming messages.

Recently Giunti and Vasconcelos proposed a type system for the pi calculus with session types that dispenses both polarities and co-variables [6]. Instead, the type system uses pair-types to denote the type of a channel in a thread that possesses the two ends of the same channel; it still uses single types when threads possess one only end. The work has then been extended to a scenario where only pair-types are allowed, meaning that threads will always know the two ends of a channel, one or more possibly at type end [7].

Subtyping for session types was first introduced by Gay and Hole [4], co-inductively given the presence of recursive types. The idea can be straightforwardly incorporated in our language; the previous version of this paper shows how [6]. Given that Gay and Hole present an algorithm for checking the subtyping relation, we still have algorithmic type checking.

A linear type system for the pi calculus was studied by Kobayashi, Pierce and Turner [11]. There, as in the lambda calculus, a linear channel is understood as resource that should be used only once. The exactly-once nature of linear values is at odds with the idea of session types capturing continuous sequences of interactions, and therefore of describing channels naturally occurring more than once in a thread. Instead, a linear channel end in this work is understood as occurring in a single thread, possibly multiple times. The machinery used in this paper, linear and unrestricted type qualifiers and context splitting, is inspired by Walker's substructural type systems [22].

The type language of this paper describes the interaction between two threads (each in possession of a channel end); sessions types to describe interaction among multiple partners are the object of several works in the literature, including, for example, the line of work originating with the work of Honda, Yoshida and Carbone [10], and the Caires and Vieira's conversation calculus [2].

Session types have also been interpreted within intuitionistic linear logic by Caires and Pfenning [1], and later extended in order to obtain a dependent session type system for the pi calculus [18].

Conclusion. We presented a formulation of session types for a pi calculus that syntactically distinguishes the two ends of the same channel, and where types for linear channels are distinguished from those describing shared channels by means of a lin/un qualifier. The formulation allows in particular for a linear channel to evolve into a shared channel. We hope that the simplified theory may lead to further developments, including its incorporation in programming languages (cf. [20]).

Acknowledgments

The author was partially supported by projects Interfaces (CMU–PT/NGN/0044/2008) and Assertion-types (PTDC/EIA–CCO/105359/2008). He thanks Marco Giunti, Francisco Martins, Dimitris Mostrous, and the anonymous referees for advice and suggestions.

References

[1] Luís Caires, Frank Pfenning, Session types as intuitionistic linear propositions, in: Proceedings of CONCUR '10, in: Lecture Notes in Computer Science, vol. 6269, Springer, 2010, pp. 222–236.
[2] Luís Caires, Hugo T. Vieira, Conversation types, Theoretical Computer Science 411 (51–52) (2010) 4399–4440.
[3] Romain Demangeon, Kohei Honda, Full abstraction in a subtyped pi-calculus with linear types, in: Proceedings of CONCUR '11, in: Lecture Notes in Computer Science, vol. 6901, Springer, 2011, pp. 280–296.
[4] Simon J. Gay, Malcolm J. Hole, Subtyping for session types in the pi calculus, Acta Informaticæ 42 (2/3) (2005) 191–225.
[5] Simon Gay, Vasco T. Vasconcelos, Linear type theory for asynchronous session types, Journal of Functional Programming 20 (1) (2010) 19–50.
[6] Marco Giunti, Vasco T. Vasconcelos, A linear account of session types in the pi calculus, in: Proceedings of CONCUR '10, in: Lecture Notes in Computer Science, vol. 6269, Springer, 2010, pp. 432–446.
[7] Marco Giunti, Vasco T. Vasconcelos, Linearity, session types and the pi calculus, 2011, submitted for publication.
[8] Kohei Honda, Types for dyadic interaction, in: Proceedings of CONCUR '93, in: Lecture Notes in Computer Science, vol. 715, Springer, 1993, pp. 509–523.
[9] Kohei Honda, Vasco T. Vasconcelos, Makoto Kubo, Language primitives and type disciplines for structured communication-based programming, in: Proceedings of ESOP '98, in: Lecture Notes in Computer Science, vol. 1381, Springer, 1998, pp. 22–138.
[10] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: Proceedings of POPL '08, vol. 43, ACM Press, 2008, pp. 273–284.
[11] Naoki Kobayashi, Benjamin Pierce, David Turner, Linearity and the pi-calculus, ACM Transactions on Programming Languages and Systems 21 (5) (1999) 914–947.
[12] Robin Milner, Functions as processes, Mathematical Structures in Computer Science 2 (2) (1992) 119–141.
[13] Robin Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, May 1999.
[14] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002.
[15] Benjamin C. Pierce, David N. Turner, Pict: A programming language based on the pi-calculus, in: Proof, Language and Interaction: Essays in Honour of Robin Milner, in: Foundations of Computing, MIT Press, May 2000.
[16] Davide Sangiorgi, π-Calculus, internal mobility and agent-passing calculi, Theoretical Computer Science 167 (1–2) (1996) 235–274.
[17] Kaku Takeuchi, Kohei Honda, Makoto Kubo, An interaction-based language and its typing system, in: Proceedings of PARLE '94, in: Lecture Notes in Computer Science, vol. 817, Springer, 1994, pp. 398–413.
[18] Bernardo Toninho, Luís Caires, Frank Pfenning, Dependent session types via intuitionistic linear type theory, in: Proceedings of PPDP '11, ACM, 2011, pp. 161–172.
[19] Vasco T. Vasconcelos, Fundamentals of session types, in: 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems, in: Lecture Notes in Computer Science, vol. 5569, Springer, 2009, pp. 158–186.
[20] Vasco T. Vasconcelos, Sessions, from types to programming languages, Bulletin of the European Association for Theoretical Computer Science 103 (2011) 53–73.
[21] Vasco T. Vasconcelos, Mario Tokoro, A typing system for a calculus of objects, in: Proceedings of ISOTAS '93, in: Lecture Notes in Computer Science, vol. 472, Springer, November 1993, pp. 460–474.
[22] David Walker, Substructural type systems, in: Advanced Topics in Types and Programming Languages, MIT Press, 2005.
[23] Nobuko Yoshida, Vasco T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication, in: Proceedings of SecReT '07, in: Electronic Notes in Theoretical Computer Science, vol. 171(4), Elsevier, 2007, pp. 73–93.