Towards Static Deadlock Resolution in the π-Calculus

Marco Giunti and António Ravara(✉)

CITI and DI-FCT, Universidade Nova de Lisboa, Lisbon, Portugal
aravara@fct.unl.pt

Abstract. Static analysis techniques based on session types discern concurrent programs that ensure the fidelity of protocol sessions – for each input (output) end point of a session there is exactly an output (input) end point available – being expressive enough to represent the standard π-calculus and several typing disciplines. More advanced type systems, enforcing properties as deadlock-freedom or even progress, sensibly reduce the set of typed processes, thus mining the expressiveness of the analysis. Herein, we propose a first step towards a compromise solution to this problem: a session based type checking algorithm that releases some deadlocks (when co-actions on the same channel occur in sequence in a thread). This procedure may help the software development process: the typing algorithm detects a deadlock, but instead of rejecting the code, fixes it by looking into the session types and producing new safe code that obeys the protocols and is deadlock-free.

1 Introduction

Background and related work. Session types, introduced for a dialect of the π-calculus of Milner et al. [1], allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol [2–5]. A key property is type-safety, which ensures that well-typed processes cannot go wrong in the sense that they do not reach neither the usual data errors, as those produced in this case by the use of base values as channels, nor *communication* errors, as those generated by two parallel processes waiting in input on a same session channel, or sending in output on the same session channel. An important feature is session delegation — the capacity to pass on the processing of a session. This is relevant for many purposes, e.g. it permits to design a FTP server that requires the presence of a daemon and of a pool of threads that will serve the client's request picked by the daemon [3].

While many session typing systems require a means to distinguish the two ends of a session channel in order to preserve type soundness [6–8], recently the first author and Vasconcelos have developed a session typing system [9,10] on top of the standard π-calculus. The main benefit is expressiveness: session delegation is described by the π-calculus communication mechanism; type-disciplines based on session [3,7] and linear [11] types can be embedded in the framework.

M. Abadi and A. Lluch Lafuente (Eds.): TGC 2013, LNCS 8358, pp. 1–20, 2014.
DOI: 10.1007/978-3-319-05119-2_9,

A drawback of most of these systems is accepting processes that exhibit various forms of deadlocks — although type safety is guaranteed, they do not ensure deadlock-freedom. For that aim, several proposals appeared recently, guaranteeing progress by inspecting causality dependencies in the processes [12–16]. Not surprisingly, these systems reduce the set of typed processes, rejecting (as usual in static analysis, which is not complete) deadlock-free processes.

Motivating example. To illustrate the problem that we are tackling in this paper, consider a synchronous π-calculus with session-based communication channels and boolean values. Take a process which behaviour consists in testing a boolean variable and, on success, in acknowledging the result on a received channel and then continuing the computation, like $a(x).b(z).\mathsf{if}\ z\ \mathsf{then}\ \overline{x}\langle\mathsf{true}\rangle.P'$. The process is well behaved and our type-system accepts it, but, the interaction with a process sending channel b can cause a runtime deadlock; in fact, several systems in the literature reject the process P below

$$P \stackrel{\text{def}}{=} (\nu a, b)(a(x).b(z).\mathsf{if}\ z\ \mathsf{then}\ \overline{x}\langle\mathsf{true}\rangle.P' \mid \overline{a}\langle b\rangle.\overline{a}\langle\mathsf{false}\rangle)$$

since

$$P \to (\nu a, b)(b(z).\mathsf{if}\ z\ \mathsf{then}\ \overline{b}\langle\mathsf{true}\rangle.P' \mid \overline{a}\langle\mathsf{false}\rangle)$$

The well-known problem of delegating a linear channel already in use by the receiver (cf. [8]) is avoided *a priori*; as a (unfortunate) by-product, legitimate processes of the form below are rejected as well:

$$LP \stackrel{\text{def}}{=} (\nu a, b)(a(x).\overline{d}\langle b\rangle.\overline{x}\langle\mathsf{true}\rangle \mid \overline{a}\langle b\rangle \mid P'')$$

The approach that we take is radically different: we aim at both type check P and at transform it in a process that is deadlock-free. To type P, we assign to a a pair type of the form $(?T_1.?\mathsf{bool}.\mathsf{end}, !T_1.!\mathsf{bool}.\mathsf{end})$, where the left entry says that an *end point* of a (actually, the one in the left thread of P) is used linearly to first receive a value of type T_1, and then to receive a boolean, while the right entry describes the dual behaviour of first sending a value of type T_1 and then a boolean, which occurs in the right thread. We assign to b the type $T \stackrel{\text{def}}{=} (?\mathsf{bool}.\mathsf{end}, !\mathsf{bool}.\mathsf{end})$; that is, we *split* T into $T_1 \stackrel{\text{def}}{=} (\mathsf{end}, !\mathsf{bool}.\mathsf{end})$ and $T_2 \stackrel{\text{def}}{=} (?\mathsf{bool}.\mathsf{end}, \mathsf{end})$, and let T_1 be delegated over a, and T_2 be used to type the input on b. The key idea towards the transformation of P into a deadlock-free process is to exploit the structure of the session type T, and check whether the end point channels described respectively by T_1 and T_2 are used sequentially, rather than in parallel.

In this paper, we still not deal with process P, but we handle processes resulting from it by a linear scan, using a constraint-based rewriting procedure.[1]

$$Q \stackrel{\text{def}}{=} (\nu a, b)(a(x).x(z).\mathsf{if}\ z\ \mathsf{then}\ \overline{x}\langle\mathsf{true}\rangle.P' \mid \overline{a}\langle b\rangle.\overline{a}\langle\mathsf{false}\rangle)$$

[1] The procedure is still at an experimental stage; we do not discuss it here.

Process Q above, resulting from P using the procedure, is typed by assigning to a the type $(?T.\mathsf{end}, !T.\mathsf{end})$. What is important is that our type checking algorithm, while typing Q and inferring that the variable x bound by the input on a has type T, detects that the two linear endpoints of x described respectively by T_1 and T_2 are used sequentially in i/o, and proposes a "fix". We generate new code for Q that mimics the behaviour described by the session type T, which, in principle, is the desired behaviour of (the communication channel of) Q. The type-assisted compositional translation $[\![\cdot]\!]$ maps the typed sequential continuation $Q' \stackrel{\text{def}}{=} x(z).\mathsf{if}\, z\, \mathsf{then}\, \overline{x}\langle\mathsf{true}\rangle.P'$ in a parallel process, using a forwarder (r) and a semaphore (m) (to impose, in the source and in the translated processes, the same order of communication); note that $v = \mathsf{true}$ is a parameter of the translation, obtained by a linear scan of Q.

$$[\![Q']\!]_v \stackrel{\text{def}}{=} (\nu r, m)(x(y).\overline{r}\langle y\rangle \mid r(z).\mathsf{if}\, z\, \mathsf{then}\, \overline{m}\langle\mathsf{true}\rangle.P' \mid \overline{x}\langle v\rangle.m(w))$$

After some confluent reduction steps, the channel sent over a is forwarded to the receiver, which can finally successfully pass the boolean test:

$$(\nu a, b)(a(x).[\![Q']\!]_v \mid \overline{a}\langle b\rangle.\overline{a}\langle\mathsf{false}\rangle) \rightarrow^* (\nu m)(\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, \overline{m}\langle\mathsf{true}\rangle.P' \mid m(w) \mid \overline{a}\langle\mathsf{false}\rangle)$$

Note that the type checking algorithm does not apply any transformation to the legitimate (rewriting of) process LP (defined above), because the endpoint of channel x that corresponds to T_2 is not used in i/o (it is indeed delegated).

Contribution. Distributed programming is known to be very hard and one makes mistakes by not taking into consideration all possible executions of the code. Therefore, to assist in the software developing process, instead of simply rejecting a process that may contain a *resource self-holding deadlock* (*RSHDF* , i.e., one or more input and output on the same channel occur in sequence in a given thread, an instance of *Wait For* deadlocks [17,18]), we devise a type checking algorithm that produces a fix for this kind of deadlocked processes by a program transformation. We show that our program transformation is both *RSHDF* -deadlock-free and error-free. These properties are achieved by relying on automatic decoration of typed channels, and on light-weight verification of the format of decorations, thus avoiding the use of type contexts and systems.

Plan of the paper. In Sect. 2, we introduce a typed π-calculus with sessions obeying a linear discipline, and review the safety properties of the typing system. In Sect. 3, we define the class of resource-holding and of deadlock-free processes, showing the latter closed under reduction. In Sect. 4, we present an untyped π-calculus with decorated channels: as decorations are based on types, we do not need to rely on typing information to identify safe processes. In Sect. 5, we devise a split-free type checking algorithm that projects typed processes into decorated ones. The aim is two-fold: (i) assess the typability of a typed process given a context; and (ii) generate untyped, resource-deadlock free code. We conclude presenting the main results of the algorithm: it accepts processes typed by the split-based type system; and any process generated by the algorithm: (i) does not reach errors during the computation; (ii) is resource-holding deadlock-free. The proofs of the results presented herein are in a technical report (cf [19]).

2 The Source Language: π-Calculus with Session Types

We present the syntax and the (static and dynamic) semantics of the monadic, choice-free, synchronous π-calculus, equipped with session types, our source language. Then we state the main properties ensured by the type system.

Syntax of processes and types. Let P, Q range over the set of *processes* $\mathcal{P}$, T range over *types* $\mathcal{T}$ and R, S over *session types*, and Γ range over *typing contexts* (or environments) $\mathcal{G}$, which are maps from variables x, y, z to types; *values* v, w are variables and the boolean constants true and false. The grammar in Fig. 1 defines the language, which is standard (cf. [1], but uses type annotations in restriction). We consider types T composed by channel types of the form (R, S), where R and S are session types, each describing an end point of a session, and the boolean type. An end point of a session S finishes with the type end. A type of the form $!T.S$ describes a channel that is used exactly once to send a value of type T, and then is used as prescribed by S, following a linear discipline. Similarly, $?T.S$ describes a channel that is used exactly once to receive a value of type T, and then is used as imposed by S. The type end describes an end point of a session on which no further interaction is possible.

Considering the usual notions of free and bound variables, α-conversion, as well as of substitution, cf. [20], we use $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$ to indicate respectively the set of free and bound variables of P, which we assume disjoint by following Barendregt's variable convention [21], and let $\mathrm{var}(P)$ be the union of $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$. A process P is *closed* whenever $\mathrm{var}(P) = \mathrm{bv}(P)$.

The processes of our language are thus synchronous output and input processes, in the forms $\overline{x}\langle v\rangle.P$ and $x(y).P$: the former sends a value v over channel x to P, the latter waits on x for a value v that will substitute the bound occurence of y in P, noted $P[v/y]$. Notice that substitution is not a total function; it is not defined, e.g., for $(\overline{y}\langle\mathsf{false}\rangle)[\mathsf{true}/y]$. When writing $P[v/y]$ we assume that the substitution operation involved is defined. The restricted process $(\nu y\colon T)P$ creates a variable y decorated with the type T; the occurrences of y in P are bound. Boolean values are contrasted using if-then-else. The remaining processes are parallel composition, replication, and inaction. We ignore trailing $\mathbf{0}$'s and write $(\nu\tilde{x}\colon\tilde{T})P$ as a shortcut for $(\nu\tilde{x}_1\colon T_1)\cdots(\nu\tilde{x}_n\colon T_n)P$, with $n \geq 0$.

Dynamic semantics: reduction. Following standard lines, we describe the operational semantics of processes through a reduction relation, and allow to rearrange processes with structural congruence. The congruence rules are standard; we note that the second rule in the second line allows to remove a restriction provided that the session type has been consumed.

The reduction rules are also standard. The only variation is that we record, as a label of the reduction arrow, the variable where the (free) synchronisation takes place (similarly to [7,11]); this is convenient, and has no semantic impact, allowing to represent the progression of type decorations in restricted processes through the next operator over types. Let μ range over variables x, y and the symbol τ, which we assume reserved (not occurring in the syntax of processes).

Syntax of typed processes

T	::=	Types	P, Q	::=	Processes
	(S, S)	session		$\overline{x}\langle v\rangle.P$	output
	bool	boolean		$x(y).P$	input
S	::=	End point		$(\nu y\colon T)P$	restriction
	$?T.S$	input		$\mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
	$!T.S$	output		$(P \mid Q)$	composition
	end	termination		$!P$	replication
v	::=	Values		$\mathbf{0}$	inaction
	$\mathsf{true}, \mathsf{false}$	constant			
	x, y	variable			

Operator for type progression

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

Rules for structural congruence

$$(P \mid Q) \equiv (Q \mid P) \qquad ((P_1 \mid P_2) \mid P_3) \equiv (P_1 \mid (P_2 \mid P_3)) \qquad (P \mid \mathbf{0}) \equiv P \qquad !P \equiv (P \mid !P)$$
$$((\nu y\colon T)P \mid Q) \equiv (\nu y\colon T)(P \mid Q), \quad \text{if } y \notin \mathrm{fv}(Q) \qquad (\nu y\colon (\mathsf{end}, \mathsf{end}))\mathbf{0} \equiv \mathbf{0}$$
$$(\nu y_1\colon T_1)(\nu y_2\colon T_2)P \equiv (\nu y_2\colon T_2)(\nu y_1\colon T_1)P \qquad P \equiv Q, \quad \text{if } P =_\alpha Q$$

Rules for reduction

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

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

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

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

Fig. 1. Typed π-calculus

Moreover, let $\Rightarrow$ indicate the reflexive and transitive closure of $\xrightarrow{\mu}$, whenever the labels are irrelevant.

Static semantics: type system. The type system uses a notion of type and context split (cf. Walker's chapter in Pierce's book [22]), noted $\circ$, defined in Fig. 2. Formally, split is a three-argument relation. We write $\Gamma_1 \circ \Gamma_2$ to refer to a type environment Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Figure 3 contains a typing system with judgements of the form $\Gamma \vdash P$, where we assume that $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{bv}(P) \cap \mathrm{dom}(\Gamma) = \emptyset$. We make use of predicates on types and contexts, *balanced* and *terminated* (noted respectively bal and term). Balancing relies on the standard duality notion of session types; we let $\overline{S}$ be the dual of S:

$$\overline{?T.S} = !T.\overline{S} \qquad \overline{!T.S} = ?T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \mathsf{bal}((S, \overline{S}))$$

Type split rules

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

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

Context split rules

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

Fig. 2. Type and context split

Typing rules for values

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

Typing rules for processes

$$\frac{\Gamma, x\colon (S, R), y\colon T \vdash P}{\Gamma, x\colon (?T.S, R) \vdash x(y).P} \qquad \frac{\Gamma_1 \vdash v\colon T \qquad \Gamma_2, x\colon (S, R) \vdash P}{\Gamma_1 \circ (\Gamma_2, x\colon (!T.S, R)) \vdash \overline{x}\langle v\rangle.P} \qquad \text{[T-In-L],[T-Out-L]}$$

$$\frac{\Gamma_1 \vdash v\colon \mathsf{bool} \qquad \Gamma_2 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q} \qquad \text{[T-If]}$$

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

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

Fig. 3. Type system

Note that booleans are not balanced, as we do not consider open processes of the form $\mathsf{if}\, y\, \mathsf{then}\, P\, \mathsf{else}\, Q$, or closed processes of the form $(\nu x\colon \mathsf{bool})P$. The terminated types are $(\mathsf{end}, \mathsf{end})$ and bool: $\mathsf{term}((\mathsf{end}, \mathsf{end}))$ $\mathsf{term}(\mathsf{bool})$.

The typing rules are inspired by the system of Giunti and Vasconcelos [9], and represent a subsystem of its recent re-formulation [10], to which we refer for all details. We note that (a) we have left and right rules for typing input and output processes, corresponding to the cases whether the type for the prefix is on the left or on the right: for compactness, we only indicate left rules; (b) Rule [T-Out-L] allow session delegation by means of context split: for instance, if $\Gamma \vdash P$ with $\Gamma = \Gamma_1 \circ (\Gamma_2, x\colon (!T_1.S, R))$, then v is both sent at type T_1 and used at type T_2 in the continuation, whereas $\Gamma(v) = T_1 \circ T_2$.

Results. The type system guarantees the usual type preservation and safety properties — basic values are not used as channels (for synchronisation) and channels are always used linearly — when considering *balanced* contexts, i.e., contexts

mapping variables to balanced types (processes must send and receive values of the same type on both end points of a same channel). The proof of these results can be found in a recent work of Giunti [23].

Theorem 1 (Subject Reduction). *Let Γ be balanced. If $\Gamma \vdash P$ and $P \Rightarrow P'$ then there is Γ' balanced such that $\Gamma' \vdash P'$.*

The main result of this section is that typed processes do not reach errors during the computation. Besides basic errors of the form $\overline{x}\langle \mathsf{true} \rangle.P \mid x(y).y(z).P$, where true can be send through [R-Com] but the substitution is not defined, we consider errors due to non-linear use of channels, as in the parallel compositions $\overline{x}\langle v \rangle.P_1 \mid \overline{x}\langle w \rangle.P_2$ and $x(y).P_1 \mid x(y).P_2$.

Definition 1 (Error Process). *A process R is a* error*, if it is of the form $R \equiv (\nu \tilde{z}\colon \tilde{T})(P \mid Q)$, for some x, v and w, where (i) $P = \overline{x}\langle v \rangle.P_1 \mid \overline{x}\langle w \rangle.P_2$, or (ii) $P = x(y).P_1 \mid x(z).P_2$.*

Theorem 2 (Type Safety). *If $\Gamma \vdash P$ with Γ balanced, and $P \Rightarrow Q$, then Q is not an error.*

In short, although session type systems accept processes with non-deterministic behaviour (due to the behaviour of parallel composition), the behaviour of each session *is* deterministic, as communication channels (used for synchronisation) must be used *linearly*. In particular, a session type system rules out a process like $a(x).\mathbf{0} \mid \overline{a}\langle v \rangle.a(x).\mathbf{0} \mid \overline{a}\langle u \rangle$, since the communication order cannot be guaranteed, but accepts deadlocks like $\overline{a}\langle v \rangle.a(x).\mathbf{0}$, $a(x).\overline{a}\langle x \rangle$, or even like $a(x).\overline{b}\langle u \rangle \mid b(x).\overline{a}\langle v \rangle$.

3 Resource-Holding Deadlocks

The aim now is to introduce a syntactic (untyped) characterisation of processes that do not contain deadlocks due to the self-holding of resources; this is a simplified variant of *Hold and Wait* or *Resource-Holding* deadlocks. Our formulation of the property is such that it is preserved by reduction, and it has a simple, decidable, proof technique to verify if the property holds for a given process. In Sect. 6 we discuss how we envision to tackle the general deadlock resolution problem for the π-calculus. We motivate first the formal definition through examples. Then present it rigourously and develop the proof technique.

Resource self-holding deadlocks: motivation. In the following we analyse *balanced, typable, self-hold and wait* deadlocks while leaving type decorations implicit. Deadlocked processes like $(\nu a)(\nu b)(a(x).\overline{b}\langle \mathsf{true} \rangle \mid b(y).\overline{a}\langle \mathsf{false} \rangle)$ are *not* resource self-holding deadlocks, and are not addressed by our analysis technique. Intuitively, a process exhibits a resource self-holding deadlock if both ends of a (private) channel appear in sequence — communication on that channel is not possible. The basic examples[2] are the processes $(\nu a)\ \overline{a}\langle \mathsf{true} \rangle.a(y)$ and $(\nu a)\ a(y).\overline{a}\langle \mathsf{true} \rangle$, which

[2] While most of the examples do not require the use of restriction, we limit the scope of channels to help the comprehension of the reader.

contain a resource self-holding deadlock, since no communication on a can occur, as the co-actions appear in sequence, instead of in parallel. More intricate resource self-holding deadlocks include processes of the form $(\nu a)(\nu b)(\overline{b}\langle \mathsf{false}\rangle.(a(x).b(y) \mid \overline{a}\langle \mathsf{true}\rangle))$, or of the form

$$(\nu a)(\nu b)(a(x).b(y).\overline{x}\langle \mathsf{true}\rangle \mid \overline{a}\langle b\rangle) \tag{1}$$

$$a(x).b(y).\overline{a}\langle y\rangle.(x(z) \mid \overline{x}\langle \mathsf{true}\rangle) \mid \overline{b}\langle c\rangle \tag{2}$$

Process (1) is a simple variant of the process P of the introduction which delegates one end point of a session; process (2) describes a delicate situation involving binders. Our algorithm still not deal with these class of processes: in (1) we should predict that the left thread will receive b, while in (2) we cannot simply put the output on a in parallel, as the variable y would escape its scope. In Sect. 6 we envision how we could tackle these deadlocks.

Resource self-holding deadlocks: formally. We consider the following auxiliary notions on multisets, and let $\Cap$ be multiset intersection, $\Cup$ multiset union, $\Subset$ multiset inclusion, and $\setminus$ multiset difference. The *subject variables* of P, noted $\mathrm{subjv}(P)$, is the *multiset* with the occurrences of $x \in \mathrm{var}(P)$ identified by the rules (1) $\mathrm{subjv}(x(y).P) = \{x\} \Cup \mathrm{subjv}(P)$, and (2) $\mathrm{subjv}(\overline{x}\langle v\rangle.P) = \{x\} \Cup \mathrm{subjv}(P)$, the remaining productions being homomorphic. The *x-variables* of P, noted $x(P)$, is the subset identified by $\{x, x, \cdots\} \Cap \mathrm{subjv}(P)$, where $\{x, x, \dots\}$ is a countably infinite multiset of x.

Definition 2 (Sequential and Parallel Variables). *The* sequential variables *of a process P, noted* $\mathrm{sv}(P)$*, is the submultiset of* $\mathrm{subjv}(P)$ *identified by the rules (the remaining cases are homomorphic):*

1. $\mathrm{sv}(x(y).P) = \{x\} \Cup \mathrm{sv}(P)$
2. $\mathrm{sv}(\overline{x}\langle v\rangle.P) = \{x\} \Cup \mathrm{sv}(P)$
3. $\mathrm{sv}(P \mid Q) = \mathrm{sv}(P) \Cup \mathrm{sv}(Q) \setminus ((\mathrm{sv}(P) \Cap \mathrm{sv}(Q)) \Cup (\mathrm{sv}(P) \Cap \mathrm{sv}(Q)))$
4. $\mathrm{sv}(\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q) = \mathrm{sv}(P) \Cup \mathrm{sv}(Q)$

The parallel variables *of a process P, noted* $\mathrm{pv}(P)$*, is* $\mathrm{pv}(P) \stackrel{def}{=} \mathrm{subjv}(P) \setminus \mathrm{sv}(P)$.

Parallel variables are those that occur as subjects in different threads. A process does not contain resource-holding deadlocks if every channel has a matching pair in another thread, giving it a chance to interact.

Definition 3 (Resource Self-Holding Deadlock Freedom). *A process P is* Resource Self-Holding Deadlock-Free (RSHDF)*, if* $\mathrm{sv}(P) = \emptyset$*, or, equivalently, if* $\mathrm{subjv}(P) = \mathrm{pv}(P)$.

Examples of (balanced typed) *RSHDF* processes include:

1. $\overline{a}\langle b\rangle.\overline{d}\langle c\rangle.a(x) \mid a(y).\overline{a}\langle \mathsf{true}\rangle \mid d(z)$ and
2. $(\nu a)(\nu b)(a(x).(\overline{x}\langle \mathsf{true}\rangle \mid b(z).x(y)) \mid \overline{b}\langle \mathsf{true}\rangle.\overline{a}\langle c\rangle)$.

Processes containing sequential variables (typed or not), and thus not *RSHDF* , are for instance:

1. $(\nu a)\overline{a}\langle \mathsf{true}\rangle \mid b(y)$, or $\overline{a}\langle b\rangle \mid a(x).(\overline{x}\langle \mathsf{true}\rangle \mid c(y))$, or even $(\nu a)(\overline{a}\langle b\rangle \mid \overline{c}\langle a\rangle)$;
2. $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \overline{a}\langle b\rangle \mid a(y)\ \mathsf{else}\ \overline{a}\langle b\rangle.a(y)$.

We do not consider processes not typable by a balanced environment like:

1. $\overline{a}\langle b\rangle.\overline{d}\langle c\rangle.a(x) \mid \overline{a}\langle \mathsf{true}\rangle.a(y) \mid d(z)$, or $\overline{a}\langle \mathsf{true}\rangle \mid \overline{a}\langle \mathsf{false}\rangle$, or even
2. $a(x).P_1 \mid \overline{a}\langle \mathsf{true}\rangle.P_2 \mid a(y).P_3$,

which are *RSHDF* .

The main result of this section is that *RSHDF* is closed under reduction.

Theorem 3 (RSHDF preservation). *If $\Gamma \vdash P$ with Γ and P balanced, P is RSHDF, and $P \Rightarrow P'$, then P' is RSHDF.*

Strongly-balanced processes and environments. It is useful to analyse the shape of types of our interest. The invariant we rely on is that if a session type provides for sending/receiving a variable, then the type of the payload is balanced (i.e., when a process outputs a channel with an "active" session, it is forced to delegate *both* end points). We call such processes (types and type environments) *strongly-balanced*. In fact, processes of the form (1) above are not strongly-balanced, and are (still) not tackled by our analysis.

We have seen that balanced types guarantee subject reduction and type safety. Still, a balanced type permits to type a input process that waits for an unbalanced variable, what is useless since the process cannot receive such variable from a balanced process. To refine our analysis on type derivation trees, since in the forthcoming developments we work in an untyped setting, we identify the class of *strongly-balanced* types, processes and contexts, noted sbal.

4 The Target Language: Decorated π-Calculus

We use this new language in the type-checking and deadlock resolution algorithm. We adopt a constructive approach: the algorithm takes a typing context and a process and while building the type derivation creates a new process in the target language, decorating the channels with the types used up until that point. In fact, it transforms linearly used session channels into linear channels that synchronise in the same order, guaranteeing absence of races, as one session channel is mapped into a tuple of linear channels.

When, during type checking, the algorithm detects a (possible) deadlock, it launches the deadlock release function on the decorated version of the original process. Since we deal with sequential threads locally, transferring information from the global typing context to the channels occurring in that thread, the algorithm is *compositional* and *linear* in the size of the input process.

Syntax and semantics of the decorated π-calculus. In this section we introduce a variant of the polyadic π-calculus [20] where channels are decorated with session

Syntax of decorated processes

σ, ρ	::=	Decoration Types	H, K	::=	Processes
	bool	boolean		$\overline{\phi}\langle\omega\rangle.H$	output
	S	end point		$\phi(y_{\sigma_1}, \ldots, y_{\sigma_n}).H$	input
	$\top$	top		$(\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H$	restriction
ϕ, φ	::=	Prefixes		$\mathsf{if}\,\omega\,\mathsf{then}\,H\,\mathsf{else}\,K$	conditional
	x_σ	decoration		$(H \mid K)$	composition
ω, ψ	::=	Values		$!H$	replication
	$x_{\sigma_1}, \ldots, x_{\sigma_n}$	tuple		$\mathbf{0}$	inaction
	$\mathsf{true}, \mathsf{false}$	constant			

Rules for reduction (extends Fig. 1)

$$\frac{S_1 = \overline{S_2} \qquad \tilde{\phi} = y_{\sigma_1}, \ldots, y_{\sigma_n} \qquad |\omega| = |\tilde{\phi}|}{(\overline{x_{S_1}}\langle\omega\rangle.H \mid x_{S_2}(\tilde{\phi}).K) \xrightarrow{x_{S_1}} (H \mid K[\omega/\tilde{\phi}])} \quad [\text{R-DCom}]$$

$$\frac{1 \leq i < j \leq n \qquad \sigma_i = S \qquad \sigma_j = \overline{S} \qquad H \xrightarrow{y_S} H'}{(\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H \xrightarrow{\tau} (\nu y_{\sigma_1}, \ldots, y_{\sigma_{i-1}}, y_{\sigma_{i+1}}, \ldots, y_{\sigma_{j-1}}, y_{\sigma_{j+1}}, \ldots y_{\sigma_n})H'} \quad [\text{R-DResL}]$$

Fig. 4. Target language

types. The algorithm projects typed processes into decorated processes, as we explain ahead. We use polyadic channels to map a channel of type $(?T.S, !T.\overline{S})$ into a tuple of the form $x_{?T.S}, x_S, \ldots, x_{!T.\overline{S}}, x_{\overline{S}}, \ldots, x_\top$, where $\top \stackrel{\text{def}}{=} (\mathsf{end}, \mathsf{end})$.

The set of decorated processes $\mathcal{H}$, ranged by H, K, is defined in Fig. 4 by adorning processes of Fig. 1 with end point types S. There is a small difference, that is that we will never use end to decorate channels, and rely on $\top$ to decorate channels carrying void capabilities. We define free and bound variables of a process, noted respectively $\mathrm{fv}(H)$ and $\mathrm{bv}(H)$, in terms of multisets, and count the occurrences of a same decorated variable by means of a function $\mathsf{occurs} : \mathcal{F} \to \mathcal{N}$, where we let $\mathcal{F}$ be the set of decorated variables, ranged by ϕ, φ. We indicate tuples of prefixes with $\tilde{\phi}$. We use "\" to remove all occurrences of an entry in a multiset. For space limitations, we omit all the details and note that, for instance, $\mathrm{bv}((\nu y_\sigma, \ldots, y_\rho)P) = \{y_\sigma, \ldots, y_\rho\} \uplus (\{y_\sigma, \ldots, y_\rho\} \Cap \mathrm{fv}(P)) \uplus \mathrm{bv}(P) = \mathrm{bv}(x_S(y_\sigma, \ldots, y_\rho).P)$. We assume that alpha-conversion preserve decorations, and define the usual rules for structural congruence, but for the axiom for the null process: $(\nu y_\top)\mathbf{0} \equiv \mathbf{0}$.

The main change to the π-calculus semantics [20] is the communication rule, [R-DCom]: two processes exchange a value only if the two end points of the channel are decorated with *dual types*; this is akin to the polarity-based communication [7], and can be easily implemented by pattern matching of decorations. Substitution of a prefix ϕ with a value ω of the same arity is noted as $[\omega/\phi]$:

whenever $\omega = x_{\sigma_1}, \ldots x_{\sigma_n}$ and $\tilde{\phi} = \phi_1, \ldots, \phi_n$, we write $P[\omega/\tilde{\phi}]$ to indicate the process $P[x_{\sigma_1}/\phi_1, \ldots, x_{\sigma_n}/\phi_n]$. As in Fig. 1, we record the prefix ϕ on the arrow, which is of help for practical purposes and has no semantic impact. We use η to range over ϕ and τ actions, and write $H \to K$ when the label is irrelevant. Rule [R-DResL] describes a reduction on a couple of dually decorated prefixes that are restricted, and its continuation where the two dual end point channels have been removed from the restriction declaration. This rule is meant to describe *linear* processes where a decoration S appears only once, as we will introduce below; for this very reason, the restriction can be removed after a synchronisation.

Sound decorations. Instead of relying on a type system to ensure safety, we exploit the decoration of variables to characterise processes that do not reach errors during the computation (henceforth called *sound* processes). This characterisation leads to a *static, syntax-directed* checking system. As we will show later, our algorithm converts well-typed processes into sound processes, as one would expect, so we do not need a static type system for decorated processes.

Sound processes H such that $z_{\mathsf{end}} \notin \mathrm{var}(H)$, for any z, are determined through four syntactic conditions. First, we mimic the type system in Fig. 1 and enforce send and receive of values of the expected types by using a *coherence* inference system. The system not only checks the consistency between subject and object types, but also guarantees balanced payload types. However, it is not equivalent to the type system, since sequential and linear behaviour are checked separately Second, we check that processes have valid decorations for the same variable, i.e., types must form a chain (enforcing the sequential behaviour prescribed by the session types). Third, we check that each channel decorated with a type S is used exactly once. Fourth, we check that the order of the exchanges prescribed by the decorations are preserved. We omit the formal definition of sound process and refer the reader to the technical report [19] for all details.

The main result of this section says that sound processes do not reach errors, which are processes containing two processes prefixed with the same variable that *do not synchronise.*

Theorem 4. *If H is a sound process and $H \Rightarrow K$, then K is not of the form:*

1. $(\nu\tilde{\varphi})(\overline{x_S}\langle\omega\rangle.K_1 \mid \overline{x_R}\langle\psi\rangle.K_2 \mid K_3)$, *for some x_S, x_R, ω and ψ*
2. $(\nu\tilde{\varphi})(x_S(\tilde{\phi}).K_1 \mid x_R(\tilde{\phi}).K_2 \mid K_3)$, *for some x_S and x_R*
3. $(\nu\tilde{\varphi})(\overline{x_S}\langle\omega\rangle.K_1 \mid x_R(\tilde{\phi}).K_2 \mid K_3)$, *for some x_S, x_R, ω such that $R \neq \overline{S}$*

In the decorated setting, the notion of resource self-holding deadlock freedom is quite intuitive: prefixes with dual decorations must run in parallel. To this aim, we define a notion of set[3] intersection modulo dual type decorations, noted $\sqcap$: $(\{x_S\} \cup A) \sqcap (\{x_{\overline{S}}\} \cup B) = \{x_S, x_{\overline{S}}\} \cup (A \sqcap B)$, $x_S \sqcap x_R = \emptyset$ if $R \neq \overline{S}$, $x_S \sqcap y_R = \emptyset$ if $x \neq y$, and extend the definition of sequential variables as expected, e.g. $\mathrm{sv}(H \mid K) = \mathrm{sv}(H) \cup \mathrm{sv}(K) \backslash (\mathrm{sv}(H) \sqcap \mathrm{sv}(K))$.

[3] The subject variables of a sound process is indeed a set.

Definition 4. *A sound process H is* Resource Self-Holding Deadlock-Free *(or* RSHDF *) if* $\mathrm{sv}(H) = \emptyset$.

Theorem 5. *If H is* RSHDF *and $H \to K$ then K is* RSHDF .

Canonical representation In this section we show that (strongly-balanced) typed π-calculus processes have a canonical representation in decorated π-calculus, and that this representation both preserves the operational semantics and is sound.

First, we formalise trough function $\mathtt{dec}$ the projection of a π-calculus variable having a strongly-balanced type T, or type bool, into a tuple of decorated variables. This assumption is needed to ensure that all type decorations generated by the projection are sound (cf. Theorem 7). The formal definition of strongly-balanced types and processes is in [19].

$$\mathtt{dec}(y, (?T.S, !T.\overline{S})) = \{y_{?T.S}, y_{!T.\overline{S}}\} \Cup \mathtt{dec}(y, (S, \overline{S})) \quad \mathtt{dec}(y, \mathsf{bool}) = y_{\mathsf{bool}}$$
$$\mathtt{dec}(y, (!T.S, ?T.\overline{S})) = \{y_{!T.S}, y_{?T.\overline{S}}\} \Cup \mathtt{dec}(y, (S, \overline{S})) \quad \mathtt{dec}(y, (\mathsf{end}, \mathsf{end})) = y_{\top}$$

Definition 5. *Let $\Gamma \vdash Q$ with Γ and Q strongly-balanced. The canonical representation of Q w.r.t. Γ, noted $\mathtt{dec}(\Gamma, Q)$, is obtained by*

1. $\mathtt{dec}(x(y).P) = x_{?T.S}(\mathtt{dec}(y,T)).\mathtt{dec}(\Gamma', P)$ *whenever* $\Gamma(x) = (?T.S, !T.\overline{S})$ *or* $\Gamma(x) = (!T.\overline{S}, ?T.S)$, *and* $\Gamma' \vdash P$ *is a sub-tree of* $\Gamma \vdash x(y).P$;
2. $\mathtt{dec}(\overline{x}\langle y\rangle.P) = x_{!T.S}(\mathtt{dec}(y,T)).\mathtt{dec}(\Gamma', P)$ *whenever* $\Gamma(x) = (!T.S, ?T.\overline{S})$ *or* $\Gamma(x) = (?T.\overline{S}, !T.S)$, *and* $\Gamma' \vdash P$ *is a sub-tree of* $\Gamma \vdash \overline{x}\langle y\rangle.P$;
3. $\mathtt{dec}((\nu x\colon T)P) = (\nu\, \mathtt{dec}(x,T))\mathtt{dec}((\Gamma, x\colon T), P)$.

The remaining cases are homomorphic.

Theorem 6 (Operational Correspondence). *Let $\Gamma \vdash P$ with Γ and P strongly-balanced.*

1. *if $P \to P'$ and $\Gamma' \vdash P'$ then $\mathtt{dec}(\Gamma, P) \to \mathtt{dec}(\Gamma', P')$*
2. *if $\mathtt{dec}(\Gamma, P) \to H$ then there are Γ', P' such that $P \to P'$ and $H = \mathtt{dec}(\Gamma', P')$*

Theorem 7 (Soundness). *If $\Gamma \vdash P$ with Γ and P strongly-balanced, then $\mathtt{dec}(\Gamma, P)$ is sound.*

5 Deadlock Resolution Algorithm

We finally present the type checking and disentangling algorithm that releases deadlocks from typed processes through a process transformation. This algorithm is implemented using an inductive function that projects couples in $\mathcal{G} \times \mathcal{P}$ of Sect. 2 into decorated processes in $\mathcal{H}$ of Sect. 4. For clarity, the implementation of this function is presented by means of pattern analysis rules: we note that the algorithmic rules do not rely on type and context split, which is inherently non-deterministic.

Our procedure resolves multiple, nested deadlocks, possibly on the same channel. It works in one linear pass (when analysing a sequential process) and it

Top-level call

$$\frac{\mathsf{sbal}(\Gamma) \qquad \Gamma; (\Gamma\lfloor) \vdash_{\mathsf{A}} P \triangleright \Gamma_1; \Delta_1; H \qquad \mathsf{term}(\Gamma_1) \qquad \mathsf{term}(\Delta_1)}{\Gamma \Vdash_{\mathsf{A}} P \triangleright H}$$

Patterns for variables

$$\frac{T = (S_1, S_2)}{\Gamma, x\colon T \vdash_{\mathsf{A}} x\colon T \triangleright \Gamma, x\colon \top} \qquad \frac{T = (S_1, S_2)}{\Gamma, x\colon T \vdash_{\mathsf{A}} x\colon \top \triangleright \Gamma, x\colon T} \qquad \text{[A-Session],[A-Top]}$$

Patterns for output processes (excerpt)

$$\frac{\Gamma \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_1 \qquad \Gamma_1, x\colon (S, \mathsf{end}); \Delta_1, x\colon (\mathsf{start}, \overline{!T.S}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon \top; \Delta_2, x\colon (\mathsf{start}, R); H \qquad (*)}{\Gamma, x\colon (!T.S, \overline{!T.S}); \Delta_1, x : \bot \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_2, x\colon (\mathsf{end}, R); \Delta_2, x\colon \bot; K} \text{[A-OutInit-L]}$$

$$\frac{\Gamma \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_1 \qquad \Gamma_1, x\colon (S, \mathsf{end}); \Delta_1, x\colon (\mathsf{start}, R) \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad (**)}{\Gamma, x\colon (!T.S, \mathsf{end}); \Delta_1, x\colon (\mathsf{start}, R) \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_2; \Delta_2; K} \text{[A-OutEnv-L]}$$

$$\frac{\Gamma \vdash_{\mathsf{A}} y\colon T \triangleright \Gamma_1 \qquad \Gamma_1, x\colon (\mathsf{end}, R); \Delta_1, x\colon (S, \mathsf{start}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad (***)}{\Gamma, x\colon (\mathsf{end}, R); \Delta_1, x\colon (!T.S, \mathsf{start}) \vdash_{\mathsf{A}} \overline{x}\langle y\rangle.P \triangleright \Gamma_2; \Delta_2; K} \text{[A-OutProj-L]}$$

$(*)$ if $R = \overline{!T.S}$ then $K := \overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H$ else
if $R = \mathsf{end}$ then $K := [\![\overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H]\!]^{\mathsf{false}}$ else raise *fail*

$(**)$ if $\Delta_2(x) = (\mathsf{start}, R)$ then $K := \overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H$ else $K := [\![\overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H]\!]^{\mathsf{false}}$

$(***)$ if $\Gamma_2(x) = (\mathsf{end}, R)$ then $K := \overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H$ else $K := [\![\overline{x_{!T.S}}\langle \mathtt{dec}(y, T)\rangle.H]\!]^{\mathsf{false}}$

Fig. 5. Type checking function (part 1)

is compositional (with respect to parallel threads). We stress again that the class of deadlocks we disentangle is restricted to the *sequential* use of both end points of a channel in a given thread. Moreover, we consider herein only finite sessions (actually, we enforce total consumption of a session type when type-checking).

The top-level call of the algorithm, defined in Fig. 5, has the form $\Gamma \Vdash_{\mathsf{A}} P \triangleright H$, meaning that given in input a strongly-balanced environment Γ (cf. sbal) and a π-calculus process P, when the call is successful a decorated process H is returned in output; this implies that $\Gamma \vdash P$ (cf. Fig. 3), as we will show. In the rest of the presentation, let the $\triangleright$ symbol be the separator between the input (on the left) and the output (on the right) of the function. Note that the strongly-balanced hypothesis crucially permits to obtain deterministic and exhaustive pattern matching, as each free and bound variable is matched by an init pattern with a channel type formed by dual endpoints: balanced environments do not enforce this invariant for bound variables. However, we may need to use backtracking (see below). Strongly-balanced (decoration) types also ease the program transformation and the proof of its soundness. The top level call $\Vdash_{\mathsf{A}}$ makes use of

Patterns for input processes (excerpt)

$$\frac{\Gamma_1, x\colon (S, \mathsf{end}), y\colon T; \Delta_1, x\colon (\mathsf{start}, R), y\colon \bot \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon \top, y\colon \top; \Delta_2, x\colon (\mathsf{start}, R), y\colon A; H \qquad \mathsf{term}(A) \qquad (\sharp)}{\Gamma_1, x\colon (?T.S, R); \Delta_1, x : \bot \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2, x\colon (\mathsf{end}, R); \Delta_2, x\colon \bot; K} \text{[A-InInit-L]}$$

$$\frac{\Gamma_1, x\colon (S, \mathsf{end}), y\colon T; \Delta_1, x\colon (\mathsf{start}, R), y\colon \bot \vdash_{\mathsf{A}} P \triangleright \Gamma_2, y\colon \top; \Delta_2, y\colon A; H \qquad \mathsf{term}(A) \qquad (\sharp\sharp)}{\Gamma_1, x\colon (?T.S, \mathsf{end}); \Delta_1, x\colon (\mathsf{start}, R) \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2; \Delta_2; K} \text{[A-InEnv-L]}$$

$$\frac{\Gamma_1, x\colon (\mathsf{end}, R), y\colon T; \Delta_1, x\colon (S, \mathsf{start}), y\colon \bot \vdash_{\mathsf{A}} P \triangleright \Gamma_2, y\colon \top; \Delta_2, y\colon A; H \qquad \mathsf{term}(A) \qquad (\sharp\sharp\sharp)}{\Gamma_1, x\colon (\mathsf{end}, R); \Delta_1, x\colon (?T.S, \mathsf{start}) \vdash_{\mathsf{A}} x(y).P \triangleright \Gamma_2; \Delta_2; K} \text{[A-InProj-L]}$$

($\sharp$) if $R = \overline{?T.S}$ then $K := x_{?T.S}(\mathtt{dec}(y, T)).H$ else if $R = \mathsf{end}$ then `Snippet` else raise *fail*

($\sharp\sharp$) if $\Delta_2(x) = (\mathsf{start}, R)$ then $K := x_{?T.S}(\mathtt{dec}(y, T)).H$ else `Snippet`

($\sharp\sharp\sharp$) if $\Gamma_2(x) = (\mathsf{end}, R)$ then $K := x_{?T.S}(\mathtt{dec}(y, T)).H$ else `Snippet`

`Snippet` if $e = \mathsf{findValue}(\overline{x_{!T.S}}, (\Gamma_1, x\colon \top), H)$ && $e \neq 0$ then $K := [\![x_{?T.S}(\mathtt{dec}(y, T)).H]\!]^e$ else $K := x_{?T.S}(\mathtt{dec}(y, T)).H$

Fig. 6. Type checking function (part 2)

the function $\vdash_{\mathsf{A}}$, which is the core of the type-checking and disentangling[4] mechanism. The formal definition of function $\vdash_{\mathsf{A}}$ is in Figs. 5, 6, and 7, where the rules are assumed to be executed in the given order. We introduce left rules for $\vdash_{\mathsf{A}}$ where the matched type for the subject does appear in the left of a type (S_1, S_2); the right rules follow the same schema. The inner call $\vdash_{\mathsf{A}}$ is a function with the following signature: $\vdash_{\mathsf{A}} : \mathcal{G} \times \mathcal{D} \times \mathcal{P} \to \mathcal{G} \times \mathcal{D} \times \mathcal{H}$. The set $\mathcal{D}$ contains *projections* Δ mapping variables to types $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T} \cup \{(S, \mathsf{start}), (\mathsf{start}, S), (\mathsf{start}, \mathsf{start})\}$; start is a new end-point type which plays a role "dual" to end. We use $\top$ as a short for $(\mathsf{end}, \mathsf{end})$, and $\bot$ as a short for $(\mathsf{start}, \mathsf{start})$. At the bootstrap, a projection Δ is generated from a type environment Γ by means of a *casting* function, noted $\lfloor\,\rfloor$, which maps all types (S_1, S_2) in range(Γ) to type $(\mathsf{start}, \mathsf{start})$, and is idempotent over type bool. The top-level call $\Vdash_{\mathsf{A}}$ is then successful whenever the inner call $\vdash_{\mathsf{A}}$ returns Γ and Δ such that both are terminated, where we let any combination of end and start to form a terminated type.

Projections are used in $\vdash_{\mathsf{A}}$ to detect whether two end points of a same session are used sequentially, rather than in parallel. That is, in a projection a variable starts with type $(\mathsf{start}, \mathsf{start})$ and then it can possibly have assigned a type of the form (S, start) or (start, S), meaning that one of the two end points have been used sequentially. If at the end of the call the variable has a type of the form $(\mathsf{end}, \mathsf{start})$ or $(\mathsf{start}, \mathsf{end})$, we know that type S have been consumed, and we launch our program transformation by signalling where the deadlock may occur. While useful, projections are source of (light) non-determinism; for instance during type checking we may have Γ, Δ and x such that $\Gamma(x) = (?T_1.S_1, \mathsf{end})$

[4] When convenient, we will say "type disentangle" to mean resolution of a (typed) wait-for sequential deadlock.

Patterns for processes (excerpt)

$$\frac{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad \Gamma_2; \Delta_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_3; \Delta_3; K}{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_3; \Delta_3; H \mid K} \quad [\text{A-Par}]$$

$$\frac{\mathsf{sbal}(T) \qquad \Gamma_1, x\colon T; \Delta_1, x\colon \top \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x\colon \bot; \Delta_2, x\colon A; H \qquad \mathsf{term}(A)}{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} (\nu x\colon T)P \triangleright \Gamma_2; \Delta_2; (\nu \mathsf{dec}(x, T))H} \quad [\text{A-Res}]$$

$$\frac{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} y\colon \mathsf{bool} \triangleright \Gamma_2; \Delta_2 \qquad \Gamma_2; \Delta_2 \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad \Gamma_2; \Delta_2 \vdash_{\mathsf{A}} Q \triangleright \Gamma_2; \Delta_2; K}{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} \mathsf{if}\, y\, \mathsf{then}\, P\, \mathsf{else}\, Q \triangleright \Gamma_2; \Delta_2; \mathsf{if}\, y_{\mathsf{bool}}\, \mathsf{then}\, H\, \mathsf{else}\, K} \quad [\text{A-If}]$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{A}} P \triangleright \Gamma; \Delta; H}{\Gamma; \Delta \vdash_{\mathsf{A}} !P \triangleright \Gamma; \Delta; !H} \qquad \Gamma; \Delta \vdash_{\mathsf{A}} \mathbf{0} \triangleright \Gamma; \Delta; \mathbf{0} \quad [\text{A-Repl}],[\text{A-Inact}]$$

Fig. 7. Type checking function (part 3)

and $\Delta(x) = (\mathsf{start}, ?T_2.S_2)$. In this case we use backtracking, and first try to use $(?T_1.S_1, \mathsf{end})$, and then, if an exception is raised, try to use $(\mathsf{start}, ?T_2.S_2)$.

We can now analyse the patterns of function $\vdash_{\mathsf{A}}$. The patterns for variables have the form $\Gamma_1 \vdash_{\mathsf{A}} x : T \triangleright \Gamma_2$ where Γ_1, x and T are respectively a context, a variable and a type received in input, and Γ_2 is a context returned in output. The patterns for processes $\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} P_1 \triangleright \Gamma_2; \Delta_2; H_2$ follow in the same figure. For each input and output there are six rules: three matching the end point type on the left and three matching the end point type on the right. Consider one of the (six) rules for output, rule [A-OutInit-l]. The rule describes the pattern matched by the identified first, second and third parameter; the body invokes type-checking of variable y at the expected type by passing context Γ taken from the first parameter, and obtains as result Γ_1; a recursive call on the continuation is then invoked by "split" the continuation type of x in the context — $(S, \overline{!T.S})$ — between the context and the projection. To enforce termination of sessions, we check that the type of x in the return environment is $\top$. To see if x is deadlocked in P, we check the type (start, R) of x in the return projection: if R is different from $\overline{!T.S}$, then it has been used, and we invoke the disentangling function $[\![\cdot]\!]$ (cf. Fig. 8) passing as arguments the decorated process $\overline{x_{!T.S}}\langle \mathsf{dec}(y, T)\rangle.H$, where H is the return process, and the boolean constant false, which, in this case, is ignored: this second parameter will be used in the clauses for input. We can now read the side condition $(*)$ and understand the result forwarded in output: $K := \overline{x_{!T.S}}\langle \mathsf{dec}(y, T)\rangle.H$ when $R = \overline{!T.S}$, and $K := [\![\overline{x_{!T.S}}\langle \mathsf{dec}(y, T)\rangle.H]\!]^{\mathsf{false}}$ when $R = \mathsf{end}$. Note the failure when $R \neq \overline{!T.S}, \neq \mathsf{end}$, meaning that $\overline{!T.S}$ is partially consumed.

Pattern [A-OutEnv-l] is matched when the environment assigns to the output channel x a type of the form $(!T.S, \mathsf{end})$. The right end point type is equal to end since the channel has been used before (in input or output): in fact the type of x in the projection is (start, R), which is different from $\bot$. We type check the variable and launch the call for the continuation by passing the

Type disentangling encoding $[\![\cdot]\!] \colon \mathcal{H} \times \mathcal{V} \to \mathcal{H}$
($R = !T.S$, $I^T \stackrel{\text{def}}{=} ?T.\mathsf{end}$, $O^T = !T.\mathsf{end}$, y_T and z_T defined accordingly)

$$[\![\overline{x_R}\langle\omega\rangle.H]\!]^{\psi} \stackrel{\text{def}}{=} (\nu r_{O^T}, r_{I^T}, r_\top) \qquad (*)$$
$$(\overline{x_R}\langle\omega\rangle.\langle\!\langle H\rangle\!\rangle^{x_R}_{r_{I^T}} \mid x_{\overline{R}}(y_T).\overline{r_{O^T}}\langle y_T\rangle)$$

$$[\![x_{\overline{R}}(y_U).H]\!]^{\psi} \stackrel{\text{def}}{=} (\nu r_{O^T}, r_{I^T}, r_\top)(\nu m_{O^\top}, m_{I^\top}, m_\top) \qquad (**)$$
$$((x_{\overline{R}}(z_T).\overline{r_{O^T}}\langle z_T\rangle \mid r_{I^T}(y_U).\langle\!\langle H\rangle\!\rangle^{x_R}_{m_{O^\top}} \mid \overline{x_R}\langle\psi\rangle.m_{I^\top}()))$$

$(*)$ $\{r_{O^T}, r_{I^T}, r_\top, y_T\} \cap \mathrm{fv}(H) = \emptyset$ $\quad (**)$ $\{r_{O^T}, r_{I^T}, r_\top, m_{O^\top}, m_{I^\top}, m_\top, z_T\} \cap \mathrm{fv}(H) = \emptyset$

Auxiliary function for processes, $\langle\!\langle\cdot\rangle\!\rangle \colon \mathcal{H} \times \mathcal{F} \times \mathcal{F} \to \mathcal{H}$

$$\langle\!\langle x_{\overline{R}}(y_U).H\rangle\!\rangle^{x_{\overline{R}}}_{\varphi} = \varphi(y_U).H$$
$$\langle\!\langle \phi(y_U).H\rangle\!\rangle^{\psi}_{\varphi} = \phi(y_U).\langle\!\langle H\rangle\!\rangle^{\psi}_{\varphi} \qquad \phi \neq \psi$$
$$\langle\!\langle \overline{x_R}\langle\omega\rangle.H\rangle\!\rangle^{x_R}_{\varphi} = \overline{\varphi}\langle\rangle.H$$
$$\langle\!\langle \overline{\phi}\langle\omega\rangle.H\rangle\!\rangle^{\psi}_{\varphi} = \overline{\phi}\langle\omega\rangle.\langle\!\langle H\rangle\!\rangle^{\psi}_{\varphi} \qquad \phi \neq \psi$$

The remaining cases are homomorphic

Fig. 8. Transformation of decorated processes

entry $x \colon (!T.S, \mathsf{end})$ for the environment and by forwarding the same projection received in input. Condition $(**)$ is similarly to condition $(*)$ of [A-OutInit-L] and permit to check the shape of the return projection Δ in order to launch the code for disentangling: if $\Delta(x)$ is unchanged then we return the decorated process $\overline{x_{!T.S}}\langle\mathsf{dec}(y,T)\rangle.H$, otherwise we invoke disentangling on $\overline{x_{!T.S}}\langle\mathsf{dec}(y,T)\rangle.H$.

Pattern [A-OutProj-L] is matched when the type of the output channel x in the projection is of the form $(!T.S, \mathsf{start})$. In this case we invoke type checking for the continuation (after contrasting the variable) by passing the same environment received in input and by passing the entry $x \colon (S, \mathsf{start})$ for the projection. Dually to [A-OutEnv-L], in $(***)$ we control the return environment Γ in order to launch disentangling: if $\Gamma(x)$ is unchanged then we forward in output $\overline{x_{!T.S}}\langle\mathsf{dec}(y,T)\rangle.H$, otherwise we return $[\![\overline{x_{!T.S}}\langle\mathsf{dec}(y,T)\rangle.H]\!]^{\mathsf{false}}$.

The rules for input, [A-InInit-L], [A-InEnv-L] and [A-InProj-L], follow in Fig. 6 and are analogous respectively to [A-OutInit-L], [A-OutEnv-L] and [A-OutProj-L] in Fig. 5. The main differences are:

(a) there is no variable to type-check;
(b) in the call for the continuation the variable y bound by the input is added to the context at the payload type of the channel, and to the projection at type $\bot$; the type of y must be terminated in both the return context

and environment (cf. condition $\mathsf{term}(A)$), to enforce a linear discipline for y whenever its type is different from bool and $\top$;

(c) function $[\![\cdot]\!]$ in Fig. 8 is invoked *after checking* that the value ω sent over the sequential output corresponding to the input prefix satisfies certain conditions (cf. `Snippet`). This is implemented through a linear scan function $\mathsf{findValue}$ (see below) which, when successful, returns a value different from 0. When successful, we invoke function $[\![\cdot]\!]$ by passing as arguments $x_{?T.S}(\mathtt{dec}(y,T)).H$ and ω, otherwise, we return $x_{?T.S}(\mathtt{dec}(y,T)).H$.

Function $\mathsf{findValue}\colon \mathcal{F}\times\mathcal{G}\times\mathcal{H}\to\mathcal{V}\cup\{0\}$ takes in input a prefix $x_S\in\mathcal{F}$, an environment $\Gamma\in\mathcal{G}$, and a decorated process $H\in\mathcal{H}$, and scans the structure of H to find the value ω sent over x_S. The function returns ω whenever ω is a boolean value, or is equal to the tuple $x_{\sigma_1},\ldots,x_{\sigma_n}$, for some $x\in\mathrm{dom}(\Gamma)$, and 0 otherwise. See the technical report [19] for the formal definition.

In Fig. 7 we have rules for compositional processes. The interesting rule is the one for parallel composition, [A-Par]. The first call on the left returns a triple (Γ_2,Δ_2,H), where Γ_2 and Δ_2 are obtained by setting to end the session end points used in P, and H is obtained by disentangling (the decoration of) P, through function $[\![\cdot]\!]$. The second call on the right uses the return context Γ_2 and the projection Δ_2 to generate the triple (Γ_3,Δ_3,K), where K is obtained by disentangling Q, using the same schema. Note that the deadlocks of P and Q are fixed compositionally: we detect whether P is deadlocked *before* analysing Q, and return the triple $(\Gamma_3,\Delta_3,H\mid K)$.

Program Transformation. The encoding $[\![\cdot]\!]$ in Fig. 8 maps decorated input and output processes in $\mathcal{H}$ into decorated processes in $\mathcal{H}$, given a parameter in $\mathcal{V}$. The partial operation $[\![\cdot]\!]$ is called only with prefixed arguments: when invoked, it disentangles the first prefix encountered. To this aim, it uses the auxiliary total function $\langle\!\langle\cdot\rangle\!\rangle$ which takes a decorated process and two prefixes and returns a process. In the *output first* case of $[\![\cdot]\!]$ (first line) we rely on a fresh (triple of) forwarder(s) r to carry the result to be received by the input prefix of x, now put in parallel; the deadlocked input occurrence of x is renamed to r by $\langle\!\langle\cdot\rangle\!\rangle$. Note that we ignore the ψ parameter; it is useful only in the case below. The *input first* case of $[\![\cdot]\!]$ (second line) follows a similar idea but is more elaborate, because of variable binding; in this case we need both a (triple of) forwarder(s) r and a (triple of) semaphore(s) m, to preserve the order of exchanges: the call $\langle\!\langle\cdot\rangle\!\rangle$ renames the deadlocked output occurrence of x in H with m, while the output on x is put in parallel by *instantiating* the tuples of values to be sent with the actual parameter of $[\![\cdot]\!]$, that is ψ. As introduced, this parameter is found (before invoking $[\![\cdot]\!]$) trough function $\mathsf{findValue}$: the function searches for the occurrence of an output prefix in a decorated process and returns the values sent in output, when these are a boolean constant or a tuple of free prefixes.

Results. The first result guarantees that the algorithm succeeds only when type-checking succeeds, that is when the process is accepted by the system in Fig. 3. The construction of the proof of the theorem is similar to the one in [23], as the process returned in output by $\Vdash_{\mathsf{A}}$ is ignored.

Theorem 8 (Typability). *If* $\Gamma \Vdash_{\mathsf{A}} P \triangleright H$ *then* $\Gamma \vdash P$.

The second result is the main one of the paper: it ensures that the process returned by the algorithm is sound and *RSHDF* .

Theorem 9 (Deadlock Freedom). *If* $\Gamma \Vdash_{A} P \triangleright H$ *then* H *is* RSHDF .

We have a stronger result for the sequential variables of P: such variables run in parallel *at the same level* in the process returned by the algorithm.

Theorem 10 (Mismatch Freedom). *If* $\Gamma \Vdash_{A} P \triangleright H$ *and* $x \in \mathrm{sv}(P)$*, then for all* $x_s \in \mathrm{var}(H)$*:* $H \Longrightarrow K_{x_s}$ *with* $K_{x_s} \equiv (\nu\tilde{\varphi})(\overline{x_S}\langle\omega\rangle.K_1 \mid x_{\overline{S}}(\tilde{\phi}).K_2 \mid K_3)$.

6 Conclusions

We propose a new approach to tackle an old dilemma: can we do something to assist the programmer instead of simply reject code that does not type check? Founding on the π-calculus [1] and on a recent formulation of session types [9,10], we devised a type-checking algorithm that, when finds a particular form of deadlocks, which we refer to as resource self-holding deadlocks, automatically generates new type safe deadlock free code that mimics the original process intended behaviour as described by session types. We assessed the feasibility of our approach by implementing the algorithm (the code is available online, see [19]) and by analysing several examples of self-holding deadlocks. We believe that our approach is interesting since it can be used to release deadlocks in systems based on session and linear types, e.g. [3,7,11], which are represented by the type system of [10]. Moreover, we ensure deadlock freedom for well-typed processes not by restricting the set of typable processes, but by "fixing" those that exhibit the problem. If adapted to session based type systems of high-level languages, it may be an useful tool to assist the programmer in the software developing process, by (automatically) repairing program errors that can lead to runtime deadlocks. Our long-term goal is deadlock resolution for untyped processes, leaving the session type construction as a blackbox: the programmer writes the code; the algorithm infers the types, resolves the deadlocks, and provides error-free code.

For what concerns the limitations of our approach, we note that our notion of deadlock seems to be a specific instance of resource holding or *Hold and Wait* deadlocks [17,18], which is identified by considering resources (interpreted as π-calculus channels) blocked by the *same* thread; this notion is thus insensitive to the presence of cycles in waiting/releasing a resource. Specifically, there are four unmanaged classes of processes that we want to deal with: (1) we do not tackle processes of the form $a(x).\overline{x}\langle\mathsf{true}\rangle.b(z) \mid \overline{a}\langle b\rangle$ (which reduces in one step to the basic example $\overline{b}\langle\mathsf{true}\rangle.b(z)$), because the type of x is not balanced, which follows from a not having a strongly-balanced type; (2) we do not tackle processes like $a(x).c(z).\overline{a}\langle z\rangle.(\overline{x}\langle\mathsf{true}\rangle \mid x(y)) \mid \overline{c}\langle b\rangle$, because the actual object of the output on a is bound; (3) we do not tackle branching processes of the form $a(x).\mathsf{if}\, x\, \mathsf{then}\, \overline{a}\langle\mathsf{true}\rangle\, \mathsf{else}\, \overline{a}\langle\mathsf{false}\rangle$, i.e. they do not (algorithmically) type check,

because we cannot resolve the non-determinism caused by the test; (4) we do not tackle processes with circular deadlocks like $a(x).\overline{b}\langle\mathsf{true}\rangle \mid b(y).\overline{a}\langle\mathsf{false}\rangle$. To solve (1) and (2), we plan to enhance function findValue and collect a series of constraints of the form $x = v$, (when possible) meaning that the (bound) variable x should be instantiated with v, *before* executing the algorithm. We can then pass as further parameter the constraints to be instantiated, e.g. (1) $x = b$, and (2) $z = b$, and re-use the pattern rules presented in this paper.

Issue (3) could be solved by considering the π-calculus with a non-deterministic choice operator, or by devising a communication protocol that implements a similar behaviour (cf. [24]), to transform the blocked processes by putting in parallel the choice $\overline{a}\langle\mathsf{true}\rangle + \overline{a}\langle\mathsf{false}\rangle$. The issue (4) seems orthogonal to our approach, and would require techniques to detect dependencies and circularities in message passing, similarly to many recent works (e.g. [12]). We leave this for future work, as well as a behavioural theory to relate the source and the resulting process of our tool.

Acknowledgments. This work is partially supported by the Portuguese Fundação para a Ciência e a Tecnologia via project "CITI/FCT/UNL 2011-2012" — grant PEst-OE/EEI/UI0527/2011 and project "Liveness, statically" — grant PTDC/EIA-CCO/117513/2010, and by the COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY). We would like to thank Adrian Francalanza for fruitful discussions and illuminating examples, and the anonymous reviewers for their careful reading and constructive criticisms.

References

1. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Inf. Comput. **100**(1), 1–77 (1992)
2. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
3. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
4. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
5. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: an overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
6. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. **207**(5), 595–641 (2009)
7. Gay, S.J., Hole, M.J.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2/3), 191–225 (2005)
8. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication. In: SecReT. ENTCS, vol. 171(4), pp. 73–93 (2007)

9. Giunti, M., Vasconcelos, V.T.: A linear account of session types in the pi calculus. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 432–446. Springer, Heidelberg (2010) AQ1
10. Giunti, M., Vasconcelos, V.T.: Linearity, session types and the pi calculus. Math. Struct. Comput. Sci. (2013, in press)
11. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999)
12. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
13. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
14. Caires, L., Vieira, H.T.: Conversation types. Theoret. Comput. Sci. **411**(51–52), 4399–4440 (2010)
15. Wadler, P.: Propositions as sessions. In: ICFP, pp. 273–286. ACM (2012)
16. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274. ACM Press (2013)
17. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Comput. Surv. **3**(2), 67–78 (1971)
18. Knapp, E.: Deadlock detection in distributed databases. ACM Comput. Surv. **19**(4), 303–328 (1987)
19. Giunti, M.: (LockRes: a deadlock resolver for the pi calculus) SML/NJ prototype of the algorithm. http://ctp.di.fct.unl.pt/~mgiunti/lockres. The web page contains the SML/NJ prototype of thealgorithm presented in this paper and the technical report.
20. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, New York (1999)
21. Barendregt, H.: The Lambda Calculus - Its Syntax and Semantics, 1st edn. North-Holland, Amsterdam (1981, revised 1984)
22. Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, pp. 3–44. MIT Press, Cambridge (2005)
23. Giunti, M.: Algorithmic type checking for a pi-calculus with name matching and session types. J. Logic Algebraic Program. **82**(8), 263–281 (2013)
24. Nestmann, U., Pierce, B.C.: Decoding choice encodings. Inf. Comput. **163**(1), 1–59 (2000)

Author Queries

Chapter 9

Query Refs.	Details Required	Author's response
AQ1	Please provide volume number and page number for Ref. [10].	