Towards static deadlock resolution in the π-calculus *

Marco Giunti and António Ravara
CITI and DI-FCT, Universidade Nova de Lisboa, Portugal

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 – while maintaining a good expressiveness that allows to represent the standard π-calculus and several typing disciplines. More advanced type systems, enforcing stronger 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: we develop 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 allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol [19, 20, 28]. Introduced for a dialect of the π-calculus of Milner et al. [25], the concept has been transferred to different realms, including functional [15, 26, 27, 29, 30] and object-oriented programming [6, 10, 11, 13, 21], and even to operating systems [12].[1] The most interesting 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, and consists in 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 [20].

While many session typing systems require a means to distinguish the two ends of a session channel in order to preserve type soundness [10, 14, 33], recently the first author and Vasconcelos have developed a session typing system [17, 18] that works 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 [14, 20] and linear [23] types can be embedded in the framework.

A drawback of most of these systems is accepting processes that exhibit various forms of deadlocks — although they guarantee type safety, they do not guarantee deadlock-freedom. For that aim, several proposals appeared recently, guaranteeing progress by inspecting causality dependencies in the processes [3, 4, 5, 7, 31]. Not surprisingly, these systems reduce the set of typed processes, namely rejecting (as usual in static analysis, which is not complete) deadlock-free processes.

*Extended version of the submitted paper — the proofs of the main results are in the appendix.
[1] The interested reader may have look at a recent overview [9].

Aim of this work. 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., input and output on the same channel occur in sequence in a given thread, an istance of *Wait For* deadlocks [8, 22]), we devise an algorithm that produces a "fix" for this kind of deadlocked processes. This situation is easy to spot in a simple process, but it is not so obvious when the two co-actions (input and output) occur far away from each other in the code. Assisting the programmer in finding and solving these errors may lead to spare time when debugging. The synthetised code can be submitted to the programmer that decides if the "fix makes sense".

Working with the standard π-calculus equipped with the Giunti and Vasconcelos type system, we propose a compromise solution to the identified drawback: rather than require stronger conditions for the analysis and type *less* processes, we devise a procedure that detects synchronisation errors leading to *RSHDF* processes, while automatically generating type-safe, deadlock-free, code that mimics the process behaviour, as described by session types. The mechanism crucially relies on the help of types to infer this kind of sequentialisation errors made by the programmer in coding sessions. More specifically, our typing algorithm takes in input a type environment Γ and process P, decides whenever P behaves as prescribed by Γ, and if P is well-typed, although (resource self-holding) deadlocked, then it generates a new, error-free, process, obtained by "disentangling" P, i.e., inputs and outputs on the same channel are no longer in the same thread, and thus, the deadlock is released.

In short, following the behaviour of a process as specified by a session type environment, the algorithm uses a process transformation function that puts in parallel threaded sequences of input/output in the same channels. The resulting process faithfully represents the intended protocol of the original one (according to the session types), is type safe, and does not present the referred deadlocks on those channels.

This idea may be applied in session based systems developed to high-level languages, providing further support to the software development process. We tested our method, implementing the algorithm (in *Standard ML*), and analysing several (complex) examples [1].

Plan of the paper. In Section 2, we introduce a typed pi calculus where sessions obey to a linear discipline, and review the soundness and safety properties of the typing system. Then, in Section 3, we define the class of resource-holding and of deadlock-free processes, showing the latter class closed under reduction. In Section 4, we present an untyped pi calculus with decorated channels: as decorations are based on types, we do not rely on typing contexts and systems in order to identify safe processes. In Section 5, we devise a split-free type checking algorithm that projects typed processes into decorated processes. 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 the paper presenting the main results concerning the algorithm: soundness — if the algorithm accepts a process, then the process is typed by the split-based system of Section 2; and any process generated by the algorithm: (i) does not reach errors during the computation; (ii) is resource-holding deadlock-free.

2 The source language: π-calculus with session types

We present first 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 show a type checking derivation, and conclude stating 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*, which are variables and the boolean constants true

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

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

Duality and balance of session types

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

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

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

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

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

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

Figure 1: Typed π-calculus

and false, are ranged by v, w. 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.

The syntax of processes is standard (cf. [25]), but for restriction for which we use type annotation. The formal definition of the language is in Figure 1. Considering the usual notions of free and bound variables, α-conversion, as well as of substitution, cf. [24], 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 [2], 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 $(\nu y\colon T)P$ provides for create a variable y decorated with the type T; the occurrences of y in P are bound. Boolean values can be contrasted by means of 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 the process $(\nu\tilde{x}_1\colon T_1)\cdots(\nu\tilde{x}_n\colon T_n)P$, for $n \geq 0$.

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. The first rule in the second line permit to open the scope of a restriction: due to variable convention, y bound in $(\nu y\colon T)P$, cannot be free in Q. The second rule in the second line allows to remove a restriction, provided that the session type has been consumed. The rule in the third line allows to exchange the order of bindings.

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 [14, 23]); this is convenient, and has no semantic impact, allowing to represent the progression of type decorations in restricted processes. Let μ range over variables x, y and the symbol τ, which we assume reserved (not occurring in the syntax of processes). Moreover, let $\Rightarrow$ indicate the reflexive and transitive closure of $\overset{\mu}{\longrightarrow}$, whenever the labels are irrelevant.

Static semantics: type system. To represent type progression, we consider a next operator over types to unfold sessions, and extend the function to typing contexts by letting $\mathsf{next}(\Gamma, \tau) = \Gamma$, and $\mathsf{next}(\Gamma, x) = \Gamma_1, x\colon \mathsf{next}(T)$ whenever $\Gamma = \Gamma_1, x\colon T$. We make use of predicates on types and contexts, *balanced* and *terminated* (noted respectively bal and term). Balanced contexts are those of our interest, as they preserve subject-reduction: processes must send and receive values of the same type on both end points of a same channel. Balancing relies on the standard duality notion of session types; we let $\overline{S}$ be the dual of S. 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 $(\nu x\colon \mathsf{bool})P$. Termination holds for boolean and end types. A context Γ is balanced whenever $\mathsf{bal}(T)$ for all $\Gamma(x) = T$; Γ is terminated whenever $\mathsf{term}(T)$ for all $\Gamma(x) = T$.

Figure 2 contains a typing system with judgements of the form $\Gamma \vdash P$, where we assume that $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{bv}(P) \cap \mathrm{dom}(\Gamma) = \emptyset$. The type system uses a notion of type and context split (cf. Walker's chapter in Pierce's book [32]), noted $\circ$. Formally, split is a three-argument relation. We write $\Gamma_1 \circ \Gamma_2$ to refer to a type environment Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. We extend the operator to type environments having identical domain. We use infix notation and write $S \circ R$, $T_1 \circ T_2$, and $\Gamma_1 \circ \Gamma_2$ to indicate respectively the composition of the end points S, R, of types T_1, T_2, and of environments Γ_1, Γ_2, whenever defined.

The typing rules are inspired by the system of Giunti and Vasconcelos [17], and represent a subsystem of its recent re-formulation [18]. Rule [T-Var] is for typing values and requires the environment pruned from the typed entry to be terminated, so to enforce that linear assumptions are used. Rule [T-Bool] is for typing boolean constants. 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. Rule [T-In-L] types an input channel x by using the end point type $?T.S$ on the left of a type $(?T.S, R)$. If x is typed with $?T.S$, we know that the bound variable y is of type T, and we type P under the extra assumption $y\colon T$. Equally important is the fact that the continuation uses channel x at continuation type (S, R), that is, process $x(y).P$ uses channel x

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

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

Termination of session types and contexts

$$\mathsf{term}(\mathsf{bool}) \qquad \mathsf{term}((\mathsf{end}, \mathsf{end}))$$

$$\mathsf{term}(\emptyset) \qquad \mathsf{term}(\Gamma, x\colon T) \ :- \ \mathsf{term}(\Gamma), \mathsf{term}(T)$$

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

$$\frac{\Gamma, x\colon (S, R), y\colon T \vdash P}{\Gamma, x\colon (?T.S, R) \vdash x(y).P} \qquad \frac{\Gamma, x\colon (R, S), y\colon T \vdash P}{\Gamma, x\colon (R, ?T.S) \vdash x(y).P} \qquad \text{[T-In-l],[T-In-r]}$$

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

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

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

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

Figure 2: Type system

of type $(?T.S, R)$ whereas P must use the *same* channel this time of type (S, R). Rule [T-In-r] types an input x described by a session type $(S, ?T.R)$ by following the same mechanism. In the typing rules for output we account for sending a variable that can have a channel or a session type; in the latter case we have *delegation* of one or both ends of the session. To illustrate the delegation mechanism we describe the rule for typing an output with the left end point of a channel type, [T-Out-l]; the remaining rules follow a similar schema. Rule [T-Out-l] allows using an environment $\Gamma = \Gamma_1 \circ (\Gamma_2, x\colon (!T.S, R))$ to send a value v of type T on x and to continue as P, given that $\Gamma_1 \vdash v\colon T$, and $\Gamma_2, x\colon (S, R) \vdash P$. For instance if $T = (?T'.S, \mathsf{end})$ and $\Gamma(v) = (?T'.S, !T'.R)$ then v is both sent at type T and used of type $(\mathsf{end}, !T'.R)$ in the continuation P. Rule [T-Out-r] describes the same mechanism for an output channel typed with the right end point. Rule [T-If] types an if-then-else by requiring that the contrasted value has a boolean type. The environment that types the two branches of the if-then-else must be terminated, to simplify the algorithm's task of finding the sequential prefix to be disentangled; we will further discuss this issue in the conclusions. Rule [T-Res] is used to type a restriction, what is possible only if the decoration type is balanced. To type a replication with rule [T-Repl] we require the context to be terminated,

to rule out multiple uses of a linear type. The last two rules, [T-Par] and [T-Inact], permit to type respectively parallel composition and inaction.

Type checking derivation by example. Take a process whose behaviour consists in printing "*ok*" after two exchanges, and then continue as P_1.

$$P \stackrel{\text{def}}{=} x(y).\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle \mid \overline{x}\langle \mathsf{true}\rangle.x(z).\overline{printf}\langle z\rangle.P_1 \tag{1}$$

This process is typed by assigning to x a type like $T \stackrel{\text{def}}{=} (?\mathsf{bool}.!\mathsf{string}.\mathsf{end},\ !\mathsf{bool}.?\mathsf{string}.\mathsf{end})$, where type string is represented by $(\mathsf{end}, \mathsf{end})$, a type that conveys no i/o capabilities. To build a type derivaton for P, we split T into $S_L \stackrel{\text{def}}{=} (?\mathsf{bool}.!\mathsf{string}.\mathsf{end}, \mathsf{end})$ (on the left) and $S_R \stackrel{\text{def}}{=} (\mathsf{end}, !\mathsf{bool}.?\mathsf{string}.\mathsf{end})$ (on the right). A (simplified) formal derivation for the left thread follows, where $S'_L \stackrel{\text{def}}{=} !\mathsf{string}.\mathsf{end}$ and $\Gamma_L \stackrel{\text{def}}{=} \Gamma', x\colon (S_L, \mathsf{end})$ are respectively the left continuation type of x and the type environment for the left thread of P. We also assume that $\Gamma'(ok) = \mathsf{string}$, that Γ' does not contain session types in its range, and use a simplified version of the rule for if-then-else ([T-If]) that avoids using split.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma', x\colon (\mathsf{end},\mathsf{end}), y\colon \mathsf{bool} \vdash ok\colon \mathsf{string}}\,([\text{T-Var}]) \qquad \dfrac{}{\Gamma', x\colon (\mathsf{end},\mathsf{end}), y\colon \mathsf{bool} \vdash \mathbf{0}}\,([\text{T-Inact}])}{\Gamma', x\colon (S'_L,\mathsf{end}), y\colon \mathsf{bool} \vdash \overline{x}\langle ok\rangle}\,([\text{T-Out-L}])}{\Gamma', x\colon (S'_L,\mathsf{end}), y\colon \mathsf{bool} \vdash \mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle}\,([\text{T-If}])}{\Gamma_L \vdash x(y).\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle}\,([\text{T-In-L}])$$

By letting $\Gamma_R \stackrel{\text{def}}{=} \Gamma'', x\colon (\mathsf{end}, S_R)$ be the type environment for the right thread of P, we have a similar derivation: $\Gamma_R \vdash \overline{x}\langle \mathsf{true}\rangle.x(z).\overline{printf}\langle z\rangle.P_1$. We conclude by finding Γ such that both $\Gamma(x) = T$ and $\Gamma = \Gamma_L \circ \Gamma_R$: this let us apply the rule for parallel composition and obtain $\Gamma \vdash P$.

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 [16].

Theorem 2.1 (Subject reduction)**.** *Let Γ be balanced. If $\Gamma \vdash P$ and $P \Rightarrow P'$ then there is Γ' balanced such that $\Gamma' \vdash P'$.*

We now define erroneous processes. As explained above, our system filters processes of the form $\mathsf{if}\, x\, \mathsf{then}\, P\, \mathsf{else}\, Q$ that do not occur under an input prefix on x, or errors arising from problems in synchronisation, as in the process $\overline{x}\langle \mathsf{true}\rangle.P \mid x(y).y(z).P$, where true can be send through [R-Com] but the substitution is not defined. Other errors are 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 2.2 (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:*

1. *$P = \overline{x}\langle v\rangle.P_1 \mid \overline{x}\langle w\rangle.P_2$, or*
2. *$P = x(y).P_1 \mid x(z).P_2$.*

Lemma 2.3. *If $\Gamma \vdash P$ then P is not an error.*

Theorem 2.4 (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 Section 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 (still) 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 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 contain a resource self-holding deadlock, since no communication on a can occur, as the co-actions appear in sequence, instead of in parallel. The obvious "fix" is the process $(\nu a)\ (a(y) \mid \overline{a}\langle\mathsf{true}\rangle)$.

More intricate resource self-holding deadlocks Consider the following process.

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

This process is deadlocked, since it is typed by a balanced context that requires both end points of each communication to be used linearly, thus disallowing any further use of the channels to parallel threads.[2] When such a context is given to our algorithm (together with the process), the result is, simplifying now to help the presentation, the process below.

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

Our algorithm solves this type of deadlocks, but does not deal for now with delicate situations involving binders. Consider the tyable process

$$a(x).b(y).\overline{a}\langle y\rangle.(x() \mid \overline{x}\langle\rangle) \mid \overline{b}\langle c\rangle.$$

We cannot simply put the output on a in parallel, as the variable y would escape its scope. Thus, we prevent the resolution function to be launched, and thus the algorithm accepts the process without modifying it.

Resource self-holding deadlocks: formally. Consider the following auxiliary notions.

- a construct α of the form $\overline{x}\langle v\rangle$ or $x(y)$ is an *x-prefix*; a process of the form $\alpha.Q$, where α is an x-prefix, is an *x-process*.
- As usual, *contexts*, ranged by C, are processes containing a hole, noted '$\cdot$'.[3] We write $C[P]$ for the process obtained by replacing the occurrence of the hole in $C[\cdot]$ with the process P, and call contexts to both processes $C[\cdot]$ and type environments Γ.
- The *subject variables* of P, noted $\mathrm{subjv}(P)$, is the *multiset* with the occurrences of $x \in \mathrm{var}(P)$ such that $P = C[Q]$, for some x, Q, and C, with Q being an x-process. Likewise, the *object variables* of P, noted $\mathrm{objv}(P)$, is the multiset with the occurrences of $y \in \mathrm{var}(P)$ such that $P = C[\overline{x}\langle y\rangle.Q]$, for some x, Q, and C.

Let $\Cap$ be multiset intersection, $\Cup$ multiset union, $\Subset$ multiset inclusion, and $\setminus$ multiset difference.

[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.

[3] To define the syntax of contexts, simply add to the production of processes the case of the hole.

Definition 3.1 (Parallel and Sequential Variables)**.**

1. *The* parallel variables *of a process P, noted* $\mathrm{pv}(P)$*, is the following submultiset of* $\mathrm{subjv}(P)$*.*
   $\{x \in \mathrm{subjv}(P) : \exists C, Q_1, Q_2 \,.P \equiv C[Q_1 \mid Q_2] \text{ and } x \in \mathrm{subjv}(Q_1) \Cap \mathrm{subjv}(Q_2)\}$
2. *The* sequential variables *of a process P, noted* $\mathrm{sv}(P)$*, is* $\mathrm{sv}(P) \stackrel{def}{=} \mathrm{subjv}(P) \setminus \mathrm{pv}(P)$*.*

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. The definition below formalises this intuition, where we let $\{x, x, \dots\}$ be a countably infinite multiset of x.

Definition 3.2 (Resource Self-Holding Deadlock Freedom)**.** *A process P is* Resource Self-Holding Deadlock-Free (RSHDF)*, if* $\mathrm{subjv}(P) = \mathrm{pv}(P)$*.*

Lemma 3.3. *P is* RSHDF *if, and only if,* $\mathrm{sv}(P) = \emptyset$*.*

Example 3.4. *Consider the composition Q of the process P on page 6 (process 1) with a process Stdout; let $\{x, \mathit{printf}\} \Cap \mathrm{subjv}(P_1) = \emptyset$ and $\{x, \mathit{printf}\} \Cap \mathrm{subjv}(\mathit{Stdout}) = \emptyset$*

$$Q \stackrel{def}{=} x(y).\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle \mid \overline{x}\langle \mathsf{true}\rangle.x(z).\overline{\mathit{printf}}\langle z\rangle.P_1 \mid \mathit{printf}(z).\mathit{Stdout}$$

If $\mathrm{sv}(P_1 \mid \mathit{Stdout}) = \emptyset$*, then Q is* RSHDF *since*

$$\mathrm{subjv}(Q) = \{x, x, x, x, \mathit{printf}, \mathit{printf}\} \Cup \mathrm{subjv}(P_1) \Cup \mathrm{subjv}(\mathit{Stdout})$$
$$\mathrm{pv}(Q) = \{x, x, x, x, \mathit{printf}, \mathit{printf}\} \Cup \mathrm{pv}(P_1 \mid \mathit{Stdout})\ .$$

Whenever $\mathrm{sv}(P_1 \mid \mathit{Stdout}) = \emptyset$*, then P is* RSHDF *.*

To show that a process P is not *RSHDF* it suffices to find a channel x such that there is no way of presenting P as $C[Q_1 \mid Q_2]$ with x in both $\mathrm{subjv}(Q_1)$ and $\mathrm{subjv}(Q_2)$. To illustrate, assume $x \notin \mathrm{subjv}(\mathrm{Stdout})$ and take

$$P' \stackrel{\text{def}}{=} x(y).\overline{x}\langle \mathsf{true}\rangle.\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle.x(z).\overline{\mathit{printf}}\langle z\rangle.P_1 \mid \mathit{printf}(z).\mathrm{Stdout} \tag{2}$$

Since there is no such $C[\cdot]$, then P' is not *RSHDF*; it is however typable by a balanced context.

The algorithm, informally. To provide an intuition to our methodology, we illustrate the approach on process P' (above).

Because of the concurrent nature of types of the form (R, S) (where each type represents the communication behaviour of one end point of a given channel), the *same* environment Γ considered to type process 1 (in page 6) also accepts the process P'.

Our split-free type checking algorithm (presented in Section 5), based on functional patterns, detects that there is an anomaly, since the inner call for the sub-process

$$\overline{x}\langle \mathsf{true}\rangle.\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle.x(z).\overline{\mathit{printf}}\langle z\rangle.P_1\ ,$$

which receives in input the type (S'_L, S_R) for x (cf. Page 6), returns in output the type $(\mathsf{end}, \mathsf{end})$; this means that *both* end points of x have been consumed: that is, x is not left available at type (end, S_R) to the next thread (as in the call for type checking the continuation $\mathsf{if}\, y\, \mathsf{then}\, \overline{x}\langle ok\rangle$ of P). We fix this by generating new code for P' that mimics the behaviour described by its session type, which is, in principle, the desired behaviour of P.

The basic idea is the following: if while analysing a prefixed process (input or output), we find in a subprocess the corresponding end point (output or input), when there are no bindings involved we take it out and put it in parallel. Since we "chopped" a sequential process into a

parallel one, we need a "forwarder" to link the pieces. Thus, the type-assisted translation $[\![\cdot]\!]$ maps a typed sequential process into a parallel process, using a forwarder (r):[4]

$$[\![P']\!] \quad \stackrel{\text{def}}{=} \quad (\nu r)(x(u).\overline{r}\langle u\rangle \mid r(y).\mathsf{if}\, y\, \mathsf{then}\, [\![\overline{x}\langle ok\rangle.x(z).\overline{printf}\langle z\rangle.P_1]\!] \mid \overline{x}\langle \mathsf{true}\rangle)$$

Notice this works because the output on x carries a value (and it would also work with a free variable), but obviously the transformation cannot be applied so simply when the output carries bound channels, like in $x(y).z(w).\overline{x}\langle w\rangle$.

After some confluent reduction steps, the boolean value sent over the occurrence of x described by the right type of T (that is S_R) is assigned to the bound variable in the occurrence of x described by the left type of T (that is S_L):

$$[\![P']\!] \Rightarrow \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, [\![\overline{x}\langle ok\rangle.x(z).\overline{printf}\langle z\rangle.P_1]\!]$$

The last (confluent) steps assign to the bound variable in the occurrence of x described by its right continuation type the string sent in the occurrence of x described by its left continuation type:

$$[\![\overline{x}\langle ok\rangle.x(z).\overline{printf}\langle z\rangle.P_1]\!] \Rightarrow P_1[ok/z]$$

Results. The main result of this section is that the *RSHDF* property is closed under reduction (cf. Theorem 3.7). It is useful to analyse the shape of types appearing in the derivations 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 delegates *both* end points).

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, we identify the class of *semi-balanced* types T, noted $\mathsf{sbal}(T)$, of the form (S, end), (end, S), and $(S, \overline{S})$, where whenever $S = ?T.R$ or $S = !T.R$ type T *must be balanced or equal to* bool, and types (R, end), (end, R) and $(R, \overline{R})$ *must be semi-balanced*.

We first define the semi-balanced predicate over types. Semi-balanced contexts are those mapping variables to semi-balanced types; as we show later, a derivation starting with a semi-balanced context contains only semi-balanced types.

Definition 3.5. *A type T is semi-balanced, noted* $\mathsf{sbal}(T)$*, if*

$$\begin{array}{l}
\mathsf{sbal}(\mathsf{bool}).\\
\mathsf{sbal}(T) \text{ :- } \mathsf{bal}(T).\\
\mathsf{sbal}(\mathsf{end}, \mathsf{end}).\\
\mathsf{sbal}((?T.S, \mathsf{end})) \text{ :- } \mathsf{sbal}(T),\ \mathsf{sbal}((S, \mathsf{end})).\\
\mathsf{sbal}((?T.S, !T.R)) \text{ :- } R = \overline{S},\ \mathsf{sbal}(T),\ \mathsf{sbal}((S, R)).\\
\mathsf{sbal}((!T.S, \mathsf{end})) \text{ :- } \mathsf{sbal}(T),\ \mathsf{sbal}((S, \mathsf{end})).\\
\mathsf{sbal}((!T.S, ?T.R)) \text{ :- } R = \overline{S},\ \mathsf{sbal}(T),\ \mathsf{sbal}((S, R)).\\
\mathsf{sbal}((\mathsf{end}, !T.S)) \text{ :- } \mathsf{sbal}(T),\ \mathsf{sbal}((\mathsf{end}, S)).
\end{array}$$

With small modifications we inherit the following result from Theorem 2.1, and use it in the proof of Theorem 3.7 below, which guarantees the of closure of *RSHDF* under reduction.

Corollary 3.6. *If $\Gamma \vdash P$, Γ is semi-balanced, and $P \Rightarrow P'$, then there is a semi-balanced Γ' such that $\Gamma' \vdash P'$.*

Notice that semi-balanced contexts force a process to delegate both end points of a channel, when sending it on another channel.

Theorem 3.7 (RSHDF preservation)**.** *If $\Gamma \vdash P$ with Γ semi-balanced, P is RSHDF, and $P \Rightarrow P'$, then P' is RSHDF.*

[4]The translation (Figure 7) is slightly more complicated, using semaphores to impose, in the source and in the translated processes, the same order of communication.

4 The target language: decorated π-calculus

We use this new language in the type-checking and deadlock resolution algorithm. We take 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, guaranteing 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, transfering information from the global typing context to the channels occuring 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 [24] where channels are decorated with session types.

The algorithm projects typed processes into decorated processes, as we explain ahead. Let $\mathcal{F}$ be the set of decorated variables. The set of decorated processes $\mathcal{H}$, ranged by H, K, is defined in Figure 3 by adorning processes of Section 2. The decoration types that we consider, ranged by σ, ρ, are bool, the end point types S of Section 2, and type $\top$, which can be seen as a representation of type $(\mathsf{end}, \mathsf{end})$; in fact, we will never use end to decore channels, and rely on $\top$ to decorate channels carrying void capabilities, as we will explain below. Termination on types σ is defined in the expected way, considering $\top$ terminated, and tuples $x_{\sigma_1}, \ldots, x_{\sigma_n}$ (with $n \geq 1$) of distinct decorations for the same variable (i.e., $i \neq j$ implies $\sigma_i \neq \sigma_j$, for all $i, j \in 1, \ldots, n$).

Let the substitution of a prefix ϕ with a value ω of arity one in P be noted as $P[\omega/\phi]$. 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 $P[x_{\sigma_1}/\phi_1, \ldots, x_{\sigma_n}/\phi_n]$. It is convenient to define free and bound variables of a process, noted respectively $\mathrm{fv}(H)$ and $\mathrm{bv}(H)$, in terms of multisets of prefixes. We overload the notation and use "," to separate the entries of a multiset, and "\" to remove all occurrences of an entry in a multiset. We assume a function occurs that given a prefix φ and a multiset $\phi, \ldots, \psi$ returns the number of occurrences of the prefix in the multiset. Let $\mathrm{var}(H) \stackrel{\text{def}}{=} \mathrm{fv}(H) \uplus \mathrm{bv}(H)$, if $\mathrm{fv}(H) \Cap \mathrm{bv}(H) = \emptyset$ ($\mathrm{var}(H)$ is the disjoint union of $\mathrm{fv}(H)$ and $\mathrm{bv}(H)$). We abuse the notation and write $\phi \in \mathrm{var}(H)$ whenever $\mathsf{occurs}(\phi, \mathrm{var}(H)) > 0$, and $\phi \notin \mathrm{var}(H)$ otherwise. We often avoid the trailing '$\mathbf{0}$' and use $(\nu\tilde{\zeta})H$ as a short of $(\nu y_{\sigma_1}, \ldots, y_{\sigma_n}) \cdots (\nu z_{\rho_1}, \ldots, z_{\rho_m})H$, whenever the identity of the bound tuples is irrelevant. The main change to the π-calculus semantics [24] 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 [14], and can be easily implemented by pattern matching of decorations. As in Figure 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. We have two rules for processes reducing under restricted channels; in the left rule [R-DResL] the decorated variable x_S which labels the free reduction appears with an index i in the sequence of bindings under restriction, while its dual appears in position $j > i$; in the right rule (which we omit) the two indexes are exchanged. The remaining reduction rules are almost standard.

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.

First, we formalise trough function $\mathtt{tp}$ the projection of a π-calculus variable having balanced type T, or type bool, into a tuple of decorated variables.

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			

Free and bound variables (cases omitted are homomorphic)

$$\mathrm{fv}(\mathbf{0}) = \emptyset \qquad \mathrm{fv}(\overline{\phi}\langle\omega\rangle.H) = \{\phi\} \Cup \mathrm{fv}(H)\,, \text{ if } \omega = \mathsf{true}, \mathsf{false}$$
$$\mathrm{fv}(\overline{\phi}\langle y_{\sigma_1}, \ldots, y_{\sigma_n}\rangle.H) = \{\phi\} \Cup (\Cup_{i=1,\ldots,n}\{y_{\sigma_i}\}) \Cup \mathrm{fv}(H)$$
$$\mathrm{fv}(\phi(y_{\sigma_1}, \ldots, y_{\sigma_n}).H) = \{\phi\} \Cup \mathrm{fv}(H) \backslash \{y_{\sigma_1}, \ldots, y_{\sigma_n}\} \quad \mathrm{fv}((\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H) = \mathrm{fv}(H) \backslash \{y_{\sigma_1}, \ldots, y_{\sigma_n}\}$$

$$\mathrm{bv}(\phi(y_{\sigma_1}, \ldots, y_{\sigma_n}).H) = \mathrm{bv}((\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H) = \{y_{\sigma_1}, \ldots, y_{\sigma_n}\} \Cup (\{y_{\sigma_1}, \ldots, y_{\sigma_n}\} \Cap \mathrm{fv}(H)) \Cup \mathrm{bv}(H)$$

Rules for structural congruence

$$(H \mid K) \equiv (K \mid H) \qquad ((H_1 \mid H_2) \mid H_3) \equiv (H_1 \mid (H_2 \mid H_3)) \qquad (H \mid \mathbf{0}) \equiv H \qquad !H \equiv (H \mid !H)$$
$$((\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H \mid K) \equiv (\nu y_{\sigma_1}, \ldots, y_{\sigma_n})(H \mid K), \text{ if } \{y_{\sigma_1}, \ldots, y_{\sigma_n}\} \Cap \mathrm{fv}(K) = \emptyset$$
$$(\nu y_\top)\mathbf{0} \equiv \mathbf{0} \qquad (\nu y_{\sigma_1}, \ldots, y_{\sigma_n})(\nu z_{\rho_1}, \ldots, z_{\rho_m})H \equiv (\nu z_{\rho_1}, \ldots, z_{\rho_m})(\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H$$
$$H \equiv K, \text{ if } H =_\alpha K, \text{ assuming that alpha-conversion preserves decorations:}$$
$$\phi(y_{\sigma_1}, \ldots, y_{\sigma_n}).H =_\alpha \phi(z_{\sigma_1}, \ldots, z_{\sigma_n}).H[z_{\sigma_1}/y_{\sigma_1}, \ldots, z_{\sigma_n}/y_{\sigma_n}], \text{ and}$$
$$(\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H =_\alpha (\nu z_{\sigma_1}, \ldots, z_{\sigma_n})H[z_{\sigma_1}/y_{\sigma_1}, \ldots, z_{\sigma_n}/y_{\sigma_n}]$$

Rules for reduction

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

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

$$\frac{H \xrightarrow{\eta} H' \qquad \eta \notin \{y_{\sigma_1}, \ldots, y_{\sigma_n}\}}{(\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H \xrightarrow{\eta} (\nu y_{\sigma_1}, \ldots, y_{\sigma_n})H'} \quad [\text{R-DRes}]$$

$$\mathsf{if\ true\ then}\ H\ \mathsf{else}\ K \xrightarrow{\tau} H \qquad \mathsf{if\ false\ then}\ H\ \mathsf{else}\ K \xrightarrow{\tau} K \quad [\text{R-DIfT}],[\text{R-DIfF}]$$

$$\frac{H \xrightarrow{\eta} H'}{(H \mid K) \xrightarrow{\eta} (H' \mid K)} \qquad \frac{H \equiv K \quad K \xrightarrow{\eta} K' \quad K' \equiv H'}{H \xrightarrow{\eta} H'} \quad [\text{R-DPar}],[\text{R-DStruct}]$$

Figure 3: Target language

Definition 4.1.

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

The example illustrates the construction of decorations.

Example 4.2. *Take the π-calculus process $x(y)$ with x having end point type $?T.S$ and y having type $T = (!\mathsf{bool.end}, ?\mathsf{bool.end})$. Its representation in decorated π is obtained by projecting y trough the call $\mathtt{tp}(y, T)$, obtaining $x_{?T.S}(y_{!bool.end}, y_{?bool.end}, y_\top)$. Note that the decoration with $\top$ permits to represent type $(\mathsf{end}, \mathsf{end})$, while preserving the unicity of decorations in tuples of the form $y_{\sigma_1}, \ldots, y_{\sigma_n}$. Similarly, a process $\overline{x}\langle y\rangle$ with x having type $!T.S$ and y having type T, where $T = (!bool.?\mathsf{bool.end}, ?\mathsf{bool}.!\mathsf{bool.end})$, is represented as $\overline{x_{!T.S}}\langle \mathtt{tp}(y, T)\rangle$, where $\mathtt{tp}(y, T) = y_{!bool.?bool.end}, y_{?bool.!bool.end}, y_{?bool.end}, , y_{!bool.end}, y_\top$.*

It is interesting to note that decorations allow a more precise characterisation of what is an error: a process containing two processes prefixed with the same variable that *do not synchronise.*

Definition 4.3. *A process of the form $J \equiv (\nu\tilde{\zeta})(H \mid K)$ is an* error*, where*

1. *$H = \overline{x_S}\langle\omega\rangle.H_1 \mid \overline{x_R}\langle\psi\rangle.H_2$, for some x_S, x_R, ω and ψ; or*
2. *$H = x_S(\tilde{\phi}).H_1 \mid x_R(\tilde{\phi}).H_2$, for some x_S and x_R; or*
3. *$H = \overline{x_S}\langle\omega\rangle.H_1 \mid x_R(\tilde{\phi}).H_2$, for some x_S, x_R, ω such that $R \neq \overline{S}$.*

Note that $\overline{x_{S_1}}\langle\omega\rangle.H_1 \mid \overline{x_{S_2}}\langle\psi\rangle.H_2 \mid x_{\overline{S_2}}(\tilde{\phi}).H_3$ is an error, because of the first condition.

Sound processes H such that $z_{\mathsf{end}} \notin \mathrm{var}(H)$, for any z, are determined through five syntactic conditions. First, we are interested only in process having valid decorations for the same variable, i.e., types must form a chain (enforcing the sequential behaviour prescribed by the session types).

Definition 4.4 (Validity)**.** *A multiset of prefixes I is* valid*, noted $\mathsf{valid}(I)$, whenever it satisfies the predicate below.*

$$\begin{aligned}
&\mathsf{valid}(I \uplus \{x_{?T.\mathsf{end}}\}) \text{ :- } x_{\mathsf{bool}} \notin I.\\
&\mathsf{valid}(I \uplus \{x_{?T.S}\} \uplus \{x_S\}) \text{ :- } x_{\mathsf{bool}} \notin I.\\
&\mathsf{valid}(I \uplus \{x_{!T.\mathsf{end}}\}) \text{ :- } x_{\mathsf{bool}} \notin I.\\
&\mathsf{valid}(I \uplus \{x_{!T.S}\} \uplus \{x_S\}) \text{ :- } x_{\mathsf{bool}} \notin I.
\end{aligned}$$

We say that a process H is valid whenever $\mathsf{valid}(\mathrm{var}(H))$.

Notice that, for example, process $H = \overline{x_{!\mathsf{bool}.!\mathsf{bool.end}}}\langle\mathsf{true}\rangle$ is not valid because $x_{!\mathsf{bool.end}} \notin \mathrm{var}(H)$, while $H' = \overline{x_{!\mathsf{bool.end}}}\langle\mathsf{true}\rangle \mid \mathsf{if}\, x_{\mathsf{bool}}\, \mathsf{then}\, \mathbf{0}$ is not valid because $\{x_{!\mathsf{bool.end}}, x_{\mathsf{bool}}\} \Subset \mathrm{var}(H')$; in contrast, $x_{?\mathsf{bool.end}}(y_{\mathsf{bool}})$ is valid.

Based on decorations, we mimic the type system in Figure 1 and enforce send and receive of values of the expected types by using a *coherence* inference system, noted $\vdash$,[5] that does not rely on type contexts. This 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, as it does only part of its job. In particular, it does not enforce neither linearity nor sequential behaviour. Those conditions are checked separately, in the Definitions 4.6 and 4.7 below.

The rules in Figure 4 inductively define the coherence inference system $\vdash$.

Definition 4.5 (Coherence)**.** *We say that a process H is* coherent*, if $\vdash H$.*

Linearity is a fundamental aspect of the session type-discipline. We can identify this property by counting the occurrences of free and bound prefixes, and of their duals. Free session channels must be used *linearly* and bound ones occur exactly twice: once in the binder and once in the process underneath the binder.

[5]The symbol $\vdash$ here denotes a unary predicate on decorated processes.

$$\frac{T = (S, \overline{S}) \text{ or } T = \mathsf{bool} \qquad \vdash H}{\vdash x_{?T.R}(\mathtt{tp}(y, T)).H} \qquad \frac{T = (S, \overline{S}) \text{ or } T = \mathsf{bool} \qquad \vdash H}{\vdash \overline{x_{!T.R}}\langle \mathtt{tp}(y, T)\rangle.H} \qquad \frac{b = \mathsf{true}, \mathsf{false} \qquad \vdash H}{\vdash \overline{x_{!\mathsf{bool}.S}}\langle b\rangle.H}$$

$$\frac{\vdash H \qquad \vdash K}{\vdash \mathsf{if}\, x_{\mathsf{bool}}\, \mathsf{then}\, H\, \mathsf{else}\, K} \qquad \frac{b = \mathsf{true}, \mathsf{false} \qquad \vdash H \qquad \vdash K}{\vdash \mathsf{if}\, b\, \mathsf{then}\, H\, \mathsf{else}\, K}$$

$$\frac{T = (S, \overline{S}) \qquad \vdash H}{\vdash (\nu\ \mathtt{tp}(x, T))H} \qquad \frac{\vdash H}{\vdash !H} \qquad \frac{\vdash H \qquad \vdash K}{\vdash H \mid K} \qquad \vdash \mathbf{0}$$

Figure 4: Coherence Inference System

Definition 4.6 (Linearity)**.** *A process H is* linear *if*

1. $\mathsf{occurs}(x_S, \mathrm{fv}(H)) = 1$ *implies* $\mathsf{occurs}(x_{\overline{S}}, \mathrm{fv}(H)) = 1$*;*
2. $\mathsf{occurs}(x_\sigma, \mathrm{fv}(H)) > 1$ *implies* $\sigma = \mathsf{bool}$ *or* $\sigma = \top$*;*
3. $\mathsf{occurs}(x_S, \mathrm{bv}(H)) = 2$ *implies* $\mathsf{occurs}(x_{\overline{S}}, \mathrm{bv}(H)) = 2$*;*
4. $\mathsf{occurs}(x_\sigma, \mathrm{bv}(H)) > 0, \neq 2$ *implies* $\sigma = \mathsf{bool}$ *or* $\sigma = \top$*.*

An example of a linear process is $\overline{x_S}\langle z_R, z_{\overline{R}}\rangle \mid x_{\overline{S}}(y_R, y_{\overline{R}}).(\overline{y_R}\langle \mathsf{true}\rangle \mid y_{\overline{R}}(w_{\mathsf{bool}}))$, whereas processes (i) $\overline{x_{!\mathsf{bool}.\mathsf{end}}}\langle \mathsf{true}\rangle \mid x_{?(\mathsf{end},\mathsf{end}).\mathsf{end}}(z_\top)$ and (ii) $(\nu y_R)\mathbf{0}$ are not linear because (i) x occurs twice but its decoration types are not dual, and (ii) variable y_R must be used (remind that we do not consider decorations of type end).

Next, we take into consideration the preservation of the order of the exchanges prescribed by the decorations. Let the *size* of an end point type S be equal to the number of prefix-end point types contained in S. Let D range over contexts of $\mathcal{H}$, which are processes H containing a hole, and let $D[H]$ be the process obtained by substituting the occurrence of the hole in D with H.

Definition 4.7 (Preservation of order)**.** *A process H is* order-preserving *if*

1. $H \equiv D[\overline{x_S}\langle\omega\rangle.K_1 \mid K_2]$ *and* $K_1 \equiv D_1[\overline{x_R}\langle\psi\rangle.K' \mid K'']$ *or* $K_1 \equiv D_1[x_R(\tilde{\phi}).K' \mid K'']$ *with* $x_R \notin \mathrm{bv}(D_1)$ *imply* $\mathit{size}(S) \geq \mathit{size}(R)$*;*
2. $H \equiv D[x_S(\tilde{\phi}).K_1 \mid K_2]$ *and* $K_1 \equiv D_1[\overline{x_R}\langle\omega\rangle.K' \mid K'']$ *or* $K_1 \equiv D_1[x_R(\tilde{\phi}).K' \mid K'']$ *with* $x_R \notin \mathrm{bv}(D_1)$ *implies* $\mathit{size}(S) \geq \mathit{size}(R)$*.*

The process $H_1 \stackrel{\text{def}}{=} \overline{x_S}\langle\rangle.\overline{x_{!T.S}}\langle\rangle \mid x_{\overline{S}}().\overline{y_\sigma}\langle\rangle$ (where we omitted the sent values and bound prefixes and left the decoration implicit) is not order-preserving, as $H_1 \xrightarrow{x_S} \overline{x_{!T.S}}\langle\rangle \mid \overline{y_\sigma}\langle\rangle$ and $\mathrm{size}(!T.S) > \mathrm{size}(S)$, whereas the process below, is order-preserving and the order is preserved by reduction.

$$\overline{x_{!T.S}}\langle\rangle \mid x_{\overline{!T.S}}().\overline{z_R}\langle\rangle \mid z_{\overline{R}}().\overline{x_S}\langle\rangle \mid x_{\overline{S}}().\overline{y_\sigma}\langle\rangle$$

The last syntactic property requires some prefixes at the same level to be compatible, i.e., able of synchronising. Requiring the compatibility of *all* prefixes is a too strong condition: it detects circular deadlocks like $y_{\overline{\sigma}}().\overline{x_S}\langle\rangle \mid x_{\overline{S}}().\overline{y_\sigma}\langle\rangle$, but also detects "good" behaviours like the process $H_2 \stackrel{\text{def}}{=} x_{\overline{S}}().y_{\overline{\sigma}}() \mid \overline{x_S}\langle\rangle \mid \overline{y_\sigma}\langle\rangle$. A notion sensitive to particular versions of deadlocks (like resource self-holding ones) needs to be parametric on a set of selected prefixes F.

Definition 4.8 (Parametric mismatch-freedom)**.** *A process H is F-mismatch-free if for all $x_S \in F \subseteq \mathcal{F}$ there is a D such that $H \equiv D[\overline{x_S}\langle\omega\rangle.H_1 \mid x_{\overline{S}}(\tilde{\phi}).H_2]$, with $D[\cdot] \not\equiv (\nu\tilde{\zeta})(\cdot \mid \overline{x_R}\langle\psi\rangle.K_1 \mid K_2)$ and $D[\cdot] \not\equiv (\nu\tilde{\zeta})(\cdot \mid x_R(\tilde{\phi}).K_1 \mid K_2)$.*

Our notion of soundness is thus parametric with respect to a given set of *subjects* of a process H (noted $\mathrm{subjv}(H)$), i.e., a set of prefixes ϕ occuring in H.

Definition 4.9 (Sound process)**.** *A process H such that $z_{\mathsf{end}} \notin \mathrm{var}(H)$, for any z, is F-*sound*, $F \subseteq \mathrm{subjv}(H)$, if it is valid, coherent, linear, order-preserving and F-mismatch-free.*

Theorem 4.10. *If H is a sound process and $H \Rightarrow K$, then K is not an error.*

In the decorated setting, the notion of resource self-holding deadlock freedom, introduced in Section 2, is quite intuitive: prefixes with dual decorations must run in parallel (as in H_2 above).

Definition 4.11. *A sound process H is* Resource Self-Holding Deadlock-Free *(or* RSHDF *), if $x_S \in \mathrm{var}(H)$ implies that $H \equiv D[K_1 \mid K_2]$ for some D, K_1 and K_2, such that $x_S \in \mathrm{fv}(K_1)$ and $x_{\overline{S}} \in \mathrm{fv}(K_2)$.*

Theorem 4.12. *If H is* RSHDF *and $H \to K$ then K is* RSHDF *.*

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 Section 2 into decorated processes in $\mathcal{H}$ of Section 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 is compositional (with respect to parallel threads). We stress again that the class 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 has the form $\Gamma \Vdash_{\mathsf{A}} P \triangleright H$, meaning that given in input an environment Γ in $\mathcal{G}$ such that Γ is semi-balanced (cf. sbal predicate in Section 2), and a process P in $\mathcal{P}$, a (disentangled) process H in $\mathcal{H}$ is returned in output; this implies that $\Gamma \vdash P$ (cf. Figure 2), 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. The top level call $\Vdash_{\mathsf{A}}$ makes use of the the function $\vdash_{\mathsf{A}}$, which is the core of the type-checking and disentangling mechanism.

The formal definitions of functions $\Vdash_{\mathsf{A}}$ and $\vdash_{\mathsf{A}}$ are in Figures 5 and 6, where the rules are assumed to be executed in the given order. In these figures we introduce the left rules for $\vdash_{\mathsf{A}}$; the right rules follow the same schema. Function $\vdash_{\mathsf{A}} : \mathcal{G} \times \mathcal{D} \times \mathcal{P} \to \mathcal{G} \times \mathcal{D} \times \mathcal{H}$, where the set $\mathcal{D}$ contains *projections* Δ mapping variables to types $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T} \cup \{(S, \mathsf{start}), (\mathsf{start}, S), (\mathsf{start}, \mathsf{start})\}$, implements the function $\Vdash_{\mathsf{A}} : \mathcal{G} \times \mathcal{P} \to \mathcal{H}$. Intuitively, in $\vdash_{\mathsf{A}}$ patterns: (i) start is a flag type that signals that a session end point has not been used; (ii) the return environment is obtained by setting to end the end point types that are used within the call (so that parallel threads cannot re-use sessions); (iii) and the return projection permits to infer whether two end points of a channel are used in parallel or in sequence — in the latter case, we invoke disentangling in order to generate a new process where the two end points run in parallel, and return it as third argument. We use A to range over $\mathcal{A}$, and use $\perp$ to indicate the type $(\mathsf{start}, \mathsf{start})$.

The procedure $\vdash_{\mathsf{A}}$ does two things: (1) checks for the typability of process received in input (with respect to a given type context); and (2) transforms resource self-holding deadlocked processes in $\mathcal{P}$ into disentangled processes in $\mathcal{H}$; this last step is preceded by decorating the input process with the expected types.

Before introducing the patterns for $\vdash_{\mathsf{A}}$, it is convenient to analyse the top-level call $\Vdash_{\mathsf{A}}$ in the first line of Figure 5. After verifying that the predicate $\mathsf{sbal}(\Gamma)$ holds, we add the projection

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 values

$$\Gamma \vdash_{\mathsf{A}} \mathsf{true}, \mathsf{false} \colon \mathsf{bool} \triangleright \Gamma \qquad \frac{T = \mathsf{bool}}{\Gamma, x \colon T \vdash_{\mathsf{A}} x \colon T \triangleright \Gamma, x \colon T} \qquad \text{[A-Constant],[A-Bool]}$$

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

Patterns for processes

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

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

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

$$\frac{b = \mathsf{true}, \mathsf{false} \qquad \Gamma_1, x \colon (S, \mathsf{end}); \Delta_1, x \colon (\mathsf{start}, \overline{!\mathsf{bool}.S}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2, x \colon \top; \Delta_2, x \colon (\mathsf{start}, R); H \qquad (\%)}{\Gamma_1, x \colon (!\mathsf{bool}.S, \overline{!\mathsf{bool}.S}); \Delta_1, x : \bot \vdash_{\mathsf{A}} \overline{x}\langle b \rangle.P \triangleright \Gamma_2, x \colon (\mathsf{end}, R); \Delta_2, x \colon \bot; K} \text{[A-OutInitB-L]}$$

$$\frac{b = \mathsf{true}, \mathsf{false} \qquad \Gamma_1, x \colon (S, \mathsf{end}); \Delta_1, x \colon (\mathsf{start}, R) \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad (\%\%)}{\Gamma_1, x \colon (!\mathsf{bool}.S, \mathsf{end}); \Delta_1, x \colon (\mathsf{start}, R) \vdash_{\mathsf{A}} \overline{x}\langle b \rangle.P \triangleright \Gamma_2; \Delta_2; K} \text{[A-OutEnvB-L]}$$

$$\frac{b = \mathsf{true}, \mathsf{false} \qquad \Gamma_1, x \colon (\mathsf{end}, R); \Delta_1, x \colon (S, \mathsf{start}) \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad (\%\%\%)}{\Gamma_1, x \colon (\mathsf{end}, R); \Delta_1, x \colon (!\mathsf{bool}.S, \mathsf{start}) \vdash_{\mathsf{A}} \overline{x}\langle b \rangle.P \triangleright \Gamma_2; \Delta_2; K} \text{[A-OutProjB-L]}$$

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

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

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

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

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

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

Figure 5: Type checking function (part 1)

parameter $(\Gamma \lfloor)$, which is obtained by *casting* all types in the range of Γ to $\bot$, and launch the inner call $\vdash_{\mathsf{A}}$. If the sub-call generates an exception (which is due to the violation of the linearity of sessions), type-checking fails, otherwise a triple (Γ_1, Δ_1, H) is returned. To enforce the consumption of all sessions, function $\Vdash_{\mathsf{A}}$ successfully returns process H when both the context Γ_1 and the

Patterns for processes

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

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

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

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

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

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

$$\frac{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} P \triangleright \Gamma_2; \Delta_2; H \qquad \Gamma_1; \Delta_1 \vdash_{\mathsf{A}} Q \triangleright \Gamma_2; \Delta_2; K \qquad b = \mathsf{true}, \mathsf{false}}{\Gamma_1; \Delta_1 \vdash_{\mathsf{A}} \mathsf{if}\, b\, \mathsf{then}\, P\, \mathsf{else}\, Q \triangleright \Gamma_2; \Delta_2; \mathsf{if}\, b\, \mathsf{then}\, H\, \mathsf{else}\, K} \text{[A-IfB]}$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{A}} P \triangleright \Gamma; \Delta; H}{\Gamma; \Delta \vdash_{\mathsf{A}} !P \triangleright \Gamma; \Delta; !H} \text{[A-Repl]}$$

$$\Gamma; \Delta \vdash_{\mathsf{A}} \mathbf{0} \triangleright \Gamma; \Delta; \mathbf{0} \text{[A-Inact]}$$

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

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

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

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

Figure 6: Type checking function (part 2)

projection Δ_1 are terminated (assuming $\mathsf{term}(\mathsf{start}, \mathsf{start})$).

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[6]; 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

[6]Rules are deterministic for semi-balanced contexts.

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. Figure 7) passing as arguments the decorated process $\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H$, 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: $K := \overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H$ when $R = \overline{!T.S}$, and $K := [\![\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H]\!]^{\mathsf{false}}$ 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 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 $\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H$, otherwise we invoke disentangling on $\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H$.

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 $\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H$, otherwise we return $[\![\overline{x_{!T.S}}\langle \mathtt{tp}(y,T)\rangle.H]\!]^{\mathsf{false}}$. The remaining rules for output, [A-OutInitB-L],[A-OutEnvB-L],[A-OutProjB-L], do apply when the sent value is a boolean constant, and respectively follow the same mechanism of [A-OutInit-L],[A-OutEnv-L] and [A-OutProj-L].

Consider now the rule for parallel composition, [A-Par], in Figure 6. 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 $(\Gamma_3, \Delta_3, H \mid K)$.

The rules for input, [A-InInit-L], [A-InEnv-L] and [A-InProj-L], follow in Figure 6 and are analogous respectively to [A-OutInit-L], [A-OutEnv-L] and [A-OutProj-L] in Figure 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) when necessary, disentangling is invoked through a *snippet* that first finds the actual (free) value (through function $\mathsf{findValue}$ defined below) sent over the sequential output corresponding to the input prefix: when the value is a bound variable the procedure returns 0, a special flag different from ω, and disentangling is discarded; otherwise the value is some ω and we invoke the disentangling function $[\![\cdot]\!]$ in Figure 7 by passing as arguments $x_{?T.S}(\mathtt{tp}(y,T)).H$ and ω.

The definition of the function $\mathsf{findValue}\colon \mathcal{F} \times \mathcal{G} \times \mathcal{H} \to \mathcal{V} \cup \{0\}$, which it is used to find the (tuple of) value(s) to be sent, is below; we use the special symbol 0 to signal when the sent values

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

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

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

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

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

Figure 7: Transformation of decorated processes

are a tuple $z_{\sigma_1}, \ldots, z_{\sigma_n}$ such that $z \notin \mathrm{dom}(\Gamma)$, i.e. the tuple is bound, and disentangling is not possible; we use e to range over $\mathcal{V} \cup \{0\}$. In the definition, we assume that $K = D[\overline{y_\sigma}\langle\omega\rangle.K_1 \mid K_2]$, for some context D such that $y_\sigma \notin \mathrm{bv}(D[\mathbf{0}])$.

$$\mathsf{findValue}(\overline{y_\sigma}, \Gamma, K) = \begin{cases} \omega & \text{, if } \omega = z_{\sigma_1}, \ldots, z_{\sigma_n} \;\&\&\; z \in \mathrm{dom}(\Gamma) \;\&\&\; \mathsf{occurs}(y_\sigma, \mathrm{fv}(K)) = 1; \\ \omega & \text{, if } \omega = \mathsf{true}, \mathsf{false} \;\&\&\; \mathsf{occurs}(y_\sigma, \mathrm{fv}(K)) = 1; \\ 0 & \text{, otherwise.} \end{cases}$$

Note that in rules [A-InEnv-L],[A-InProj-L], and rules [A-OutEnv-L],[A-OutProj-L], we may use backtracking when the types for the channel respectively in the context and in the projection have the same shape. For instance, in contrasting an input channel x w.r.t. an environment Γ and projection Δ such that $\Gamma(x) = (?T_1.S_1, \mathsf{end})$ and $\Delta(x) = (\mathsf{start}, ?T_2.S_2)$, we first try to type-check x w.r.t. $(?T_1.S_1, \mathsf{end})$, and then, if an exception is raised, we try to type-check x w.r.t. $(\mathsf{start}, ?T_2.S_2)$.

The encoding $[\![\cdot]\!]$ in Figure 7 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 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.

The translation $[\![\cdot]\!]$ works on *untyped decorated processes*: during type checking we decorate each variable with its type, and identify the pair of prefixes to be put in parallel by inspecting decorations. The main result is that processes generated by $[\![\cdot]\!]$ are both (resource self-holding) deadlock-free and error-free. The last property is shown by relying on light-weight verification of the format of decorations, thus avoiding the use of type contexts and systems.

Results. The type-checking and resource self-holding deadlock resolution algorithm has three crucial properties that guarantee the correctness of our approach:

1. it is sound with respect to the type system (Theorem 5.1);
2. returns error-free *RSHDF* processes (Theorem 5.3).

The first result guarantees that the algorithm succeeds only when type-checking succeeds. To establish the soundness of the algorithm, we project the algorithmic pattern rules in Figure 5 and 6 into the split-based typing rules in Figure 2. Note that $\Vdash_{\mathsf{A}}$ enforces the context received in input to be semi-balanced in order to launch the inner call $\vdash_{\mathsf{A}}$. The construction of the proof of the theorem is similar to the one in [16], as the process returned in output by $\Vdash_{\mathsf{A}}$ is ignored.

Theorem 5.1 (Type soundness)**.** *If* $\Gamma \Vdash_{\mathsf{A}} P \triangleright H$ *then* $\Gamma \vdash P$.

To identify the deadlocked prefixes of the source process, we let the algorithm return both the source and the target decorated process: in $\Gamma \Vdash\!\!\Vdash_A P \triangleright H_P; H$,[7] let H_P be the decoration of P and H be obtained by disentangling H_P with $[\![\cdot]\!]$.

We use a function $\mathsf{erase}\colon \mathcal{H} \Rightarrow \mathcal{P}$ to remove decorations and transform object tuples into single variables, transforming a polyadic into a monadic process; the non-homomorphic instances of the function are below.

Definition 5.2.

$$\mathsf{erase}((\nu\ \mathtt{tp}(y,T))H = (\nu y\colon T)\mathsf{erase}(H) \qquad \mathsf{erase}(x_S(\mathtt{tp}(y,T)).H = x(y).\mathsf{erase}(H)$$
$$\mathsf{erase}(\overline{x_S}\langle \mathtt{tp}(y,T)\rangle.H = \overline{x}\langle y\rangle.\mathsf{erase}(H) \qquad \mathsf{erase}(\overline{x_S}\langle b\rangle.H) = \overline{x}\langle b\rangle.\mathsf{erase}(H) \qquad b = \mathsf{true}, \mathsf{false}$$

We expect a strong operational correspondence between typed processes of the source language and their decorated versions (obtained during the execution of the algorithm): the reduction steps of the source process and of the decorated one should coincide exactly.

Let $\Gamma \Vdash\!\!\Vdash_A P \triangleright H_P; H$.

1. If $P \xrightarrow{x} P'$ then $H_P \xrightarrow{x_R} H'$ where $\Gamma(x) = (S, \overline{S})$, $R = S$ or $R = \overline{S}$, and $\mathsf{erase}(H') = P'$;
2. If $P \xrightarrow{\tau} P'$ then $H_P \xrightarrow{\tau} H'$ and $\mathsf{erase}(H') = P'$;
3. If $H_P \xrightarrow{x_R} H'$ then $P \xrightarrow{x} P'$ where $\Gamma(x) = (S, \overline{S})$, $R = S$ or $R = \overline{S}$, and $\mathsf{erase}(H') = P'$;
4. If $H_P \xrightarrow{\tau} H'$ then $P \xrightarrow{\tau} P'$ and $\mathsf{erase}(H') = P'$.

The result below is the main one of the paper: it ensures that the process returned by the algorithm is sound with respect to the sequential variables of the source process (cf. Definition 4.9) and resource self-holding deadlock-free (cf. Definition 4.11).

Theorem 5.3. *If* $\Gamma \Vdash\!\!\Vdash_A P \triangleright H_P; H$ *then* H *is* $\mathrm{sv}(H_P)$*-sound and* RSHDF .

[7] System $\Vdash\!\!\Vdash_A$ is a straightforward modification of system $\Vdash_A$.

It is worth noting that, for simplicity sake, we deliberately build our construction on top of a standard π-calculus with booleans and described by the usual reduction semantics. In this setting, a behavioural correspondence among the original, deadlocked process and the new, deadlock-free one (translated to the source language using the erase function), is not straightforward to achieve — an operational correspondence or a simulation relation is not meaningful, as the original process does not reduce. Actually, the situation is even more delicate, as a well typed process with a balanced context is behavioural equivalent to inaction, as all its free channels are linearly used in the process (which "consumes" both end points), so no interaction with a process context is possible.

Therefore, a relationship between the processes can be established by abstracting the behaviour of the deadlocked process by means of its session types: one may infer the substitutions from variables to values, as prescribed by the session protocol for the deadlocked process, and show that any reduction on the deadlock-free process faithfully preserves these substitutions. We leave this to future work.

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 [25] and on a recent formulation of session types [17, 18], 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.

Our construction is as follows: the type checking and disentangling algorithm, based on functional patterns, given in input a type environment and a process, outputs a process, when type checking is successful. We prove the algorithm type sound by showing that accepted process are typed by a sub-system of [18], which satisfies subject reduction and type-safety. We analyse the behaviour of the output and show that the algorithm generates resource self-holding deadlock free processes that cannot go wrong, and that obey to the session protocol as prescribed by the type environment.

For what concerns the limitations of our approach, the first immediate point to note is that our notion of deadlock seems to be a specific instance of resource holding or *Hold and Wait* deadlocks [8, 22], 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 shape of the type of a is $T \stackrel{\text{def}}{=} (?T'.S, !T'.\overline{S})$ with $T' \stackrel{\text{def}}{=} (\mathsf{end}, R)$ or $T' \stackrel{\text{def}}{=} (R, \mathsf{end})$, i.e. $\mathsf{sbal}(T)$ does not hold; (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 type check, because we cannot resolve the non-determinism caused by the test; (4) we do not tackle processes with circular deadlocks like $a(x).\overline{b}\langle\mathsf{true}\rangle \mid b(y).\overline{a}\langle\mathsf{false}\rangle$. We are already working on an extension that should solve both (1) and (2). The idea is to perform a linear scan of the structure of (decorated) processes *before* executing the algorithm, to collect a series of constraints of the form $x = v$, meaning that the (bound) variable x should be instantiated with v; this is reminding of type-inference techniques. We then call (a variant of) the type checking and disentangling algorithm by passing as further parameter the constraints, e.g. (1) $x = b$, and (2) $z = b$; after applying the constraints to the process, in case (1) we disentangle process $a(x).\overline{b}\langle\mathsf{true}\rangle.b(z) \mid \overline{a}\langle b\rangle$ while in case (2) we disentangle process $a(x).c(z).\overline{a}\langle b\rangle.(\overline{x}\langle\mathsf{true}\rangle \mid x(y)) \mid \overline{c}\langle\mathsf{o}\rangle$, where o is a dummy variable [8]. Issue (3) could be solved by considering the π-calculus with a non-deterministic choice

[8] Note that z cannot be used in the continuation of $c(z)$ because type-checking enforces its delegation in $\overline{a}\langle z\rangle$.

operator, or by devising a communication protocol that implements a similar behaviour (cf. [**?**]), 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. [3]). We leave this for future work, as well as a behavioural theory to relate the source and the resulting process of our tool.

We believe that our technique is interesting because it permits to unblock processes described by session types of the form (S, S); since such types subsume several systems, e.g. [20, 14, 23], our algorithm can be used to release deadlocks in these systems as well. Preliminary tests done in a *ML* implementation of the algorithm are encouraging [1].

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 a useful tool to assist the programmer in the software developing process, by (automatically) releasing deadlocked programs.

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.

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] LOCKRES: a deadlock resolver for the pi calculus. Standard ML of New Jersey, `http://tinyurl.com/mg-source`.

[2] Henk Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. North-Holland, 1981 (1st ed.), revised 1984.

[3] Lorenzo Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.

[4] Luis Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[5] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.

[6] 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.

[7] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.

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

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

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

[11] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopolou. A distributed object-oriented language with session types. In *TGC*, volume 3705 of *LNCS*, pages 299–318. Springer, 2005.

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

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

[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. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[16] Marco Giunti. Algorithmic type-checking for a pi-calculus with name matching and session types. *Journal of Logic and Algebraic Programming*, 2013. In press, pre-print available at `http://authors.elsevier.com/sd/article/S156783261300026X`.

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

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

[19] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

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

[21] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[22] Edgar Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, 1987.

[23] 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.

[24] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[25] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

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

[27] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell*, pages 25–36. ACM, 2008.

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

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

[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] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.

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

[33] 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 *SecReT*, volume 171(4) of *ENTCS*, pages 73–93, 2007.

A Proofs for Section 3

We first show that the substitution of a variable having a balanced type does not introduce resource-self-holding deadlocks.

Lemma A.1. *Let* $\Gamma_1, x\colon T \vdash P$ *with* $T = \mathsf{bool}$*, or* $T = (R, S)$ *such that* $\mathsf{bal}(T)$*, and assume that* $x \notin \mathrm{sv}(P)$*. Let* $\Gamma_2 \vdash v\colon T$ *with* $\Gamma_1 \circ \Gamma_2$ *defined. Then* $\Gamma_1 \circ \Gamma_2 \vdash P[v/x]$ *and* $\mathrm{sv}(P[v/x]) = \mathrm{sv}(P)$*.*

Sketch. The typability part is inherited from the substitution lemma (cf. [18]). The hypotheses T balanced or $T = \mathsf{bool}$, and $x \notin \mathrm{sv}(P)$ are crucial to close the remaining part. Assume $\Gamma_1 \circ \Gamma_2$ defined. We distinguish among the case $T = \mathsf{bool}$ and T a session. When $T = \mathsf{bool}$ we have $\Gamma_1(v) = \mathsf{bool}$ and thus $\{x, v\} \cap \mathrm{subjv}(P) = \emptyset$, because of [T-In-L],[T-In-R],[T-Out-L],[T-Out-R]. Otherwise let $T = (S_1, S_2)$; the hypothesis T balanced tell us that $S_1 = \overline{S}_2$. There two cases corrsponding to (a) $S = \mathsf{end}$ and (b) $S \neq \mathsf{end}$. In case (a) $x \in \mathrm{fv}(P)$ implies $x \in \mathrm{objv}(P)$, again because of [T-In-L],[T-In-R],[T-Out-L],[T-Out-R]. Thus $\mathrm{subjv}(P[v/x]) = \mathrm{subjv}(P)$, and the result follows. In case (b) we have $\Gamma(v) = (\mathsf{end}, \mathsf{end})$, and in turn $v \notin \mathrm{subjv}(P)$, by [T-In-L],[T-In-R],[T-Out-L],[T-Out-R]. The result follows from $x \notin \mathrm{sv}(P)$ and $v \notin \mathrm{subjv}(P)$, which imply $v \notin \mathrm{sv}(P[v/x])$. □

We also need the following result, which could be proved by induction on the length of the typing judgement.

Lemma A.2. *If* $\Delta \vdash Q$ *is a judgement appearing in a derivation of* $\Gamma \vdash P$ *with* Γ *semi-balanced, then* Δ *is semi-balanced.*

The next result says that types that are not balanced must be consumed as prefixes, in the following sense.

Lemma A.3. *Let* $\Gamma, x\colon (R, S) \vdash P$ *with* $\Gamma, x\colon (R, S)$ *semi-balanced. If* $R \neq \overline{S}$ *then* $x \in \mathrm{subjv}(P)$*.*

Sketch. We show that the hypothesis $R \neq \overline{S}$ and $x \notin \mathrm{subjv}(P)$ leads to a contradiction. Take a tree i in the forest for the derivation $\Gamma, x\colon (R, S) \vdash P$; then the root of the tree has the form $\Delta_i, x\colon (\mathsf{end}, \mathsf{end}) \vdash \mathbf{0}$, for some Δ_i such that $\mathrm{dom}(\Gamma, x\colon (R, S)) \subseteq \mathrm{dom}(\Delta_i)$ and $\mathsf{term}(\Delta_i)$, because of [T-Inact]. This means that (R, S) has been consumed in one or more tree of the forest. Now if we assume that $x \notin \mathrm{subjv}(P)$, then there must be at least one tree i containing a node of the form $\Omega_i \vdash \overline{y}\langle x\rangle.Q$ with $\Omega(y) = (!T.U_1, U_2)$ or $\Omega(y) = (U_1, !T.U_2)$. This leads to a contradiction since $\neg\mathsf{bal}(T)$ and all judgements in the tree $\Gamma, x\colon (R, S) \vdash P$ are semi-balanced, by applying Lemma A.2. Therefore $x \in \mathrm{subjv}(P)$, as requested. □

We need a lemma in order to analyse the behaviour of parallel processes; the proof is straightforward.

Lemma A.4. *If* $\Gamma_1 \vdash P_1$*,* $\Gamma_2 \vdash P_2$*, and* $\Gamma_1 \circ \Gamma_2$ *semi-balanced, then* $\mathrm{sv}(P_1 \mid P_2) = \mathrm{sv}(P_1) \cup \mathrm{sv}(P_2) \setminus \mathrm{subjv}(P_1) \cap \mathrm{subjv}(P_2)$*.*

An useful notion to show that a process is *RSHDF* is the one below (cf. Theorem 3.7).

Definition A.5. *A process P is* Weakly Resource-Self-Holding Deadlock-Free (WRSHDF)*, if $x \in \mathrm{bv}(P)$ implies $\{x, x, \dots\} \Cap \mathrm{subjv}(P) \subseteq \mathrm{pv}(P)$.*

Lemma A.6. *If P is RSHDF then P is WRSHDF.*

We prove below the lemma that is the wedge of the proof of closure of *RSHDF* under reduction.

Lemma A.7. *Let $\Gamma \vdash P$ with Γ semi-balanced and P WRSHDF, and assume that $P \xrightarrow{\mu} P'$. Then $\mathsf{next}(\Gamma, \mu) \vdash P'$ and $\mathrm{sv}(P) = \mathrm{sv}(P')$*

Proof. Typability of the redex is inherited by subject reduction. We close the remaining part by proceeding by induction on $P \xrightarrow{\mu} P'$. We draw a couple of examples.

In case [R-Com] the hypothesis is that $P = \overline{x}\langle v\rangle.Q_1 \mid x(z).Q_2$ and $P \xrightarrow{x} Q$ where $Q \stackrel{\text{def}}{=} Q_1 \mid Q_2\{v/z\}$, and $\Gamma \vdash P$ with $\Gamma = \Gamma_1 \circ \Gamma_2$, $\Gamma_1 \vdash \overline{x}\langle v\rangle.Q_1$ and $\Gamma_2 \vdash x(z).Q_2$. By subject reduction (Theorem 2.1) we infer $\Delta \vdash Q$ where $\Delta \stackrel{\text{def}}{=} \mathsf{next}(\Gamma, x)$. To show that Q has the same sequential variables of P we analyze the occurrences of x and v in Q; the remaining free and bound variables are invariant. Consider x: whenever $x \notin \mathrm{subjv}(Q)$ we are done, otherwise we have $x \in \mathrm{subjv}(Q)$ because (a) $x \in \mathrm{subjv}(Q_1)$ or (b) $x \in \mathrm{subjv}(Q_2\{v/z\})$. In case (a) we know that $\Gamma_1 \vdash \overline{x}\langle v\rangle.Q_1$ has been inferred from (i) $\Gamma_1 = \Gamma' \circ \Gamma'', x\colon (!T.S, R)$ $\Gamma' \vdash v\colon T$, and $\Gamma'', x\colon (S, R) \vdash Q$, or (ii) $\Gamma_1 = \Gamma' \circ \Gamma'', x\colon (R, !T.S)$ $\Gamma' \vdash v\colon T$, and $\Gamma'', x\colon (R, S) \vdash Q$. In both cases we infer that $R = \mathsf{end}$ and $\Gamma_2(x) = (\mathsf{end}, ?T.\overline{S})$ in case (i), or $\Gamma_2(x) = (?T.\overline{S}, \mathsf{end})$ in case (ii). Is then easy to apply Lemma A.3 to these results and obtain that $x \in \mathrm{subjv}(Q_2\{v/z\})$, as desired. Case (b) is analogous. Consider now v: we show that $v \in \mathrm{sv}(Q)$ iff $v \in \mathrm{sv}(P)$. We know that $\Gamma_2 \vdash x(z).Q_2$ has been inferred from $\Gamma_2 = \Gamma'', x\colon (?T.S, R)$ and $\Gamma'', x\colon (S, R), z\colon T \vdash Q_2$, or $\Gamma_2 = \Gamma'', x\colon (R, ?T.S)$ and $\Gamma'', x\colon (R, S), z\colon T \vdash Q_2$. In both cases the assumption Γ semi-balanced let us infer that T is balanced. From P *WRSHDF* we know that $z \notin \mathrm{sv}(Q_2)$; we apply Lemma A.4 and obtain $z \notin \mathrm{sv}(Q_1 \mid Q_2)$. From this we can easily conclude by applying substitution (Lemma A.1) and obtain $\mathrm{sv}(Q) = \mathrm{sv}(P)$.

Cases [R-IfT] and [R-IfF] follow directly from the I.H.

As further example, take [T-Par]: $\Gamma_1 \cdot \Gamma_2 \vdash P \mid Q$ inferred from $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$, and $P \mid Q \xrightarrow{\mu} P' \mid Q$ inferred from $P \xrightarrow{\mu} P'$. Subject reduction tell us that both $\mathsf{next}(\Gamma_1, \mu) \vdash P'$ and $\mathsf{next}(\Gamma_1 \cdot \Gamma_2, \mu) \vdash P' \mid Q$. By I.H. we have that $\mathrm{sv}(P) = \mathrm{sv}(P')$. The result then follows by applying Lemma A.4.

Case [R-Struct] follows from the I.H. and from the fact that the structural congruence preserves sequential variables: note indeed that the definition of subject, object, parallel and sequential variables are insensitive to parallel rearrangement and scope extrusion. □

Proof of Theorem 3.7. We proceed by induction on the number of reductions and show that $\mathrm{sv}(P) = \emptyset$ and $P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_{n-1}} Q \xrightarrow{\mu_n} P'$ imply $\mathrm{sv}(P') = \emptyset$, where $n \geq 0$. The base of the induction arises when $n = 0$, that is $P' = P$. Otherwise assume $P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_{n-1}} Q \xrightarrow{\mu} P'$. We use a variant of Theorem 2.1: if $\Gamma \vdash P$ with Γ semi balanced and $P \Longrightarrow Q$ then $\Gamma_1 \vdash Q$ for some Γ_1 semi balanced (cf. [16]). The induction hypothesis is that Q is *RSHDF*; thus $\mathrm{sv}(Q) = \emptyset$. By Lemma A.6 we have that Q is *WRSHDF*; we apply Lemma A.7 and infer that $\mathrm{sv}(P') = \mathrm{sv}(Q)$, that is $\mathrm{sv}(P') = \emptyset$. Thus P' is *RSHDF* , as required. □

Corollary A.8 (Shape of process end)**.** *Let P be a typed closed RSHDF process and assume that $P \Rightarrow P'$ with P' having no reductions. If $P' \equiv (\nu \tilde{x}\colon \tilde{T})(Q_1 \mid Q_2)$ with Q_1 an x-process and $x \notin \mathrm{fv}(Q_2)$, then $x \notin \tilde{x}$.*

Corollary A.9. *Typed closed WRSHDF processes reduce to WRSDHF processes.*

Proof. Let P be a typed closed *WRSHDF* . Since $\mathrm{fv}(P) = \emptyset$, P is *RSHDF* as well. Therefore if $P \Longrightarrow P'$ then P' is *RSHDF* , by applying Theorem 3.7 above. The result then follows from Lemma A.6. □

B Proofs for Section 4

In this section we prove that sound decorated π-calculus processes cannot reach errors during the computation (cf. Theorem 4.10). The following lemma establishes that sound processes are safe.

Lemma B.1. *If a decorated process is sound, then is not an error.*

Proof. Take $H \equiv (\nu\tilde{\phi})(K_1 \mid K_2)$, and assume H sound w.r.t. a set of prefixes $F \subseteq \mathcal{F}$. Assume (a) $K_1 = \phi(\tilde{y_\sigma}).K' \mid \varphi(\tilde{z_\rho}).K''$ or (b) $K_1 = \overline{\phi}\langle\psi\rangle.K' \mid \overline{\varphi}\langle\omega\rangle.K''$. If there is x such that $\phi = x_S$ and $\varphi = x_R$, for some S, R, then $\{\phi, \varphi\} \cap F = \emptyset$, since mismatch-freedom rules out the possibilities that two input/output on a same decorated variable run in parallel. Otherwise, assume (c) $K_1 = \overline{x_S}\langle\omega\rangle.K' \mid x_R(\psi).K''$. If $\{x_S, x_R\} \cap F \neq \emptyset$, then because of mismatch-freedom we have $R = \overline{S}$ and $K_2 \not\equiv (\nu\tilde{\zeta})[\overline{x_U}\langle\theta\rangle.K''' \mid K_3]$, $K_2 \not\equiv (\nu\tilde{\zeta})[x_V(\phi).K'''' \mid K_4]$. □

The next lemma says that soundness is preserved by reduction; this a key property in order to prove the main result of this section.

Theorem B.2. *If H is a sound process and $H \to K$, then K is sound.*

Proof. By induction on $H \to K$. We use a function $=_{\mathsf{Sess}}$ which return *true* whenever the multisets received in input do contain the same number of entries x_S, for all x and S, and returns *false* otherwise; we will use often infix notation. We note that whenever $\mathrm{fv}(H_1) =_{\mathsf{Sess}} \mathrm{fv}(H_2)$ or $\mathrm{bv}(H_1) =_{\mathsf{Sess}} \mathrm{bv}(H_2)$, the two multisetes differ only in the number of entries of the form x_{bool} and $x_\top$: that is, they contain the same sessions. The proof relies on the following result ($*$): $H \xrightarrow{\eta} K$ with H sound implies:

A $(\eta = x_S)$ $\mathrm{fv}(K) =_{\mathsf{Sess}} \mathrm{fv}(H)\backslash\{x_S, x_{\overline{S}}\}$ and $\exists y\,.\ \mathrm{bv}(K) =_{\mathsf{Sess}} \mathrm{bv}(H)\backslash t_p(y,T)$ where $S = !T.S'$, for some S';

B $(\eta = \tau)$ $\mathrm{fv}(K) =_{\mathsf{Sess}} \mathrm{fv}(H)$ and (1) $\mathrm{bv}(H) =_{\mathsf{Sess}} \mathrm{bv}(K)$ or (2) $\exists y, S, z\,.\ \mathrm{bv}(K) =_{\mathsf{Sess}} \mathrm{bv}(H)\backslash(\{y_S, y_{\overline{S}}\} \cup \mathtt{tp}(z,T))$ where $S = !T.S'$, for some S'.

We assume H sound and proceed by induction on $H \xrightarrow{\eta} K$ and show simultaneously both ($*$) and K sound. Take case [R-DCom] and let $I \stackrel{\text{def}}{=} \overline{x_S}\langle\omega\rangle.H \mid x_{\overline{S}}(\mathtt{tp}(y,T)).K$, $I \xrightarrow{x_S} J \stackrel{\text{def}}{=} H \mid K[\omega/\mathtt{tp}(y,T)]$. To see that A holds, assume $x_\sigma \in \mathrm{fv}(K)$. By the hypothesis I sound we infer I linear, and in turn $\sigma = \mathsf{bool}$ or $\sigma = \top$. Moreover, we know that T is balanced. Suppose $T = (R, \overline{R})$ with $R = c_1T_1 \ldots c_nT_n.\mathsf{end}$, $c \in \{!, ?\}$, $n \geq 1$; the case $T = \mathsf{bool}$ or $T = \mathsf{end}$ is trivial. Thus there is z such that $\omega = \mathtt{tp}(z,T)$, because of coherence. Then by linearity $\mathsf{occurs}(y_R, \mathrm{fv}(K)) = 1$,$\mathsf{occurs}(y_{\overline{R}}, \mathrm{fv}(K)) = 1$, …, $\mathsf{occurs}(y_{c_nT_n}, \mathrm{fv}(K)) = 1$, $\mathsf{occurs}(y_{\overline{c_nT_n}}, \mathrm{fv}(K)) = 1$. We infer $\mathsf{occurs}(z_R, \mathrm{fv}(K[\mathtt{tp}(z,T)/\mathtt{tp}(y,T)])) = 1$,$\mathsf{occurs}(x_{\overline{R}}, \mathrm{fv}(K[\mathtt{tp}(z,T)/\mathtt{tp}(y,T)])) = 1$, …, $\mathsf{occurs}(x_{c_nT_n}, \mathrm{fv}(K[\mathtt{tp}(z,T)/\mathtt{tp}(y,T)])) = 1$,$\mathsf{occurs}(x_{\overline{c_nT_n}}, \mathrm{fv}(K[\mathtt{tp}(z,T)/\mathtt{tp}(y,T)])) = 1$. That is: $\mathrm{fv}(J) =_{\mathsf{Sess}} \mathrm{fv}(I)\backslash\{x_S, x_{\overline{S}}\}$, as requested. The fact $\mathtt{tp}(y,T) \notin \mathrm{bv}(K[\mathtt{tp}(z,T)/\mathtt{tp}(y,T)]))$ let us easily infer $\mathrm{bv}(I)\backslash\mathtt{tp}(y,T) =_{\mathsf{Sess}} \mathrm{bv}(J)$, and in turn the proof of A.

Next, we prove that J is sound. To show that J is valid we need to show that S cannot "be contained" in any type R such that $x_R \in \mathrm{fv}(J)$, and that $\overline{S}$ cannot "be contained" in any type $\overline{R}$ such that $x_{\overline{R}} \in \mathrm{fv}(J)$; the fact $x_{\mathsf{bool}} \notin \mathrm{fv}(J)$ is deduced from I valid. We exploit the preservation of order of I and infer that $x_R \in \mathrm{fv}(I)$ imply $\mathrm{size}(S) \geq R$, as required. Next, we need to show that $\vdash J$. From $\vdash I$ we know that $S = ?T.R$ and (i) $\omega = \mathtt{tp}(z,T)$, for some z, or (ii) $T = \mathsf{bool}$ and $\omega = \mathsf{true}, \mathsf{false}$. In both cases we use a substitution lemma that let us infer $\vdash K[\omega/\mathtt{tp}(y,T)]$. From this and the hypothesis $\vdash H$ we obtain $\vdash J$ by applying the parallel rule. Next, we infer the linearity of J. From I sound we know that I is a linear: it satisfy all conditions Def. 4.6(1)–(4). Take condition (1): from (A) we have $\mathrm{fv}(j) =_{\mathsf{Sess}} \mathrm{fv}(I)\backslash\{x_S, x_{\overline{S}}\}$, and in turn we infer that J satisfies (1); similarly, we obtain that I satisfying (3) implies J satisfying (3). Take condition (2): (B(1)) is straightforward, while in case (B(2)) we have that there exist y, S, z such that $\mathrm{bv}(K) =_{\mathsf{Sess}} \mathrm{bv}(H)\backslash(\{y_S, y_{\overline{S}}\} \cup \mathtt{tp}(z,T))$. The type T is the payload of the first end point

in S: note that T is balanced. In particular, this means that $T = \mathsf{bool}$, $T = \top$, or $T = (R, \overline{R})$, for some R. In the last case, we let $R = c_1 U_1 \dots c_n U_n.\mathsf{end}$, where $c_i \in \{?, !\}$, and infer the shape of $\mathtt{tp}(z, T)$: since all decorations in a canonical tuple that are distinct from $\top$ are in $\mathrm{bv}(I)$ and not in $\mathrm{bv}(J)$, we infer that I satisfying Def. 4.6(2) implies J satisfying (2), as desired. Condition (4) is verified analogously. Therefore J satisfies Def. 4.6(1)–(4), that is: J is linear. The fifth soundness condition is satisfied since J is order-preserving by hypothesis, since I is order-preserving. Lastly, if I is F-mismatch-free, for some $F \subseteq \mathcal{F}$, then by definitio J is F'-mismatch-free, where $F' = F$ when $\{x_S, x_{\overline{S}}\} \cap F = \emptyset$, and $F' = F \backslash \{x_S, x_{\overline{S}}\}$ otherwise. Thus J is sound, as desired.

The remaining cases mostly follow by the induction hypothesis, whether to close [R-DRES] we exploit the fact that the reduction does occur on some x_S bound by restriction such that S is the greater decoration type in the bound tuple $x_{S_1}, \dots, x_{S_n}$, because of order preservation. □

Corollary B.3. *If H is a sound process and $H \Rightarrow K$, then K is sound.*

Proof of Theorem 4.10. Apply Corollary B.3 and Theorem B.1.

C Proofs for Section 5

We build towards the main result of this section (cf. Theorem 5.3): the decorated π-calculus processes generated by the algorithm are (1) sound and (2) resource self-holding deadlock free.

The next lemma ensures that the translation preserves soundness.

Lemma C.1. *If K is F-sound, where $F \subseteq \mathcal{F}$, and (a) $K = \overline{x_R}\langle\omega\rangle.H$, or (b) $K = x_R(\mathtt{tp}(y, T)).H$, then $[\![K]\!]$ is $F \uplus \{x_R\}$-sound.*

Sketch. By case analysis of $[\![\cdot]\!]$. To see item (a), note that the hypothesis $\vdash K$ implies that $R = !T.S$ or $R = !T$, for some T and S. We know that $\mathsf{occurs}(x_{\overline{R}}, fv(H)) = 1$, because of K linear; moreover, we claim that $x_{\overline{R}} \in \mathrm{subjv}(H)$. To see this, note that the hypothesis $\vdash K$ and the coherence rules for output require that when the object of an output is a tuple of variables, this must be of the form $\mathtt{tp}(z, T)$ with $\mathsf{bal}(T)$ or $T = \mathsf{bool}$. Since $\mathsf{occurs}(x_R, K) = 1$, this excludes the possibility that $x_{\overline{R}}$ is an object. We find a context D such that $H = D[x_{\overline{R}}(\mathtt{tp}(z, T)).H_1]$; note that $x_{\overline{R}} \notin \mathrm{var}(D)$, by the Barendregt convention and because of linearity. By definition[9] we have:

$$[\![\overline{x_R}\langle\omega\rangle.H]\!] = (\nu r_{!T.\mathsf{end}}, r_{?T.\mathsf{end}}, r_\top)(\overline{x_R}\langle\omega\rangle.D[r_{?T}(\mathtt{tp}(z, T)).H_1] \mid x_{\overline{R}}(\mathtt{tp}(y, T)).\overline{r_{!T\mathsf{end}}}\langle\mathtt{tp}(y, T)\rangle)$$

with the condition that the new tuple is fresh. We then check that the conditions for soundness hold for $[\![K]\!]$. Validity follows from validity of K, and from the fact that $r_{!T.\mathsf{end}}, r_{?T.\mathsf{end}}, r_\top$ are a chain. Coherence is inferred from $\vdash K$, and from the fact that $r_{!T.\mathsf{end}}$ and $r_{?T.\mathsf{end}}$ are used to send and receive a tuple of values decorated with the expected types. Linearity follows from linearity of K, and from $\mathsf{occurs}(r_{!T.\mathsf{end}}, \mathrm{var}([\![K]\!])) = 2$, $\mathsf{occurs}(r_{?T.\mathsf{end}}, \mathrm{var}([\![K]\!])) = 2$, $\mathsf{occurs}(r_\top, \mathrm{var}([\![K]\!])) = 1$. To see order preservation, assume $x_S \in \mathrm{fv}(D)$. From K order preserving we infer $\mathrm{size}(R) \geq \mathrm{size}(S)$. Thus $x_{\overline{R}}$ cannot be prefixed by x_S, because of $\mathrm{size}(R) = \mathrm{size}(\overline{R})$: that is, x_S occurs in the free variables of a parallel composition. Thus $[\![K]\!]$ does preserve the order as well. Last, that $[\![K]\!]$ is F-mismatch-free follows from K F-mismatch-free. We add the new couple of matching prefixes introduced by the translation, $x_R, x_{\overline{R}}$, and obtain the desired result, that is $[\![K]\!]$ $F \uplus \{x_R\}$-mismatch-free, which in turn together with the previous results implies that $[\![K]\!]$ is $F \uplus \{x_R\}$-sound.

Consider item (b). We proceed similarly to above and infer that ($\sharp$) x_R appears exactly once as subject of an output, and find a context D such that $H = D[\overline{x_R}\langle\omega\rangle.H_1]$ and $x_R \notin \mathrm{var}\, D$, i.e. ω is the parameter to be passed to the encoding $[\![\cdot]\!]$. Note that this is possible since K is sound by hypothesis; in the algorithm we must use function $\mathsf{findValue}$ in order to recover ω, when possible. From $\vdash K$ we infer $\omega = \mathtt{tp}(w, T)$, for some w, or $\omega = \mathsf{true}, \mathsf{false}$, and $T = \mathsf{bool}$. Next, from ($\sharp$) and the definition of $\langle\!\langle\cdot\rangle\!\rangle$ we infer $\langle\!\langle H \rangle\!\rangle^{x_R}_{m_{!\top.\mathsf{end}}} = D[\overline{m_{!\top.\mathsf{end}}}\langle\rangle.H_1]$. This let us infer

$$\begin{aligned}[\![x_{\overline{R}}(\mathtt{tp}(y, U)).H]\!]^\omega =& (\nu r_{!T.\mathsf{end}}, r_{?T.\mathsf{end}}, r_\top)(\nu m_{!\top.\mathsf{end}}, m_{?\top.\mathsf{end}}, m_\top) \\ & ((x_{\overline{R}}(\mathtt{tp}(z, T)).\overline{r_{!T.\mathsf{end}}}\langle\mathtt{tp}(z, T)\rangle \mid r_{?T.\mathsf{end}}(\mathtt{tp}(y, T)).D[\overline{m_{!\top.\mathsf{end}}}\langle\rangle.H_1] \mid \overline{x_R}\langle\omega\rangle.m_{?\top.\mathsf{end}}()))\end{aligned}$$

[9] We ignore the encoding's parameter since it is irrelevant for the output first case of $[\![\cdot]\!]$

We check that the soundness conditions holds for $[\![x_{\overline{R}}(\mathtt{tp}(y,U)).H]\!]^{\omega}$; we omit the details, which are analogous to the construction above. To conclude, we obtain that $[\![K]\!]$ is F-mismatch-free because K is F-mismatch-free; we add the matching pair introduced by the translation, that is $\{x_R, x_{\overline{R}}\}$, and obtain that $[\![K]\!]$ is $F \Cup \{x_R, x_{\overline{R}}\}$-mismatch-free, and in turn that $[\![K]\!]$ is $F \Cup \{x_R, x_{\overline{R}}\}$-sound, as desired. □

The next lemma ensures that the translation maps sound processes into *RSHDF* processes.

Lemma C.2. *If K is sound, (a) $K = \overline{x_R}\langle\omega\rangle.H$, or (b) $K = x_R(\mathtt{tp}(y,T)).H$, and H is RSHDF, then $[\![K]\!]$ is RSHDF.*

Sketch. Take item (a). We know that there is D such that $H = D[x_{\overline{R}}(\mathtt{tp}(z,T)).H_1]$ and $x_{\overline{R}} \notin \mathrm{var}(D)$ (see the proof of Lemma C.1). By definition of $[\![\cdot]\!]$ we have:

$$[\![\overline{x_R}\langle\omega\rangle.H]\!] = (\nu r_{!T.\mathsf{end}}, r_{?T.\mathsf{end}}, r_{\top})(\overline{x_R}\langle\omega\rangle.D[r_{?T.\mathsf{end}}(\mathtt{tp}(z,T)).H_1] \mid x_{\overline{R}}(\mathtt{tp}(y,T)).\overline{r_{!T.\mathsf{end}}}\langle\mathtt{tp}(y,T)\rangle)$$

with the condition that the new tuple is fresh. If we show that $D[r_{?T.\mathsf{end}}(\mathtt{tp}(z,T)).H_1]$ is *RSHDF* , we clearly obtain that $[\![K]\!]$ is *RSHDF* as well, as desired. Thus we need to show that if $D[\cdot] \equiv (\nu\tilde{\zeta})(H' \mid \cdot)$ with $x_S \in \mathrm{fv}(H')$, then $S \neq R$. Because of K linear, we know that $\mathsf{occurs}(x_R, \mathrm{var}(K)) = 1$; this implies the desired result, and we are done. Item (b) is analogous, whether we retrieve the value ω to be passed to the encoding $[\![\cdot]\!]$ by exploiting the soundness hypothesis, as in Lemma C.1. □

The next lemma says that the decoration made by a successful run of the algorithm is as expected. Note that mismatch-freedom is not guarantee; this will be enforced by $[\![\cdot]\!]$. We remind that $\Vdash_A$ is a modified version of the algorithm that returns a pair in $\mathcal{H}\times\mathcal{H}$ formed by the decoration of the process received in input (on the left), and the disentangled version of the decorated process (on the right).

Lemma C.3. *If $\Gamma \Vdash_A P \triangleright H_P; H$, then H_P is valid, coherent, linear and order-preserving.*

Sketch. We devise a variant of the typing system in Figure 2 containing judgements of the form $\Gamma \Vdash P \triangleright H_P$, where H_P is a decorated π calculus process forwarded in output that is obtained by (i) storing which end point type (i.e., left of right) has been chosen in the derivation in order to type the subject, (ii) storing the type chosen for the object, (iii) storing that the test variable in if-then-else has been typed with bool. We then show a stronger soundness result (cf. Theorem 5.1): $\Gamma \Vdash_A P \triangleright H_P; H$ implies $\Gamma \Vdash P \triangleright H_P$. The conditions above can then be easily established by reasoning on the typing system in Figure 2, by a case analysis of both the context split rules and the typing rules. □

To show the main result of this section, that is Theorem 5.3, we enlarge the syntax of processes H so to include process variables X, and let $K[H/X]$ be defined as the process obtained by substituting all occurences of X in K with H.

Definition C.4. *We say that a process H is a E/I-variant of process K if there are G, E, I such that $K = G[E/X]$ and $H = G[I/X]$.*

The next lemma identifies the shape of the processes returned by the (variant of the) algorithm. In the following, we use α to range over prefixes of the form $\overline{\phi}\langle\omega\rangle$ and $\phi(\psi)$.

Lemma C.5. *Let $\Gamma \Vdash_A P \triangleright H_0; H$. If $H_0 \neq H$, then there is $n > 0$ and $H_1, \ldots, H_{n-1}$ and $\alpha_1.K_1, \ldots, \alpha_{n-1}.K_{n-1}, \alpha_n.K_n$ such that $H = H_n$ and H_i is a $\alpha_i.K_i/[\![\alpha_i.K_i]\!]$-variant of H_{i-1}, for all $i \in 1, \ldots, n$.*

Proof of Theorem 5.3. We rely on Lemmas C.1, C.2, C.3, C.5. Let $\Gamma \Vdash_A P \triangleright H_0; H_n$. When $n = 0$, we show that $\mathrm{sv}(P) = \emptyset$; we then apply Lemma C.3 and show that H_0 is sound and *RSHDF* . The proof relies on the following result: if $\Gamma; \Delta \vdash_{\mathsf{A}} P \mid Q \triangleright \Gamma_1; \Delta_1; H \mid K; H \mid K$ with Γ and Δ *compatible*, and $\mathsf{term}(\Gamma')$ and $\mathsf{term}(\Delta')$, where Γ', Δ' are obtained respectively by restricting the

domain of Γ_1 and Δ_1 to $\mathrm{fv}(P)$ and $\mathrm{fv}(Q)$, then $\mathrm{pv}(P \mid Q) = \mathrm{subjv}(P \mid Q)$. Γ and Δ are compatible whenever $\Gamma(x) = (S_1, S_2)$ and $\Delta(x) = (R_1, R_2)$ implies $\neg(S_i \neq \mathsf{end} \wedge R_i \neq \mathsf{start})$. Now let $n > 0$. By Lemma C.5, for all $i \in 1, \ldots, n$, H_i is a $\alpha_i.K_i/[\![\alpha_i.K_i]\!]$-variant of H_{i-1}, for some $\alpha_i.K_i$. We prove the following result: each $\alpha_i.K_i/[\![\alpha_i.K_i]\!]$-variant of H_{i-1} is $\mathrm{subjv}(\alpha_i), \ldots, \mathrm{subjv}(\alpha_1) \cap \mathrm{sv}(H_0)$-sound and $\mathrm{subjv}(\alpha_i), \ldots, \mathrm{subjv}(\alpha_1) \cap \mathrm{sv}(H_0)$-*RSHDF* ; this last notion is obtained as expected. The base case is $i = 1$ and is obtained by $H_1 = G[[\![\alpha_1.K_1]\!]/X]$, $H_0 = G[\alpha_1.K_1/X]$, and by applying Lemmas C.1, C.2. Take $i > 1$; the induction hypothesis is that H_{i-1} is $\mathrm{subjv}(\alpha_{i-1}), \ldots, \mathrm{subjv}(\alpha_1) \cap \mathrm{sv}(H_0)$-sound. We have $H_i = G[[\![\alpha_i.K_i]\!]/X]$, $H_{i-1} = G[\alpha_i.K_i/X]$. We apply Lemmas C.1, C.2 and infer the desired result. □