Fundamentals of Session Types*

Vasco T. Vasconcelos

University of Lisbon

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. In our language each channel is characterised 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 a same channel. This mechanism allows a precise control of resources via a linear type system. We build the language gradually, starting from simple input/output, then adding choice, recursive types, replication and finally subtyping. 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 protocol. The nature and order of this messages are a natural candidate for structuring interactions themselves. It is in this context that session types make their contribute by allowing a concise description of the continuous interactions among partners in a concurrent computation.

For example, consider a simplified distributed auction system with three kinds of players: sellers that want to sell items, auctioneers that sell items on their behalf, and bidders that bid for an item being auctioned. The protocol for sellers is simple: there is only one operation that sellers may invoke on an auctioneer—selling—where they provide the auctioneer with a description of the item to be sold (a string), and the minimum price they are willing to sell the item for. The protocol starts as follows, where $\oplus$ introduces the choices available to the seller, and ! the output of a value.

$$\oplus\{\mathit{selling}\colon !\mathrm{String}.!\mathrm{Price}\ldots\}$$

Sellers then wait on the outcome of their request. Two things can happen: either the item was sold (in which case the seller gets the price the item was sold for), or the item was not sold. The protocol then continues as below, where & denotes

* M. Bernardo, L. Padovani, and G. Zavatarro (Eds.): SFM 2009, LNCS 5569, pp. 158–186, 20099, DOI 10.1007/978–3–642–01918–0_4. Some typos fixed, replication simplified on Jun 6, 2009. Algorithmic type checking fixed on Oct 16, 2010.

the range of alternatives offered by the seller at this point, and ? represents input.

$$\&\{\mathit{sold}\colon ?\mathrm{Price}\ldots, \mathit{notSold}\colon \ldots\}$$

In either case the protocols halts; we indicate that with the end mark. The complete protocol as seen by the seller can be concisely described.

$$\oplus\{\mathit{selling}\colon !\mathrm{String}.!\mathrm{Price}.\&\{\mathit{sold}\colon ?\mathrm{Price}.\mathsf{end}, \mathit{notSold}\colon \mathsf{end}\}\}$$

The protocol for auctioneers is slightly more complex, for they must interact not only with sellers but with bidders as well. Starting with the interaction with sellers, we know that auctioneers must offer a *selling* alternative, and if such alternative is taken, then they must accept a string (the item be sold) followed by the price the seller is asking.

$$\&\{\mathit{selling}\colon ?\mathrm{String}.?\mathrm{Price}\ldots\}$$

The auctioneer then puts the item on sale, and gets back to the seller with one of the possible outcomes: *sold* or *notSold*.

$$\oplus\{\mathit{sold}\colon !\mathrm{Price}\ldots, \mathit{notSold}\colon \ldots\}$$

Putting everything together we have two session types, the first for the seller, the second for the auctioneer.

$$\oplus\{\mathit{selling}\colon !\mathrm{String}.!\mathrm{Price}.\&\{\mathit{sold}\colon ?\mathrm{Price}.\mathsf{end}, \mathit{notSold}\colon \mathsf{end}\}\}$$
$$\&\{\mathit{selling}\colon ?\mathrm{String}.?\mathrm{Price}.\oplus\{\mathit{sold}\colon !\mathrm{Price}.\mathsf{end}, \mathit{notSold}\colon \mathsf{end}\}\}$$

The description implies that sellers should be able to safely interact with auctioneers; the session types for the two partners make this clear: when the seller selects the *selling* choice, the auctioneer offers that exact choice, and conversely for choices *sold* and *notSold*. Furthermore, when the seller outputs a value, the auctioneer inputs a value of the same type, and when the seller ends the protocol, so does the auctioneer. We say that the two types are *dual*, a notion central to session types.

But the auctioneer should also interact with bidders. Bidders start by registering themselves, then enter an interactive bidding session, and eventually unregister, thus leaving the protocol. The auctioneer offers a second option—register—to be used by bidders.

$$\&\{\mathit{selling}\ldots, \mathit{register}\colon \ldots\}$$

Bidders on the other hand must follow a protocol of the form $\oplus\{\mathit{register}\colon \ldots\}$, dual to that of the corresponding branch in the auctioneer. In summary we have the following situation

$$\begin{aligned}\text{auctioneer}&\colon \&\{\mathit{selling}\ldots, \mathit{register}\colon \ldots\}\\ \text{seller}&\colon \oplus\{\mathit{selling}\colon \ldots\}\\ \text{bidder}&\colon \oplus\{\mathit{register}\colon \ldots\}\end{aligned}$$

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

Fig. 1. The syntax of processes

but now the protocol of the auctioneer is not dual to neither that of seller nor that of the bidder. *Subtyping* allows to specialize the type of the auctioneer to that of the seller, as in $\&\{\mathit{selling}\ldots\}$, or to that of the bidder, $\&\{\mathit{register}\ldots\}$, as required by duality.

This chapter introduces a reconstruction of session types based on the ideas of linear type systems. Session types describe communication channels in the pi calculus, both linear and shared (or unrestricted). The various concepts usually associated to session types are introduced piecewise. We start by studying a language with input, output, parallel composition, and scope restriction. We then incorporate choice in the form of branching (external choice) and selection (internal choice). Even though the required machinery is in place, the particular form of types does not allow to type useful unrestricted channels—recursive types provide such a facility. Up to this point the language does not allow describing unbounded computations—we introduce replication for the effect. The next step is to introduce subtyping, thus enlarging the class of typable programs. The last step in the development of our language introduces an algorithmic type checking system. The closing section includes references to the sources of this chapter and discusses related work.

2 Syntax

Figure 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 word channel to denote 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—one party writes on one end, others read from the other end. Interacting threads do not share variables for communication; since

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

Fig. 2. The syntax of types

a channel is represented as a pair of co-variables, each thread owns its variable. This mechanism allows a precise control of resources via a 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 $\overline{x}\,v.P$ writes value v on variable x and continues as P. Conversely, the input process $y(z).P$ receives on variable y a value it uses to substitute 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 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 hides 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. 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.

3 Typing

The syntax of types is described in Figure 2. Type qualifiers annotate pretypes. For pretypes we have bool, the type of the boolean values. Pretype end may be used to represent a co-variable on which no further interaction is possible. Pretypes $!T.U$ and $?T.U$ describe channels ready to send or to receive a value of type T and then continuing its interaction as prescribed by type U.

Linearly qualified types describe variables that occur in exactly *one thread*, a thread being any process not comprising parallel composition. The unrestricted

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

Fig. 3. The dual function on types

qualifier indicates that the value can occur in multiple threads. A type $\mathsf{lin}\,\mathsf{bool}$ represents a boolean value that can be tested exactly once, whereas $\mathsf{un}\,\mathsf{bool}$ describes a boolean value that can be tested a variable number of times. Similarly a type $\mathsf{lin}\,!T.U$ represents a channel that can be used once for sending a value of type T before becoming a channel that behaves as U. A channel $\mathsf{un}\,!T.U$ can be used multiple types to send values of type T. Typing contexts, also introduced in Figure 2, gather type information on variables.

To lighten the syntax in examples, we adopt a few abbreviations. First, we omit all unrestricted qualifiers and only annotate linear types. Second we omit the trailing **0** in processes. Third, we omit the trailing $\mathsf{un}\,\mathsf{end}$ in types. In examples involving communication we also assume that co-variables are annotated with subscripts 1 and 2, for example (x_1, x_2) and (y_1, y_2).

If x is a variable of an arbitrarily qualified type, a is a variable of an unrestricted type and c a variable of a linear type, then the first two processes are well formed, whereas the last one is not.

$$\overline{x}\,\mathsf{true}.x(y) \qquad :\text{-)}$$
$$\overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{false} \qquad :\text{-)}$$
$$\overline{c}\,\mathsf{true} \mid \overline{c}\,\mathsf{false} \qquad :\text{-(}$$

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 $q!S.T$. Pretype end is dual of itself; duality is not defined for the bool type. The definition is in Figure 3.

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

$$\overline{x_1}\,\mathsf{true} \mid x_2(z) \qquad :\text{-)}$$
$$\overline{x_1}\,\mathsf{true}.x_1(w) \mid x_2(z).\overline{x_2}\,\mathsf{false} \qquad :\text{-)}$$
$$\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{false} \qquad :\text{-(}$$
$$\overline{x_1}\,\mathsf{true}.x_1(w) \mid x_2(z).x_2(t) \qquad :\text{-(}$$

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. Consider the process:

$$\overline{x_1}\,y_2 \mid x_2(z).\overline{z}\,\mathsf{true} \mid \overline{y_1}\,\mathsf{false} \qquad :\text{-(}$$

Context split

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma}{\Gamma, x\colon \mathsf{un}\, p = (\Gamma_1, x\colon \mathsf{un}\, p) \circ (\Gamma_2, x\colon \mathsf{un}\, p)}$$

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

Context update

$$\Gamma = \Gamma + \emptyset \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x\colon T = \Gamma_1 + (\Gamma_2, x\colon T)} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x\colon \mathsf{un}\, p = (\Gamma_1, x\colon \mathsf{un}\, p) + (\Gamma_2, x\colon \mathsf{un}\, p)}$$

Fig. 4. Context split and context update

The following context is expected to type the process, where the argument $y_2\colon !\mathsf{bool}$ of the send operation on x_1 is dual of parameter $z\colon ?\mathsf{bool}$ in the receive operation on x_2.

$$x_1\colon !(!\mathsf{bool}), x_2\colon ?(?\mathsf{bool}), y_1\colon !\mathsf{bool}, y_2\colon !\mathsf{bool}$$

Yet the process reduces to an illegal process, where y_1 and y_2 are not dual.

$$\overline{y_2}\,\mathsf{true} \mid \overline{y_1}\,\mathsf{false} \qquad :\text{-(}$$

For each qualifier q we define a predicate also named q which is true of types qp and also of contexts $x_1\colon qp_1, \ldots, x_n\colon qp_n$. We maintain the linearity invariant through the standard linear context splitting 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 two and passing a different part to each process. In this way, if x is a linear variable then the process $\overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{true}$ is not typable, since x can only occur in one of the parts, allowing to type one but not both processes. Figure 4 defines the context splitting relation $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that in the third rule, x is not in Γ_2 since it is not in $\Gamma = \Gamma_1 \circ \Gamma_2$, and similarly for the last rule and Γ_1.

Equipped with the notions of context splitting and type duality we are ready to introduce the typing rules. We distinguish typing rules for values with judgments of the form $\Gamma \vdash v\colon T$, from those for processes with judgments $\Gamma \vdash P$. The rules are in Figure 5.

Our type system maintains the following invariants.

- Linear channels occur in exactly one thread;
- Co-variables have dual types.

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

Typing rules for values

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{true}\colon q\,\mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{false}\colon q\,\mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x\colon T, \Gamma_2 \vdash x\colon T}$$

[T-False] [T-True] [T-Var]

Typing rules for processes

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

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

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

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

Fig. 5. Typing rules

rules for values are those one finds in the linear lambda calculus—boolean values have type bool, variables have the type prescribed by the context. Rule [T-Var] allows variable x to occur anywhere in the context, as opposed to just at the beginning or at the end.

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. Rule [T-If] for the conditional process splits the incoming context in two parts: one used to check the condition, the other to check both branches. The same context for the two branches is justified by the fact that only one of P or Q will be executed. The qualifier of the boolean value is unimportant.

For rule [T-Res] we add to the context two extra hypotheses for the newly introduced variables, at dual types. The rule captures the essence of co-variables: they must have dual types.

Similarly to the rule for parallel composition, 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 $y\colon T$. 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 updating is only defined when U is equal to $q?T.U$, which will become possible in Section 6.

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

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

Fig. 6. Structural congruence

type, that is, $\overline{x}\, v.P$ uses x at type $q!T.U$, whereas P uses the same variable at type U.

The dual function is not total: it is not defined on bool, nor on any type "terminating" in bool, such as $?\mathsf{bool}.\mathsf{bool}$. 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 process

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

or any process reducing to it.

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 $x\colon !\mathsf{bool}.T$. Then the third premise of rule [T-Out] reads $(x\colon !\mathsf{bool}.T) + (x\colon T)$ which cannot be fulfilled by any type T built from the syntax in Figure 2. Clearly, so far, we are dealing with a language of linear channels only.

The following structural property of the type system is useful in the proof of preservation (Theorem 1).

Lemma 1 (Unrestricted weakening). *If $\Gamma \vdash P$ then $\Gamma, x\colon \mathsf{un}\, p \vdash P$.*

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 two applicable typing rules. The hypothesis $\mathsf{un}(\Gamma)$ in rule [T-Inact] establishes the base case. □

4 Operational Semantics

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 $P[v/x]$. 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.

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

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

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

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

$$\frac{P \;\to\; Q}{P \mid R \;\to\; Q \mid R} \qquad \text{[R-Par]}$$

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

Fig. 7. Operational semantics

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, $\equiv$, is the smallest congruence relation on processes that satisfies the axioms in Figure 6. The axioms are standard in pi calculus. The first three say that parallel composition is commutative, associative and contains the terminated process $\mathbf{0}$ for neutral. The first rule 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 rules allow to collect unused restrictions and to exchange the order of bindings.

The operational semantics is defined in Figure 7. 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-channels, 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 further interactions in the resulting process. Process R witnesses reduction on unrestricted channels; it may represent the terminated process $\mathbf{0}$ on reduction on linear channels. 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.

Unlike the linear lambda calculus, our type system offers no guarantee of progress. If fact processes can deadlock quite easily, it suffices to create two

sessions that read and write in the "wrong" order.

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

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.

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

The rest of this section is dedicated to the proof of the main results of our language.

Equipped with the notion of free variables and substitution we can prove two important results of our type system. Strengthening allows to remove extraneous entries from the context, but only when the variable does not occur free in the process. Clearly we have $x\colon ?\mathsf{bool} \vdash x(y)$, but not $\vdash x(y)$. Also, linear variables occur in the context only if free in the process, e.g., $x\colon \mathsf{lin}?\mathsf{bool} \vdash \mathbf{0}$ is not a valid judgement.

Lemma 2 (Strengthening). *If* $\Gamma, x\colon T \vdash P$ *and* $x \notin \mathrm{fv}(P)$ *then* $\Gamma \vdash P$ *and* $\mathsf{un}(T)$.

Proof. The proof is by induction on the structure of the derivation. The hypothesis $\mathsf{un}(\Gamma)$ in rule [T-Inact] establishes the base case. □

The following result relates judgments $\Gamma \vdash P$ and the free variables of P.

Lemma 3 (Free variables).

- *If* $\Gamma \vdash P$ *and* $x \in \mathrm{fv}(P)$ *then* $x \in \Gamma$.
- *If* $\Gamma, x\colon \mathsf{lin}\,p \vdash P$ *then* $x \in \mathrm{fv}(P)$.

Proof. The proofs are by induction on the derivations. □

The Substitution Lemma plays a central role in proof of type preservation (Theorem 1).

Lemma 4 (Substitution). *If* $\Gamma_1 \vdash v\colon T$ *and* $\Gamma_2, x\colon T \vdash P$ *and* $\Gamma_1 \circ \Gamma_2$ *is defined then* $\Gamma_1 \circ \Gamma_2 \vdash P[v/x]$.

Proof. The proof is by induction on the typing derivation and uses Strengthening and Weakening and Free variables (Lemmas 1, 2, and 3). This is the most elaborate proof in this section. We start with the simple observation that if $\Gamma_1 \vdash v\colon T$ then *either* $v = \mathsf{true}$ and $\mathsf{un}(\Gamma_1)$ *or* v is a variable and $\Gamma_1 = \Gamma_3, v\colon T, \Gamma_4$ and $\mathsf{un}(\Gamma_3, \Gamma_4)$. For the base case (rule [T-Inact]), we know that $\mathsf{un}(\Gamma_2)$ and $\mathsf{un}(T)$. The result follows by Strengthening and Weakening. For each inductive case we prove two situations separately: $\mathsf{lin}(T)$ and $\mathsf{un}(T)$. □

The next lemma states that structural equivalent processes can be typed under the same contexts, and is used in the [R-Struct] case of the proof of preservation.

Lemma 5 (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, Strengthening, and Free variables (Lemmas 1, 2, and 3), and must not forget to check the two directions of each axiom.

A representative 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 a derivation for $\Gamma \vdash (\nu xy)P \mid Q$, to conclude that Γ must be of the form $\Gamma_1 \circ \Gamma_2$, that $\Gamma_1, x\colon T, y\colon \overline{T} \vdash P$, and that $\Gamma_2 \vdash Q$. To build a derivation for the conclusion we start with $\Gamma_2 \vdash Q$ and distinguish two cases. If T is linear, then $(\Gamma_1, x\colon T, y\colon \overline{T}) \circ \Gamma_2 = \Gamma_1 \circ \Gamma_2, x\colon T, y\colon \overline{T}$; otherwise use Weakening to conclude that $\Gamma_2, x\colon T, y\colon \overline{T} \vdash Q$ and $(\Gamma_1, x\colon T, y\colon \overline{T}) \circ (\Gamma_2, x\colon T, y\colon \overline{T}) = \Gamma_1 \circ \Gamma_2, x\colon T, y\colon \overline{T}$. 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 consider two cases, depending on whether rule [T-Res] introduces an unrestricted or a linear type. For the former, applying rules [T-Res] and [T-Par] from the conclusion $\Gamma \vdash (\nu xy)(P \mid Q)$, we know that $\Gamma = \Gamma_1 \circ \Gamma_2$, that $\Gamma_1, x\colon T, y\colon \overline{T} \vdash P$ and that $\Gamma_2, x\colon T, y\colon \overline{T} \vdash Q$. To build a derivation for the conclusion, we apply Strengthening to the hypothesis on Q to obtain $\Gamma_2 \vdash Q$, and then apply [T-Res] and [T-Par] as required.

If on the other hand T is linear, by Free variables there is one only way to split $\Gamma_1 \circ \Gamma_2, x\colon T, y\colon \overline{T}$; we have $\Gamma_1, x\colon T, y\colon \overline{T} \vdash P$ and $\Gamma_2 \vdash Q$ and we conclude the proof using rules [T-Res] and [T-Par]. □

Theorem 1 (Preservation). *If $\Gamma \vdash P$ and $P \to Q$ then $\Gamma \vdash Q$.*

Proof. The proof is by induction on the reduction derivation, and uses Weakening and Substitution (Lemmas 1 and 4). The inductive cases are straightforward; we use Lemma 5 in case [R-Struct].

The most interesting case is when the derivation of the reduction step ends with rule [R-Com]. Suppose that [T-Res] introduces $x\colon q!T.U, y\colon q?T.\overline{U}$. Building the 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 nature of qualifier q. If linear then we have $\Gamma_1, x\colon U \vdash P$ and $\Gamma_2, z\colon T, x\colon \overline{U} \vdash Q$ and $\Gamma_4 \vdash v\colon T$. From $\Gamma_4 \vdash v\colon T$ and $\Gamma_2, z\colon T, x\colon \overline{U} \vdash Q$ we use Substitution to obtain $\Gamma_4 \circ \Gamma_2, x\colon \overline{U} \vdash Q[v/z]$. We then conclude the proof with rules [T-Par], [T-Par], [T-Res].

If q is unrestricted, we have $(\Gamma_1, x\colon q!T.U) \circ x\colon U \vdash P$, and $(\Gamma_2, y\colon q?T.\overline{U}, z\colon T) \circ y\colon \overline{U} \vdash Q$ and $\Gamma_4, x\colon q!T.U \vdash v\colon T$. The first context splitting 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 $\Gamma_1, x\colon U \vdash P$ to $\Gamma_1, x\colon U, y\colon \overline{U} \vdash P$, from $\Gamma_2, z\colon T, y\colon \overline{U} \vdash Q$ to $\Gamma_2, z\colon T, x\colon U, y\colon \overline{U} \vdash Q$, from $\Gamma_3 \vdash R$ to $\Gamma_3, x\colon U, y\colon \overline{U} \vdash R$, and from $\Gamma_4, x\colon U \vdash v\colon T$ to $\Gamma_4, x\colon U, y\colon \overline{U} \vdash v\colon T$. Using Substitution, we conclude the proof as in the case of q linear. □

We now look at the guarantees offered by typable processes. To study what can go wrong with our machine, we look at the syntax of processes (Figure 1) and the reduction relation (Figure 7), and try to figure out in which cases can the machine get stuck, that is, not able to proceed because of ill-formed processes. There is an obvious case: the value in the condition is neither true nor false in rules [R-IfT] and [R-IfF]. But there are other processes that may prevent the machine from advancing. These include processes with two threads sharing a variable, but using it with distinct interaction patterns, and two threads each possessing a co-variable, but using them in non-dual patterns.

$$\overline{a}\,\mathsf{true} \mid a(z) \qquad :\text{-(}$$
$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true}) \qquad :\text{-(}$$
$$(\nu x_1 x_2)(x_1(z) \mid x_2(w)) \qquad :\text{-(}$$

We say that a process is *non well-formed* if it can be written as $(\nu \tilde{x}\tilde{y})(P \mid Q \mid R)$ up to structural congruence, and one of the following happens.

1. P is of the form $\mathsf{if}\ v\ \mathsf{then}\ P'\ \mathsf{else}\ P''$ and $v \neq \mathsf{true}, \mathsf{false}$; or
2. P is of the from $\overline{x}\,v.P'$ and Q is $x(z).Q'$; or
3. P is of the from $\overline{x_i}\,u.P'$ and Q is $\overline{y_i}\,v.Q'$, or P is of the form $x_i(z).P'$ and Q is $y_i(z).Q'$.

Typable processes are not necessarily well-formed. The process $\mathsf{if}\ x\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$ is typable under context $x\colon \mathsf{bool}$, 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 2) and x is bound by a (νxy) binder, then rule [T-Res] introduces two dual types in the context, $x\colon T, y\colon \overline{T}$, where T is necessarily different from bool, for duality would not be defined otherwise.

Theorem 2. *If $\vdash P$ then P is well formed.*

Proof. The proof is by contradiction. We build the derivation for $\vdash (\nu \tilde{x}\tilde{y})(P_1 \mid P_2 \mid P_3)$; in each of the three cases a simple analysis of the hypothesis shows that P is not typable. □

5 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 Figure 8, 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 $x \rhd \{l_i\colon P_i\}_{i\in I}$ offers a range of options, each labelled with a different label in the set $\{l_i\}_{i\in I}$. Such a process handles a selection at label l_j by executing process P_j.

New syntactic forms

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

New duality rules

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

New typing rules

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

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

New reduction rules

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

Fig. 8. Choice

Imagine a data structure mapping elements from a given type Key to a type Value. Among its various operations one finds *put* and *get*. To *put* key k associated to a value v one writes:

$$
\mathit{map} \lhd \mathit{put}.\, \overline{\mathit{map}}\, k.\, \overline{\mathit{map}}\, v
$$

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 we should be notified of the fact. We use labels *some* and *none* to denote the result of the *get* operation. Further, if 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.

$$
\mathit{map} \lhd \mathit{get}.\, \overline{\mathit{map}}\, k.\, \mathit{map} \rhd \{\mathit{some} \colon \mathit{map}(x).P, \mathit{none} \colon Q\}
$$

Types for the new constructors are $\oplus\{l_i \colon T_i\}_{i \in I}$ and $\&\{l_i \colon T_i\}_{i \in I}$, 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. Select types are akin to labelled variants in sequential languages, whereas branching types can be compared to labelled records. The new type structures are interpreted as non-ordered records; we do not distinguish $\&\{l \colon T, m \colon U\}$ from $\&\{m \colon U, l \colon T\}$.

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

$$\oplus\{put\colon !\mathrm{Key}.!\mathrm{Value}, get\colon !\mathrm{Key}.\&\{some\colon ?\mathrm{Value}.\mathsf{end}, none\colon \mathsf{end}\}\}$$

The two new pretypes are dual to each other. In the third example below x is obviously unrestricted.

$$\begin{aligned}
x_1 \lhd l \mid x_2 \rhd \{l\colon \mathbf{0}\} &\quad :\text{-)}\\
x_1 \lhd l \mid x_2 \rhd \{l\colon \mathbf{0}, m\colon \mathbf{0}\} &\quad :\text{-)}\\
x_1 \lhd l \mid x_1 \lhd m \mid x_1 \lhd m \mid x_2 \rhd \{l\colon \mathbf{0}, m\colon \mathbf{0}\} &\quad :\text{-)}\\
\overline{x_1}\,\mathsf{true} \mid x_2 \rhd \{l\colon \mathbf{0}\} &\quad :\text{-(}\\
x_1 \lhd l \mid x_2(z) &\quad :\text{-(}\\
x_1 \lhd l \mid x_2 \rhd \{m\colon \mathbf{0}\} &\quad :\text{-(}
\end{aligned}$$

To type check a branching process prefixed by x at type $\&\{l_i\colon T_i\}_{i\in I}$ we have to check each of the possible continuations P_i at $x\colon T_i$. 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. If rule [T-Branch] introduces an external choice type $\&\{l_i\colon T_i\}_{i\in I}$, rule [T-Sel] eliminates the dual, internal choice type $\oplus\{l_i\colon T_i\}_{i\in I}$. To type check a process selecting label l_j at name x at type $\oplus\{l_i\colon T_i\}_{i\in I}$, we have to type check the continuation process at the correspondent type $x\colon T_j$. In both cases, and similarly to the rules for output and input in Figure 5, context splitting $\Gamma \circ x\colon T$ must be defined.

The operational semantics is extended with rule [R-Case]. The rule follows the pattern of [R-Com]: 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.

Exercise 1. Sketch a proof of type preservation for the new language.

Exercise 2. What are the new errors associated with the constructs for branching and selection? Redefine the notion of *well formed* processes. Sketch a proof for the type safety result.

6 Recursive Types

The typing rule for the output process (rule [T-Out] in Figure 5) does not allow to type check a process $\overline{x}\,v.P$ with x unrestricted, for it requires the continuation T of type $\mathsf{un}!T.U$ to be equal to $\mathsf{un}!T.U$ itself. We would like to consider as a type the regular infinite tree solution to the equation $U = \mathsf{un}!T.U$. A finite notation for such a type uses the μ-notation, as in $\mu a.\mathsf{un}!T.a$.

Figure 9 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

New syntactic forms

$$\begin{array}{rll} T & ::= & \ldots & \text{Types:} \\ & & a & \text{type variable} \\ & & \mu a.T & \text{recursive type} \end{array}$$

New duality rules

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

Fig. 9. Recursive types

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. Rather than defining type equivalence directly, we rely on the definition of subtyping discussed in Section 8. In any case, types are understood up to type equivalence, so that, for example, in any mathematical context, types $\mu a.T$ and $T[\mu a.T/a]$ can be used interchangeably, effectively adopting the equi-recursive approach.

The dual function 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. 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 underneath 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. Since not all type constructors are qualified anymore, we have to adjust the un and the lin predicates on types. Predicate q is true of types qp as before; and is now true of type $\mu a.T$ if it is true of type T.

Unlike the linear lambda calculus where unrestricted data structures may not contain linear data structures, unrestricted channels can carry both unrestricted and linear channels. Consider the type $?(\mathsf{lin}!\mathsf{bool}).T$ of an unrestricted channel that receives a linear channel capable of outputting a boolean value. The following sequent is easy to establish,

$$x_2 \colon ?(\mathsf{lin}!\mathsf{bool}).T \vdash x_2(z).\overline{z}\,\mathsf{true} \mid x_2(w).\overline{w}\,\mathsf{false} \qquad :\text{-)}$$

but only for an appropriate type T. We have seen that it must be equivalent to $?(\mathsf{lin}!\mathsf{bool}).T$, that is T must be $\mu a.?(\mathsf{lin}!\mathsf{bool}).a$. This form of types is so common that we introduce a short form for them, simply writing $*?(\mathsf{lin}!\mathsf{bool})$.

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.

$$\overline{x}\,\langle u, v\rangle.P = \overline{x}\,u.\overline{x}\,v.P$$

If the tuple is to be passed on a unrestricted channel, then we must protect the receiving operations from interference, creating a new $\mathsf{lin}?\mathrm{T}.\ \mathsf{lin}?\mathrm{U}$ channel to carry the values. The standard encoding for the binary sending and receiving operations are as follows.

$$\begin{aligned}\overline{x_1}\,\langle u, v\rangle.P &= (\nu y_1 y_2)\overline{x_1}\,y_2.\overline{y_1}\,u.\overline{y_1}\,v.P\\ x_2(w, t).P &= x_2(z).z(w).z(t).P\end{aligned}$$

The encodings are typable in our language, if we choose variable y_1 of appropriate linear type, $\mathsf{lin}!T.\mathsf{lin}!U$, and dually for y_2. Variable x_1 is then of type $*!(\mathsf{lin}?T.\mathsf{lin}?U)$, 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. Below is a process that writes two boolean values on a given channel z and then returns the channel (on a given channel w) so that it can be further used.

$$p_1(z, w).\overline{z}\,\mathsf{true}.\overline{z}\,\mathsf{true}.\overline{w}\,z \qquad :\text{-})$$

A process that calls p_1 to read two boolean values and then writes a third on channel x can be written as

$$\overline{p_2}\,\langle c, x_1\rangle \mid x_2(z).\overline{z}\,\mathsf{false} \qquad :\text{-})$$

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

A once linear channel can become unrestricted, we just have to get the right types. For example, type $T = \mathsf{lin}!\mathsf{bool}.*?\mathsf{bool}$ 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 $\overline{T}$.

$$\overline{x_1}\,\mathsf{true}.(x_1(y) \mid x_1(z)) \mid x_2(x).(\overline{x_2}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{false} \mid \overline{x_2}\,\mathsf{true}) \qquad :\text{-})$$

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

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

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 $*!\mathsf{bool} = \mu b.?\mathsf{bool}.b$. What about a channel that can we used both for reading and for writing? There is no such thing in this theory; the channel is in reality a pair of co-variables, one to read, the other to write.

Equipped with the equi-recursive notion of types, typing rules (in Figure 5) remain unchanged. More importantly, the preservation theorem holds as before, and we do not even need to touch the proof.

New syntactic forms

$$\begin{array}{rll} P & ::= & \ldots & \text{Processes:} \\ & & *\,x(x).P & \text{replication} \end{array}$$

New typing rules

$$\frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash *P} \qquad \text{[T-Repl]}$$

New reduction rules

$$(\nu xy)(\overline{x}\,v.P \mid *y(z).Q \mid R) \;\rightarrow\; (\nu xy)(P \mid Q[v/z] \mid *y(z).Q \mid R) \qquad \text{[R-Repl]}$$

Fig. 10. Replication

7 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 Figure 10.

The reduction rule [R-Repl] for a replicated process $*x(y).P$ is similar to that of a simple receptor (rule [R-Com] in Figure 7) in all respects except that process $*x(y).P$ persists in the resulting process.

As an example, 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. A client that reads and discards every value from the iterator can be written as follows.

$$*\,loop(y).y \lhd hasNext.y \rhd \{yes\colon y \lhd next.y(z).\overline{loop}\,y,\, no\colon \mathbf{0}\} \mid \overline{loop}\,x_2 \qquad (1)$$

Clearly, the communication pattern of the iterator, as seen by the client at variable x_2, is of the form

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

which can be written in finite form as follows.

$$\mu a.\mathsf{lin} \oplus \{hasNext\colon \mathsf{lin}\,\&\{no\colon \mathsf{end}, yes\colon \mathsf{lin} \oplus \{next\colon \mathsf{lin}\,!\mathsf{bool}.a\}\}\} \qquad (2)$$

Notice that the type in equation 2 is equivalent to the following,

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

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.

New typing rule

$$\frac{\Gamma \vdash v : T \qquad T <: U}{\Gamma \vdash v : U} \qquad \text{[T-Sub]}$$

Fig. 11. Subtyping

The typing rule [T-Repl] for the replicated process $*x(y).P$ directly calls rule [T-In] in Figure 5 for the input process $x(y).P$. In addition, it requires the context that types the body $x(y).P$ of the replicated process to be unrestricted. To understand what would happen if we relax this restriction, consider the following process

$$*x_2(z).\overline{c}\,\mathsf{true} \mid \overline{x_1}\,\mathsf{true} \mid \overline{x_1}\,\mathsf{false} \qquad \text{:-(}$$

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

$$*x_2(z).\overline{z}\,\mathsf{true} \qquad \text{:-)}$$

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.

Exercise 3. Prove that type preservation still holds for the language with replication.

Exercise 4. In this section we made the input process persistent by using replication. Branching, introduced in Section 5, can be made persistent as well. Devise a typing and a reduction rule for a replicated branching process $*x \triangleright \{l_i : P_i\}_{i \in I}$. Sketch the proof of the corresponding case in the type preservation theorem.

8 Subtyping

Subtyping brings extra flexibility to our type system. The insistence that arguments in output processes exactly match input parameters in corresponding receivers leads to the rejection of programs that will never go wrong when executed.

One example can be found in the introduction. For another, the iterator discussed in Section 6 imposes a strict discipline on its clients: they must alternate between operations *hasNext* and *next*, as long as *hasNext* returns *yes*. A more liberal server would allow clients to call, after the first *hasNext*, not only *next*

but also and again *hasNext*.

$$\mathsf{lin}\,\&\{\mathit{hasNext}\colon \mu b.\mathsf{lin}\oplus\{\mathit{no}\colon \mathsf{end}, \mathit{yes}\colon \mathsf{lin}\,\&\{\mathit{next}\colon \mathsf{lin}\,!\mathsf{bool}.\ \mathsf{lin}\,\&\{\mathit{hasNext}\colon b\}, \mathit{hasNext}\colon b\}\}\} \quad (3)$$

Now imagine the situation where we have typed both the iterator (prefixed at x_1) and its client (at x_2) in context $x_2\colon T, x_1\colon \overline{T}$, where T is the type in equation 2, and we now replace the iterator to conform to the type U in equation 3. Operationally there should be no problem. Below are two snapshots of the system where the client is about to ask *next*, first to the old iterator, and then to the new.

$$\begin{aligned}&(\nu x_2x_1)(x_2 \lhd \mathit{next}.x_2(y).\overline{\mathit{loop}}\,x_2 \mid x_1 \rhd \{\mathit{next}\colon P\} \mid R) \quad (4)\\ &(\nu x_2x_1)(x_2 \lhd \mathit{next}.x_2(y).\overline{\mathit{loop}}\,x_2 \mid x_1 \rhd \{\mathit{next}\colon P, \mathit{hasNext}\colon Q\} \mid R)\end{aligned}$$

The types for the client and the server are dual in the first case, but not in the second. The client at x_2 asks $\oplus\{\mathit{next}\colon \ldots\}$ whereas the iterator at x_1 offers $\&\{\mathit{next}\colon \ldots, \mathit{hasNext}\colon \ldots\}$. One solution to the problem allows the server to "forget" options, thus obtaining a type $\&\{\mathit{next}\colon \ldots\}$, which is now dual to that of the client.

Subtyping has in our language the generally accepted meaning, where $T <: U$ indicates that any value of type T can be safely used in a context where a value of type U is expected, or "every value described by T is also described by U". The new rule is in Figure 11.

To feel how the subsumption rule works we build a derivation for the parallel composition of the the new iterator and its client. Let

$$\begin{aligned}T &= \mathsf{lin}\,\&\{\mathit{next}\colon T'\}\\ U &= \mathsf{lin}\,\&\{\mathit{next}\colon T', \mathit{hasNext}\colon U'\}\\ \Gamma_1 &= \Gamma_1', x_1\colon T\\ \Gamma_2 &= \Gamma_2', x_2\colon \overline{T}\\ P_1 &= x_1 \rhd \{\mathit{next}\colon P, \mathit{hasNext}\colon Q\}\\ P_2 &= x_2 \lhd \mathit{next}.x_1(y).\overline{\mathit{loop}}\,x_2\end{aligned}$$

in

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma_1 \vdash x_1\colon T} \quad T <: U}{\Gamma_1 \vdash x_1\colon U}\,[\text{T-Sub}] \qquad \begin{array}{c}\vdots\\ \Gamma_1, x_1\colon T' \vdash P\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma_1, x_1\colon U' \vdash Q\end{array}}{\Gamma_1 \vdash P_1} \qquad \begin{array}{c}\vdots\\ \Gamma_2 \vdash P_2\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2}$$

So here is our first rule of finite subtyping.

$$\dfrac{I \subseteq J \qquad T_i <: U_i \qquad \forall i \in I}{q\,\&\{l_i\colon T_i\}_{i\in I} <: q\,\&\{l_j\colon U_j\}_{j\in J}} \qquad [\text{S-BranchFin}]$$

Conversely, we can fix the mismatch between the new iterator and its client by allowing the client to select more options.

$$\frac{I \supseteq J \qquad T_j <: U_j \qquad \forall j \in J}{q \oplus \{l_i : T_i\}_{i \in I} <: q \oplus \{l_j : U_j\}_{j \in J}} \qquad \text{[S-SelFin]}$$

To understand why we have $T <: U$ in the hypothesis of rule [S-BranchFin], remember that type S in $\&\{next\colon S\}$ of the iterator above (equation 2) contains multiple (infinite, in fact) nested copies of $\&\{next\colon S\}$ itself, and we want each of them to be a subtype of the larger type $\&\{next\colon \ldots, hasNext\colon \ldots\}$. A similar reason conducts to the exactly same conclusion in the case of [S-SelFin]. For this reason, in all four session-type constructors, continuations are always co-variant.

There remains to study the input and output operations. Suppose the client delegates its variable just before selecting operation *next*. Towards this end, the client uses another channel (another pair of co-variables, y_1, y_2) and sends its variable x_1 on y_1. The receiver gets it in y_2 and calls the pending *next* operation on the iterator. The code for the receiver, the recipient of delegation, is as follows,

$$y_2(z).z \lhd next$$

where y_2 is naturally typed at $\mathsf{lin}\,?(\mathsf{lin} \oplus \{next\colon \ldots\})$. When the new iterator enters operation, the recipient of delegation can call *hasNext* (as well as *next*), and thus rewrites its code to become:

$$y_2(z).z \lhd hasNext$$

where y_2 is now typed at $?(\mathsf{lin} \oplus \{next\colon \ldots, hasNext\colon \ldots\})$. We have seen that $q \oplus \{next\colon \ldots, hasNext\colon \ldots\} <: q \oplus \{next\colon \ldots\}$. If we make

$$q?(q' \oplus \{next\colon \ldots, hasNext\colon \ldots\}) <:\ q?(q' \oplus \{next\colon \ldots\})$$

then the piece of code $y_2(z).z \lhd hasNext$ typed at the subtype can also be used where the supertype is expected.

The example allows us to conclude that input is co-variant. A similar reasoning on the iterator side would allow to conclude that output is contra-variant.

$$\frac{T' <: T \qquad U <: U'}{q!T.U <:\ q!T'.U'} \qquad \frac{T <: T' \qquad U <: U'}{q?T.U <:\ q?T'.U'} \qquad \text{[S-SendFin],[S-RcvFin]}$$

In summary:

- Input operations $(?, \&)$ are co-variant; output operations $(!, \oplus)$ contra-variant;
- Continuations are always co-variant.

Subtyping in the pi calculus is reversed with respect to that found in the lambda calculus. The notion of co-variable helps in understanding the phenomenon. Our understanding of $T <: U$ is that x_1 of type T can be safely used in a context where a type U is expected. Then it must be the case that the context uses, not x_1, but x_2 the co-channel, hence U must offer more choices, so that it may be used by $x_2 : \overline{U}$ that selects more choices.

Because of recursive types we use a co-inductive definition, rather than an inductive definition based on the rules we have sketched above.

Definition 1 (Subtyping). *Define the operator* $F \in \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ *as follows.*

$$\begin{aligned}
F(R) = {} & \{(q\,\mathsf{end}, q\,\mathsf{end}), (q\,\mathsf{bool}, q\,\mathsf{bool})\} \\
& \cup \{(q?T.U, q?T'.U') \mid (T, T'), (U, U') \in R\} \\
& \cup \{(q!T.U, q!T'.U') \mid (T', T), (U, U') \in R\} \\
& \cup \{(q\&\{l_i\colon T_i\}_{i\in I}, q\&\{l_j\colon T'_j\}_{j\in J}) \mid I \subseteq J, (T_i, T'_i) \in R, \forall i \in I\} \\
& \cup \{(q\oplus\{l_i\colon T_i\}_{i\in I}, q\oplus \{l_j\colon T'_j\}_{j\in J}) \mid I \supseteq J, (T_j, T'_j) \in R, \forall j \in J\} \\
& \cup \{(\mu a.T, T') \mid (T[\mu a.T/a], T') \in R\} \\
& \cup \{(T, \mu a.T') \mid (T, T'[\mu a.T'/a]) \in R\}
\end{aligned}$$

Contractivity ensures that F *is monotone. By the Knaster-Tarski theorem,* F *has least and greatest fixed points; we take the greatest fixed point to be the subtyping relation, writing* $T <: U$ *if the pair* (T, U) *is in the relation.*

Lemma 6. *Subtyping is a pre-order.*

As mentioned in Section 6 type equivalence is defined on top of subtyping: we say that types T and U are *equivalent*, and write $Y = U$, when $T <: U$ and $U <: T$. In this case we simply write and write $T = U$, which should not be confused with syntactic equality.

The interested reader may have notice that there is already a flavor of subtyping in rule [T-Sel], Figure 8, where given label l_j in a program, we guess the remaining labels in the $\oplus$-type. In fact equipped with subtyping, the rule can be simplified avoiding mentioning extraneous labels.

$$\frac{\Gamma_1 \vdash x\colon q \oplus \{l\colon T\} \qquad \Gamma_2 \circ x\colon T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l.P} \qquad \text{[T-SelSimple]}$$

9 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.

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 $(\nu xy\colon T)P$, where x is supposed to be of type T and y of type $\overline{T}$ in scope P. Changes are in Figure 12.

We have introduced our language piecewise. To simplify the exposition, we address in this section the language formed by the basics in Figure 5, extended

New syntactic forms

$$\begin{array}{lll} P & ::= & \ldots \qquad\qquad\qquad\qquad\qquad\qquad \text{Processes:} \\ & & (\nu xy\colon T)P \qquad\qquad \text{annotated scope restriction} \end{array}$$

Context difference

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div L = \Gamma_2, x\colon \mathsf{un}\, p, \Gamma_3}{\Gamma_1 \div (L, x) = \Gamma_2, \Gamma_3} \qquad \frac{\Gamma_1 \div L = \Gamma_2 \quad x\colon T \notin \Gamma_2}{\Gamma_1 \div (L, x) = \Gamma_2}$$

Typing rules for values

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

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

Typing rules for processes

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

$$\frac{\Gamma_1 \vdash v : q\,\mathsf{bool}; \Gamma_2 \qquad \Gamma_2 \vdash P : \Gamma_3; L_3 \qquad \Gamma_2 \vdash Q : \Gamma_3; L_3}{\Gamma_1 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Gamma_3; L_3} \qquad \text{[A-If]}$$

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

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

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

$$\frac{\Gamma_1 \vdash P : \Gamma_2; \emptyset}{\Gamma_1 \vdash *P : \Gamma_2; \emptyset} \qquad \text{[A-Repl]}$$

Fig. 12. Algorithmic type checking

with recursive types in Figure 9 and replication in Figure 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 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 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 algorithms and T, Γ_2, and L is the output. Set L collects linear (free) variables in P that occur in subject position,[1] and plays its role in the rule for parallel composition.

[1] A variable x occurs in subject position in processes $\overline{x}\, v.P$ and $x(y).P$.

The main change in the re-engineered type system is the treatment of linear variables (which has moved from the axioms (and rule[T-Repl]) to the rules that introduce assumptions in the context, [A-Res], [A-In]), of replication and of parallel composition, [A-Repl], [A-Par]. The base cases for variables and constants allow any context to pass through the judgement, even when containing linear types. Two rules, [A-UnVar] and [A-LinVar], replace the single rule for variables [T-Var] in Figure 5. The former keeps the entry $x\colon T$ in the returned context, the latter removes the entry.

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

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

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

$$\begin{gathered} x\colon \mathsf{lin}\,!\mathsf{bool}.\mathsf{lin}\,!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x\colon \mathsf{lin}\,!\mathsf{bool}); \{x\} \\ x\colon \mathsf{lin}\,!\mathsf{bool} \vdash \mathbf{0} : (x\colon \mathsf{lin}\,!\mathsf{bool}); \emptyset \\ x\colon \mathsf{lin}\,!\mathsf{bool}, y\colon *!(\mathsf{lin}\,!\mathsf{bool}) \vdash \overline{y}\,x : (y\colon *!(\mathsf{lin}\,!\mathsf{bool})); \{y\} \\ x\colon \mathsf{lin}\,!\mathsf{bool} \vdash \overline{x}\,\mathsf{true} : (x\colon \mathsf{end}); \{x\} \end{gathered}$$

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

$$\begin{gathered} x\colon \mathsf{lin}\,!\mathsf{bool}.\mathsf{lin}\,!\mathsf{bool}. \vdash \overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{false} : (x\colon \mathsf{end}) \\ x\colon \mathsf{lin}\,!\mathsf{bool}, y\colon *!\mathsf{end} \vdash \overline{x}\,\mathsf{true} \mid \overline{y}\,x : (x\colon \mathsf{end}, y\colon *!\mathsf{end}) \end{gathered}$$

but we know that $x\colon \mathsf{lin}\,!\mathsf{bool}.\mathsf{lin}\,!\mathsf{bool}. \not\vdash \overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{false}$ and $x\colon \mathsf{lin}\,!\mathsf{bool}, y\colon *!\mathsf{end} \not\vdash \overline{x}\,\mathsf{true} \mid \overline{y}\,x$. Instead, we collect in set L all linear (free) subjects in process P and use context difference to ensure that they do not remain linear in context Γ_2. Type checking continues with process Q in a context where the assumptions for the (unrestricted) names in L_2 have been removed.

Rule [A-Res] ensures that newly introduced linear variables are used to the end. The premise $\Gamma_1, x\colon T, y\colon \overline{T} \vdash P : \Gamma_2; L_2$ 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 always appear in Γ_2. The case for y is similar. Unrestricted types for x, y must be deleted from the outgoing context of the rule. To handle both the check that linear variables do not appear in contexts and the removal of unrestricted variables we use a *context difference* operator, $\div$, in Figure 12. Notation L, x denotes the set $L' = L \cup \{x\}$ where $x \notin L$. Using this operator, the outgoing context of the rule is $\Gamma_2 \div \{x, y\}$. Notice that this operator is undefined when we try to remove a variable of a linear type from a context. 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, to obtain 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 Figure 5, 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 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 rule for replication, [A-Repl], ensures that there are no (free) inputs and outputs on linear channels in process P by requiring an empty set 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 proof of equivalence can be broken in two standard parts, *soundness* and *completeness* of the algorithm with respect to the declarative system. 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 erase(P) that removes all types from an annotated process P. Function erase is a homomorphism everywhere, except at scope restriction where $\mathrm{erase}((\nu xy\colon T)P) = (\nu xy)\,\mathrm{erase}(P)$.

Notation $\mathcal{U}(\Gamma)$ and $\mathcal{L}(\Gamma)$ refers to the set of unrestricted and linear assumptions in Γ, respectively. The following basic properties of the context split operation are used in the remaining proofs.

Proposition 1. *Let Γ_1 and Γ_2 be two contexts such that $\Gamma_1 \circ \Gamma_2$ is defined. Then*

1. $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$;
2. $\Gamma_1 \circ \Gamma_2 = \mathcal{L}(\Gamma_1), \Gamma_2$;
3. $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\mathcal{L}(\Gamma_2)) = \emptyset$.

From clause 2 above, taking $\mathcal{U}(\Gamma_1)$ for Γ_2, we obtain as a corollary that $\Gamma_1 \circ \mathcal{U}(\Gamma_1) = \Gamma_1$. From the same clause, when $\mathsf{un}(\Gamma_1)$ we know that $\Gamma_1 \circ \Gamma_2 = \Gamma_2$ and, in particular $\Gamma_1 \circ \Gamma_1 = \Gamma_1$.

Lemma 7 (Algorithmic monotonicity).

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

The following lemma is used in the proof of soundness.

Lemma 8 (Algorithmic linear strengthening).

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

The proviso that x is not in L is important. Take for T the type $\mu\alpha.\mathsf{lin!bool}.\alpha$. We have

$$x\colon T \vdash \overline{x}\,\mathsf{true} : (x\colon T; \{x\})$$

where the type T of x is invariant, but we know that $\emptyset \nvdash \overline{x}\,\mathsf{true} : \emptyset; \_$.

The following lemma is used in the proof of completeness.

Lemma 9 (Algorithmic weakening).

Values. *If* $\Gamma_1 \vdash v : T_1; \Gamma_2$ *then* $\Gamma_1, x\colon T_2 \vdash v : (T_1; \Gamma_2, x\colon T_2)$*;*
Processes. *If* $\Gamma_1 \vdash P : \Gamma_2; L$ *then* $\Gamma_1, x\colon T \vdash P : (\Gamma_2, x\colon T; L)$*.*

Theorem 3 (Algorithmic soundness).

Values. *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*.*
Processes. *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 derivation of the sequent. The case for processes follows by induction on the structure of derivation of the hypothesis. The most interesting case happens when the derivation ends with rule [A-Par]. By induction we know that $\Gamma_2 \div L_2 \vdash \mathrm{erase}(Q)$ (*). We claim that there are Γ_4 and Γ_5 such that $\Gamma_4 \vdash P : \Gamma_5; L$, $\mathsf{un}(\Gamma_5)$ and $\Gamma_4 \circ (\Gamma_2 \div L_2) = \Gamma_1$. Then we use induction again to obtain $\Gamma_4 \vdash \mathrm{erase}(Q)$ (**), and conclude the proof by applying rule [T-Par] to (*) and (**). The claim follows from the definition of $\div$, monotonicity (lemma 7), strengthening (lemma 8), and proposition 1. □

Theorem 4 (Algorithmic completeness).

Values. *If* $\Gamma_1 \circ \Gamma_2$ *defined and* $\Gamma_1 \vdash v : T$ *then* $\Gamma_1 \circ \Gamma_2 \vdash v : T; \Gamma_2$*;*
Processes. *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 derivation of the sequent. The case for processes follows by induction on the structure of derivation of the hypothesis. The most interesting case happens when the derivation ends with rule [T-Par]. By induction we know that $\Gamma_1 \vdash P : \Gamma_3; L_3$ and $\mathsf{un}(\Gamma_3)$. Since $\Gamma_1 \circ \Gamma_2 = \Gamma_1, \mathcal{L}(\Gamma_2)$, we weaken the derivation to obtain $\Gamma_1 \circ \Gamma_2 \vdash P : \Gamma_3, \mathcal{L}(\Gamma_2); L_3$ (*). Again by induction we know that $\Gamma_2 \vdash Q : \Gamma_4; L_4$ (**) and $\mathsf{un}(\Gamma_4)$. We claim that $(\Gamma_3, \mathcal{L}(\Gamma_2)) \div L_3 = \Gamma_2$, and apply rule [A-Par] to (*) and (**) to obtain the result, $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q : \Gamma_4; L_4$ and $\mathsf{un}(\Gamma_4)$. The claim follows from the definition of $\div$, monotonicity (lemma 7) and proposition 1. □

10 Notes

Session types for the pi calculus. Work on session types goes back to Honda and its colleagues at Keio University—Kubo, Takeuchi, and Vasconcelos—first centering on the type structure, then introducing the notion of channel, and finally extending the ideas into a more general setting [18, 19, 28]. The original work introduces session types, describing chained continuous interactions composed of communication (input and output) and binary choice [18]. 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 common 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 [18]. The language in reference [19] 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 reference [19], channel passing embodies a technique similar to internal mobility [27] whereby the sender and the receiver must agree on the exact channel being handed over, *prior to communication* itself. Using the notation of this paper and forgoing the variable convention, if x and y are linear co-variables, the rule for communicating a linear variable z is of the following form where z is both free in $\overline{x}\, z.P$ and bound $y(z).Q$,

$$\overline{x}\, z.P \mid y(z).Q \;\rightarrow\; P \mid Q$$

with the understanding that if the receiving process happens to look like $y(w).Q$ then the bound variable w is renamed as z prior to reduction, *if possible*.

Gay and Hole proposed a variant to this work by introducing two novelties: they work directly on the pi calculus and use free session passing [14]. Their language is similar to that in this paper, except for one small detail: it annotates variables with polarities $+, -$. The new reduction rule for session passing is a pi calculus conventional communication rule (x and y are co-variables).

$$\overline{x}\, v.P \mid y(z).Q \;\rightarrow\; P \mid Q[v/z]$$

Rather than using distinct identifiers x, y that are made co-variables at binding time $(\nu xy)P$, they use one identifier only (x) with polarity annotations (x^+, x^-) that is bound as a single variable in process $(\nu x)P$. The relation that associates x^+ to x^- is left implicit in reduction. In either case, the reason behind the need for syntactically distinguishing the two ends of a 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.

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 [14], polarity annotated variables are associated to channels; names use non-annotated variables. As such, there are two communication rules: for channels, on processes of the form $\overline{x^+}\, v.P \mid x^-(z).Q$, and for names on processes $\overline{x}\, v.P \mid x(z).Q$. We work with co-variables in all cases, using a single communication rule for processes of the form $\overline{x}\, v.P \mid y(z).Q$ where x and y are co-variables. If needed the distinction between channels and names is made by the type qualifiers associated to variables x and y, linear or unrestricted.

Yoshida and Vasconcelos use the polarity technique to endow the language in [19] with free session passing [34]. 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. Interesting enough, channel polarities were used in [28], then dropped in [19], and finally recovered in [14].

The technique of binding the two ends of a channel together is due to Gay and Vasconcelos [16], 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. The same idea is explored by Giunti el al. [17] to show that the language in [34] equipped with the type system in [19] is type safe even though it does not satisfy type preservation.

Typing and subtyping. Due to delegation [19], types are usually stratified into two categories: one for sessions, the other for names. Types for channels include constructors for input, output, branching, and selection. Those for names include the standard pi calculus types. The separation of the two universes leads to duplication (recursion, input/output) and omissions (there is no choice on names). We take a different approach, starting from *pretypes* comprising all the basic types and required constructors, and then using a linear or unrestricted qualifier depending on the intended usage for the variable, channel or name.

Subtyping as presented in this paper was first introduced by Gay and Hole [14], co-inductively given the presence of recursive types. Rather than using a separate subsumption rule, Gay and Hole distribute the possible occurrences of subtyping by the relevant typing rules. They further present an algorithm for checking the subtyping relation, used for type checking their language. A proof for Lemma 6 in Section 8 can be found in reference [14].

Gay introduces a notion of bounded polymorphism for the pi calculus with session types [13] where polymorphism is associated with labels in branching processes, in such a way that clients selecting a particular branch also instantiate the polymorphic variable with some type. Capecchi, Dezani-Ciancaglini et al. propose a variant of session types for object-oriented languages where choice is provided, based not on labels, but on classes [2, 4]. Castagna, Dezani-Ciancaglini et al. propose a set-theoretic semantics for session types based on a labelled tran-

sition system and on a coinductively-defined notion of duality [6]. The semantics yields a notion of subtyping and they present an algorithm for deciding the relation. The session types considered in the paper generalize those found in this work by replacing constructors for branching with boolean expressions.

Linear type systems. A linear type system for the pi calculus was studied by Kobayashi, Pierce and Turner [21]. 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 naturally occurring more than once in a thread. Instead, a linear channel in this work is understood as occurring in a single thread, possibly multiple times. The machinery used here, linear and unrestricted type qualifiers and context splitting, is inspired by Walker's substructural type systems [33].

Session types in functional languages. Session types emerged in conjunction with process calculi. Gradually, the notion was adapted to other paradigms, including functional languages, object-oriented languages and service-oriented computing. Together, Gay, Ravara, and Vasconcelos proposed the first functional language with session types [16, 30, 32]. Neubauer and Thiemann [25] took a different approach, embedding session types within the type system of Haskell. Similarly to this chapter, the language in reference [16] works within the standard framework the linear lambda calculus, treating session types as linear in order to guarantee that each co-channel is owned by a unique thread. For example, the type of the receive operation is $?T.U \to T \otimes U$ so that the channel, with its new type U, is returned together with the received value T.

Session types in object-oriented languages. The area of session types for object-oriented languages has attracted a lot of attention. The work by Vallecillo, Vasconcelos, and Ravara [29] shows how to type the behavior of objects in component models, Corba in particular (the example in the introduction is taken from this work). Starting with the work by Dezani-Ciancaglini, Yoshida et al. [11] that incorporates channel-based communication in a Java-like language, many have followed, including [2, 4, 7, 9, 10, 20]. A characteristic of these works is that a channel is always created and completely used within a single method call, or else delegated to another method which will have to use the channel to the end. Mostrous and Yoshida [23] add sessions to Abadi and Cardelli's object calculus [1]. Vasconcelos, Gay, et al. use session types to describe the evolving visible interface of an object, according to the object's state [31].

Session types in service-oriented computing. The natural ability of session types to describe protocols have been explored in the realm of service-oriented calculi. Works like [3, 5, 8, 22], to cite a few, use session types to discipline the interaction between service providers and clients.

Buffered semantics for session types. Neubauer and Thiemann first proposed an asynchronous, buffered semantics, allowing two communicating partners to proceed at distinct rates [26]. The idea is to associate to each co-variable a buffer to hold both values and labels—readers (input and branching processes) read from their own buffer; writers (output and selecting processes) write on the co-channel buffer. Gay and Vasconcelos propose a simpler buffered semantics [15, 16]; Fähndrich et al. [12] also use buffered communication but have not published a formal semantics.

An interesting application of session types for buffered communication is that buffer size can be predicted from the session type that describes the channel, thus ensuring that well-typed programs do not overflow their buffers. This fact is observed in [12] and proved in [16], where it is shown that static type information can be used to decrease the runtime buffer size and ultimately deallocate the buffer. Another application explores optimisations by exchanging the order by which certain communications are performed, allowing for a large transfer to proceed in front of other lighter transfers [24]. The valid communication exchanges are captured by a subtyping relation.

Multi-party session types. The language of this chapter disciplines the interaction between two threads; sessions types to describe interaction among multiple partners is the object another chapter in this book.

Acknowledgments. The author was partially supported by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004). He thanks Marco Giunti and Francisco Martins for advice and suggestions.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and union types for object oriented programming. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 659–680. Springer, 2008.
3. Michele Boreale, Roberto Bruni, Rocco Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In *Proceedings of FMOODS'08*, LNCS, pages 19–38. Springer, 2008.
4. Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science*, 410(2-3):142–167, 2009.
5. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Proceedings of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
6. Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundation of session types. Unpublished, 2009.
7. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous session types and progress for object-oriented languages. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.

8. Luís Cruz-Filipe, Ivan Lanese, Francisco Martins, António Ravara, and Vasco T. Vasconcelos. Behavioural theory at work: program transformations in a service-centred calculus. In *Proceedings of FMOODS'08*, volume 5051 of *LNCS*, pages 59–77. Springer, 2008.
9. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded session types for object-oriented languages. In *Proceedings of FMCO'07*, volume 4709 of *LNCS*, pages 207–245. Springer, 2007.
10. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. In *Proceeding of ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
11. Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopolou. A distributed object-oriented language with session types. In *Proceedings of TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer, 2005.
12. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Operating Systems Review*, 40(4):177–190, 2006.
13. Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.
14. Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
15. Simon J. Gay and Vasco T. Vasconcelos. Asynchronous functional session types. TR 2007–251, Department of Computing, University of Glasgow, May 2007.
16. Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. Submitted, 2008.
17. Marco Giunti, Kohei Honda, Vasco T. Vasconcelos, and Nobuko Yoshida. Session-based type discipline for pi calculus with matching. In *PLACES'09*, 2009.
18. Kohei Honda. Types for dyadic interaction. In *Proceedings of CONCUR '93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
19. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP '98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
20. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Proceedings of ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
21. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
22. Ivan Lanese, Vasco T. Vasconcelos, Francisco Martins, and António Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
23. Dimitris Mostrous and Nobuko Yoshida. A session object calculus for structured communication-based programming. Unpublished, 2008.
24. Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *Proceedings of ESOP'09*, LNCS. Springer, 2009.
25. Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of PADL'04*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
26. Matthias Neubauer and Peter Thiemann. Session types for asynchronous communication. Unpublished, 2004.

27. Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(1,2):235–274, 1996.
28. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE '94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
29. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.
30. Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.
31. Vasco T. Vasconcelos, Simon J. Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Dynamic interfaces. In *FOOL'09*, 2009.
32. Vasco T. Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In *Proceedings of CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004.
33. David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.
34. Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *Proceedings of SecReT'07*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier Science, 2007.