Progress as Compositional Lock-Freedom

Marco Carbone *[1], Ornela Dardha[2], and Fabrizio Montesi[1]

[1] IT University of Copenhagen, Denmark
{carbonem,fmontesi}@itu.dk
[2] University of Glasgow, United Kingdom
Ornela.Dardha@glasgow.ac.uk

Abstract. A session-based process satisfies the *progress* property if its sessions never get stuck when it is executed in an adequate context. Previous work studied how to define progress by introducing the notion of catalysers, execution contexts generated from the type of a process. In this paper, we refine such definition to capture a more intuitive notion of context adequacy for checking progress. Interestingly, our new catalysers lead to a novel characterisation of progress in terms of the standard notion of lock-freedom. Guided by this discovery, we also develop a conservative extension of catalysers that does not depend on types, generalising the notion of progress to untyped session-based processes. We combine our results with existing techniques for lock-freedom, obtaining a new methodology for proving progress. Our methodology captures new processes wrt previous progress analysis based on session types.

1 Introduction

Progress is a fundamental property of safe programs in a language model. Intuitively, a program with the progress property should never get "stuck", i.e., reach a state that is not designated as a final value and that the language semantics does not tell how to evaluate further [23]. Progress is well-understood in models such as the λ-calculus, and typically analysed in closed terms through type systems. On the other hand, we have only recently begun to scratch the surface of its meaning in models for concurrency. A basic property related to progress in concurrency is deadlock-freedom: a process is deadlock-free if it can always reduce unless it terminates [16]. In a deadlock-free process, some subprocesses can get stuck. For instance, consider the following process in the π-calculus [18]:

$$P \;=\; (\boldsymbol{\nu} x)\big(x?(y).\mathbf{0} \mid \mathit{\Omega}\big)$$

where $\mathit{\Omega}$ is a diverging process executing an infinite series of internal actions. Although the subterm $x?(y).\mathbf{0}$ will never reduce, process P is deadlock-free. Following this observation, lock-freedom has been proposed as a stronger property that requires every input/output action to be eventually executed under fair process scheduling [15]: all communications must be reduced even if the whole process diverges. Various static analyses, in particular many type systems, have been proposed for ensuring deadlock- or lock-freedom [15–17, 5, 6].

* Research supported by the Danish Agency for Science, Technology and Innovation.

The aforementioned analyses are applied to closed processes, i.e., processes that do not communicate with the environment. However, process models are often used to capture open-ended systems where participants can join the system dynamically [10, 20, 21, 19]. A recent line of work has begun investigating a *compositional* formulation of progress for such systems, which are captured by open processes missing some participants. An open process has then the progress property if it can reduce within all adequate execution contexts, called *catalysers*, that provide the missing participants [4, 8]. Interestingly, this compositionality seems to lead back to the notion of lock-freedom, in that both notions inspect the behaviour of subprocesses in a system. Thus, we ask:

What is the relationship between the notions of lock-freedom and progress for open-ended systems?

Answering the question above would lead to a better understanding of the progress property for concurrent systems. Ideally, it would allow techniques and results obtained for one property to be applied to the other.

1.1 Contributions

We list our major contributions. Full proofs and definitions can be found in [3].

Progress through typed closure. We study progress and lock-freedom in the π-calculus with sessions [25], by conservatively extending the notion of catalysers based on session types [13, 25] (§ 3). We show that progress and lock-freedom coincide for well-typed closed processes (§ 3, Theorem 2). Building on this result we construct a procedure, called *typed closure*, that wraps an open process in a special catalyser to transform it into a closed process. Typed closure allows us to relate the progress and lock-freedom properties for well-typed processes: a well-typed process has progress if and only if its typed closure is lock-free (§ 3, Theorem 4), i.e., *progress is a compositional form of the notion of lock-freedom.*

Progress through untyped closure. We explore an alternative procedure for closing a process that is not based on session types, but rather on the structure of the process itself, called *untyped closure* (§ 4). Interestingly, we can show that a process has progress if and only if its untyped closure is lock-free, yielding a new characterisation of progress that can capture also untyped processes.

Progress through lock-freedom. We combine our results with existing techniques for guaranteeing lock-freedom, obtaining a new methodology for proving progress in the π-calculus with sessions (§ 5). Specifically, we present how Kobayashi's type system for lock-freedom, from [15], can be reused for establishing whether a process has progress. Our methodology captures new processes wrt previous progress analysis based on session types (§ 5, Comparison).

2 The Model

In this section we introduce the π-calculus with sessions and its typing discipline, from [25], which we will use as reference model for our investigation of progress.

2.1 The π-calculus with sessions

Syntax. The syntax of the π-calculus with sessions is given in Fig.1.

$$
\begin{array}{rlllll}
P, Q, \ldots ::= & x!\langle v\rangle.P & \textit{(output)} & \mid & x?(y).P & \textit{(input)} \\
\mid & x \triangleleft \{l_i.P_i\}_{i\in I} & \textit{(selection)} & \mid & x \triangleright \{l_i : P_i\}_{i\in I} & \textit{(branching)} \\
\mid & P \mid Q & \textit{(parallel)} & \mid & (\boldsymbol{\nu} xy)P & \textit{(restriction)} \\
\mid & \mathbf{rec}X.P & \textit{(rec)} & \mid & X & \textit{(call)} \\
\mid & \mathbf{0} & \textit{(inaction)} & & & \\
v ::= & x & \textit{(var)} & \mid & \mathsf{unit} & \textit{(unit)}
\end{array}
$$

Fig. 1. π-calculus with sessions, syntax.

P, Q range over processes, x, y over variables, and v over values. Values can be either variables or the unit value unit, which abstracts basic values. An output process $x!\langle v\rangle.P$ sends a value v on channel x and proceeds as process P; the input process $x?(y).P$ receives a value on channel x, stores it in variable y and proceeds as P. Process $x \triangleleft \{l_i.P_i\}_{i\in I}$ is a generalisation of the standard selection $x \triangleleft l_j.P_j$ found in [13, 25]: it sends on channel x the selection of a label l_j among the labels in $\{l_i\}_{i\in I}$, and then proceeds as the corresponding process P_j. This generalised selection will be important for our characterisation of progress, in § 3. A label selection is received by a branching process $x \triangleright \{l_i : P_i\}_{i\in I}$, which offers a range of labelled alternatives on channel x followed by their respective process continuations. Term $(\boldsymbol{\nu} xy)P$ binds two variables x and y in P as the two respective endpoints of a session; when restricted together as in $(\boldsymbol{\nu} xy)P$, we say that x and y are co-variables. All the other terms are standard.

Semantics. We give semantics to the π-calculus with sessions in terms of the reduction relation $\rightarrow$, a binary relation over processes, defined by the rules in Fig. 2. Rule (R-Com) is the rule for communication: the process on the left sends a value v on x, while the process on the right receives the value on y and substitutes the placeholder z with it. A key difference wrt the standard π-calculus is that the subject of the output, x, and the subject of the input, y, are required to be co-variables of each other, formalised by the external restriction $(\boldsymbol{\nu} xy)$. A consequence of this is that communication happens only on bound variables. Rule (R-Choice) models an internal choice, in which a process $x \triangleleft \{l_i.P_i\}_{i\in I}$ nondeterministically chooses one of its possible labelled continuations. Rule (R-Sel) is similar to rule (R-Com), but in this case captures the communication of a label selection. We require the label selected by the process on the left to be among the labels offered by the process on the right. Rule (R-Rec) models the recursion process reduction. The remaining rules and the structural congruence $\equiv$ are standard (see [25] for a more complete explanation).

2.2 Typing the π-calculus with sessions

We report a typing discipline for typing sessions in processes, from [25].

$$\begin{array}{rl}
\text{(R-C{\scriptsize OM})} & (\boldsymbol{\nu} xy)(x!\langle v\rangle.P \mid y?(z).Q \mid R) \rightarrow (\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid R) \\
\text{(R-C{\scriptsize HOICE})} & x \triangleleft \{l_i.P_i\}_{i\in I} \rightarrow x \triangleleft l_j.P_j \quad \text{ if } j \in I \\
\text{(R-S{\scriptsize EL})} & (\boldsymbol{\nu} xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R) \rightarrow (\boldsymbol{\nu} xy)(P \mid P_j \mid R) \quad \text{ if } j \in I \\
\text{(R-R{\scriptsize EC})} & P[\mathbf{rec}X.P/X] \rightarrow P' \quad \Rightarrow \quad \mathbf{rec}X.P \rightarrow P' \\
\text{(R-R{\scriptsize ES})} & P \rightarrow Q \quad \Rightarrow \quad (\boldsymbol{\nu} xy)P \rightarrow (\boldsymbol{\nu} xy)Q \\
\text{(R-P{\scriptsize AR})} & P \rightarrow P' \quad \Rightarrow \quad P \mid Q \rightarrow P' \mid Q \\
\text{(R-S{\scriptsize TRUCT})} & P \equiv P',\ P' \rightarrow Q',\ Q' \equiv Q \quad \Rightarrow \quad P \rightarrow Q
\end{array}$$

Fig. 2. π-calculus with sessions, semantics.

Types. The syntax of types is given in Fig. 3.

$$\begin{array}{rlll}
q ::= & \mathsf{lin} & \textit{(linear)} & \mid\ \mathsf{un} \quad \textit{(unrestricted)} \\
p ::= & !T.U & \textit{(send)} & \mid\ ?T.U \quad \textit{(receive)} \\
\mid & \oplus\{l_i : T_i\}_{i\in I} & \textit{(select)} & \mid\ \&\{l_i : T_i\}_{i\in I} \quad \textit{(branch)} \\
T, U ::= & q\ p & \textit{(qualified pretype)} & \\
\mid & \mathsf{end} & \textit{(termination)} & \mid\ \mathbf{1} \quad \textit{(unit type)} \\
\mid & \mu\mathbf{t}.T & \textit{(recursive type)} & \mid\ \mathbf{t} \quad \textit{(rec var)}
\end{array}$$

Fig. 3. Session types, syntax.

Let q range over type qualifiers, p over pretypes, $q\ p$ over qualified pretypes, and T, U over types. Qualifiers are lin (for linear) or un (for unrestricted) and are used respectively to distinguish between types for sessions, i.e., channels whose pretype is executed exactly once, and standard channel types that can be used any number of times in parallel. In the pretypes, $!T.U$ and $?T.U$ are, respectively, the types of a sending and receiving of a value of type T with continuation of type U. Select and branch are sets of labelled session types indicating, respectively, internal and external choice. A type T can be a qualified pretype $q\ p$; end, the type of a terminated session; the unit type unit; a recursive type $\mu\mathbf{t}.T$; or, finally, a type variable $\mathbf{t}$. Recursive types are required to be *contractive*. Type equality in recursive types is based on the regular infinite trees and we consider a recursive type and its unfolding to be equal. In the rest of the paper, we implicitly assume that the qualifier lin is used in every qualified pretype unless it is explicitly stated otherwise. Also, we refer to types with a lin qualifier as session types.

Session Typing. We present now the session typing discipline for the π-calculus with sessions, which avoids communication errors such as type mismatches and race conditions. The syntax of typing environments is defined as:

$$\begin{array}{rcl}
\Gamma & ::= & \emptyset \quad \mid \quad \Gamma,\ x : T \\
\Theta & ::= & \emptyset \quad \mid \quad \Theta,\ X : \Gamma
\end{array}$$

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

$$\frac{\mathsf{un}(\Gamma)}{\Theta;\Gamma,x:T\vdash x:T}\ (\text{T-Var}) \qquad \frac{\Theta;\Gamma,x:T,y:T'\vdash P \quad T\perp T'}{\Theta;\Gamma\vdash(\boldsymbol{\nu}xy)P}\ (\text{T-Res})$$

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

$$\frac{\Theta;\Gamma_1\vdash x:q!T.U \quad \Theta;\Gamma_2\vdash v:T \quad \Theta;\Gamma_3+x:U\vdash P}{\Theta;\Gamma_1\circ\Gamma_2\circ\Gamma_3\vdash x!\langle v\rangle.P}\ (\text{T-Out})$$

$$\frac{\Theta;\Gamma_1\vdash x:q\&\{l_i:T_i\}_{i\in I} \quad \Theta;\Gamma_2+x:T_i\vdash P_i \quad \forall i\in I}{\Theta;\Gamma_1\circ\Gamma_2\vdash x\triangleright\{l_i:P_i\}_{i\in I}}\ (\text{T-Brch})$$

$$\frac{\Theta;\Gamma_1\vdash x:q\oplus\{l_i:T_i\}_{i\in I} \quad \Theta;\Gamma_2+x:T_i\vdash P_i \quad J\subseteq I}{\Theta;\Gamma_1\circ\Gamma_2\vdash x\triangleleft\{l_j.P_j\}_{j\in J}}\ (\text{T-Sel})$$

$$\frac{\Theta,X:\Gamma;\Gamma\vdash P}{\Theta;\Gamma\vdash\mathbf{rec}X.P}\ (\text{T-RecP}) \qquad \frac{\Theta(X)=\Gamma}{\Theta;\Gamma\vdash X}\ (\text{T-RecV})$$

Fig. 4. π-calculus with sessions, typing rules.

We adopt the standard convention that we can write $\Gamma,\ x:T$ only if x does not appear in Γ, and $\Theta,\ X:\Gamma$ only if X does not appear in Θ. Therefore, we can write Γ,Γ' (or Θ,Θ') only if the two environments have disjoint domains. Typing judgements have the form $\Theta;\Gamma\vdash P$, reading "process P is well-typed using variables according to Γ and recursion variables according to Θ". With an abuse of notation, we also write $\Theta;\Gamma\vdash x:T$ for "x has type T in Γ". We report the typing rules in Fig. 4. Rule (T-Inact) states that the terminated process $\mathbf{0}$ is well-typed under an unrestricted Γ, i.e., a Γ containing only types qualified with un, and any Θ. Rule (T-Par) types the parallel composition of two processes; it uses the split operator for typing environments $\circ$, which is defined by the following equations, and is undefined otherwise.

$$\begin{aligned} \emptyset\circ\emptyset &= \emptyset & \\ \Gamma\circ x:T &= \Gamma,x:T & \text{if } x\notin\mathsf{dom}(\Gamma) \\ (\Gamma,x:T)\circ x:T &= \Gamma,x:T & \text{if } T \text{ is not a session type,} \end{aligned}$$

The operator $\circ$ ensures that each linearly-typed channel x occurs either in P or in Q but never in both, to avoid races. Rule (T-Var) says that a variable x has type T if the pair $x:T$ is in the environment Γ. Rule (T-Res) states that $(\boldsymbol{\nu}xy)P$ is well-typed if P is well-typed and the co-variables have dual types. Type duality $\perp$ is standard, as $\perp_c$ in [12], and relates two types that describe compatible behaviours (for example, inputs are matched with outputs and selections are matched with compatible branchings). Rules (T-In) and (T-Out) type, respectively, the receiving and the sending of a value; these rules deal with both linear and unrestricted types. Rule (T-Brch) types an external choice on channel x, checking that each branch continuation P_i follows the respective type continuation in the type of x. Similarly, rule (T-Sel) types an internal choice communicated on channel x by checking the possible continuations. The operator $+$ is used to update the type of a variable with the continuation type in order to enable typing after an input (or branch) or an output (or select) operation has occurred. Rules (T-RecP) and (T-RecV) are standard, and type respectively a recursive process and a recursive process variable.

The type system above guarantees type preservation.

Theorem 1 (Preservation [25]). *If $\Theta; \Gamma \vdash P$ and $P \to Q$ then $\Theta; \Gamma \vdash Q$.*

Remark 1 (Type Safety and Well-Formedness). In [25], type safety is defined using an auxiliary definition of well-formedness. Intuitively, all enabled actions in a well-formed process must be such that (i) guards of conditionals are boolean values; (ii) unrestricted channels are used in the same way; (iii) actions on co-variables form a redex. Well-formedness is then guaranteed to follow from well-typedness, but only in the case of closed processes due to a technicality with condition (i). In our setting without conditionals, condition (i) does not apply and therefore well-typed processes are always well-formed.

3 Lock-freedom and Progress

3.1 Definitions

Lock-Freedom. Intuitively, a process is lock-free if any communication action that becomes active during execution is eventually consumed. Below, we assume that reduction sequences are fair, as formalised in [15].

Definition 1 (Lock-Freedom for Sessions). *A process P_0 is* lock-free *if for any fair reduction sequence $P_0 \to P_1 \to P_2 \to \ldots$, we have that*

1. *$P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x!\langle v\rangle.Q \mid R)$, for $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\boldsymbol{\nu}\widetilde{x'y'})(x!\langle v\rangle.Q \mid y?(z).R_1 \mid R_2)$ and $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{x'y'})(Q \mid R_1[v/z] \mid R_2)$;*
2. *$P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft l_j.Q \mid R)$, for some $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\boldsymbol{\nu}\widetilde{x'y'})(x \triangleleft l_j.Q \mid y \triangleright \{l_k : R_k\}_{k \in I \cup \{j\}} \mid S)$ and $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{x'y'})(Q \mid R_j \mid S)$.*

For simplicity, above we have omitted the dual cases for input and branching.

Progress. Before giving the formal definition of progress, we first need to introduce some auxiliary definitions. We start with the definition of characteristic process, which is the simplest process that can inhabit a type:

Definition 2 (Characteristic Process). *Given a type T, its* characteristic process $\llparenthesis T \rrparenthesis_g^x$ *is inductively defined on the structure of T as:*

$$
\begin{array}{rl}
(\textsc{inVal}) & \llparenthesis q?\mathbf{1}.U \rrparenthesis_g^x = x?(y).\llparenthesis U \rrparenthesis_g^x \\
(\textsc{outVal}) & \llparenthesis q!\mathbf{1}.U \rrparenthesis_g^x = x!\langle \mathsf{unit} \rangle.\llparenthesis U \rrparenthesis_g^x \\
(\textsc{inSess}) & \llparenthesis q'?(qp).U \rrparenthesis_g^x = x?(y).(\llparenthesis U \rrparenthesis_g^x \mid \llparenthesis qp \rrparenthesis_g^y) \\
(\textsc{outSess}) & \llparenthesis q'!(qp).U \rrparenthesis_g^x = (\boldsymbol{\nu} zw)(x!\langle z \rangle.(\llparenthesis U \rrparenthesis_g^x \mid \llparenthesis \overline{qp} \rrparenthesis_g^w)) \\
(\textsc{inSum}) & \llparenthesis q\&\{l_i : (q_i p_i)_i\}_{i \in I} \rrparenthesis_g^x = x \triangleright \{l_i : \llparenthesis q_i p_i \rrparenthesis_g^x\}_{i \in I} \\
(\textsc{outSum}) & \llparenthesis q \oplus \{l_i : (q_i p_i)_i\}_{i \in I} \rrparenthesis_g^x = x \triangleleft \{l_i : \llparenthesis q_i p_i \rrparenthesis_g^x\}_{i \in I} \\
(\textsc{end}) & \llparenthesis \mathsf{end} \rrparenthesis_g^x = \mathbf{0} \\
(\textsc{recVar}) & \llparenthesis \mathbf{t} \rrparenthesis_g^x = g(\mathbf{t}) \\
(\textsc{rec}) & \llparenthesis \mu \mathbf{t}.T \rrparenthesis_g^x = \mathbf{rec}X.\llparenthesis T \rrparenthesis_{g,\{\mathbf{t} \mapsto X\}}^x
\end{array}
$$

Above, the characteristic process $\llparenthesis T \rrparenthesis_g^x$ is a process that implements type T on session channel x; function g maps type variables for recursion in T to the recursion variables in the process that implements them. The definition above is a refinement of that in [4, 8], with two modifications. The first is an extension to recursive processes. The second is that our rule (OutSum) produces a process that may select any label among those reported in the selection type. Previous work, instead, limited the characteristic process to selecting only the first label. We will show that this difference directly refines our definition of progress.

We now define *catalysers*, execution contexts that contain only restrictions and characteristic processes:

Definition 3 (Catalyser). *A catalyser $\mathcal{C}[\cdot]$ is a context such that:*

$$
\mathcal{C}[\cdot] ::= \ [\cdot] \quad \mid \quad (\boldsymbol{\nu} xy)\mathcal{C}[\cdot] \quad \mid \quad \mathcal{C}[\cdot] \mid \llparenthesis qp \rrparenthesis_g^x
$$

Example 1. The following context $\mathcal{C}[\cdot]$ is a catalyser obtained by composing the characteristic processes P_1 and P_2 respectively of the types $T_1 = \mathsf{un}\ ?(!\mathbf{1}.\mathsf{end}).\mathsf{un}\,\mathsf{end}$ and $T_2 = \mathsf{lin} \oplus \{l_1 : \mathsf{end}, l_2 : !\mathbf{1}.\mathsf{end}\}$:

$$
\begin{array}{rcl}
\mathcal{C}[\cdot] & = & (\boldsymbol{\nu} wx)(\boldsymbol{\nu} uy)([\cdot] \mid P_1 \mid P_2) \\
P_1 & = & x?(z).(z!\langle \mathsf{unit} \rangle.\mathbf{0} \mid \mathbf{0}) \\
P_2 & = & y \triangleleft \{l_1.\mathbf{0},\ l_2.y!\langle \mathsf{unit} \rangle.\mathbf{0}\}
\end{array}
\qquad \square
$$

The duality operator $\bowtie$ is a relation over processes with respective co-actions.

Definition 4 ($\bowtie$). *The* duality $\bowtie_{\{x,y\}}$ *is defined as follows:*

$$
\begin{array}{l}
x!\langle v \rangle.P \bowtie_{\{x,y\}} y?(z).Q \\
x \triangleleft \{l_i.P_i\}_{i \in I} \bowtie_{\{x,y\}} y \triangleright \{l_i : Q_i\}_{i \in I}
\end{array}
$$

As last auxiliary definition for progress, we define evaluation contexts. An *evaluation context* (or context, for short) $\mathcal{E}[\cdot]$ is a process with holes such that:

$$
\mathcal{E}[\cdot] ::= \quad [\cdot] \mid P \quad \mid \quad (\boldsymbol{\nu} xy)\mathcal{E}[\cdot] \quad \mid \quad \mathcal{E}[\cdot] \mid \mathcal{E}[\cdot] \quad \mid \quad \mathbf{rec}X.\mathcal{E}[\cdot]
$$

We are now ready to give the formal definition of progress.

Definition 5 (Progress). *A process P has* progress *if for all catalysers $\mathcal{C}[\cdot]$ such that $\mathcal{C}[P]$ is well-typed, $\mathcal{C}[P] \rightarrow^* \mathcal{E}[R]$ (where R is an input or an output) implies that there exist $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{C}'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu} xy)$ is a restriction in $\mathcal{C}'[\mathcal{E}[R]]$.*

Remark 2. Our formulation of progress is inspired by [4, 8]. However, our catalysers are different when it comes to selection. Consider the following example:

$$P \;=\; x \triangleright \left\{ \begin{array}{l} l_1 : \mathbf{0}, \\ l_2 : (\boldsymbol{\nu} y_1 y_2)(y_1!\langle \mathsf{unit} \rangle.y_2?(z).\mathbf{0}) \end{array} \right\}$$

Process P above offers branches l_1 and l_2 on x. If l_2 is chosen, then P gets stuck into a deadlock. In previous works, P has progress since only the first branch is checked (l_1 in our example). This is unsatisfactory, because P may be composed with other systems that select branch l_2, and then get stuck. Instead, process P does not satisfy Definition 5 since all branches are checked by our characteristic processes (Definition 2, rule (outSum)).

3.2 Properties

We now move to presenting the relationship between progress and lock-freedom. For well-typed closed terms, i.e., well-typed processes with no free variables, the properties of lock-freedom and progress coincide. Intuitively, this is because closed processes cannot interact with catalysers and the latter are always lock-free by construction. We formalise this aspect in the theorem below.

Theorem 2 (Lock-freedom $\Leftrightarrow$ Closed Progress). *Let P be a well-typed closed process. Then, P is lock-free if and only if P has progress.*

We now switch to a more general setting, i.e., processes that can be open. Differently than in the case of closed terms, the definitions of lock-freedom and progress do not coincide for open terms. For example, consider the process:

$$P \;=\; x!\langle \mathsf{unit} \rangle.x?(z).\mathbf{0} \tag{1}$$

Process P above has progress, since we can find a catalyser for reducing it, but it is not lock-free as it does not respect Definition 1.

Even if progress and lock-freedom do not coincide for open terms, we can still formally relate the two properties. The key idea for reaching this objective is to wrap an open term using catalysers until all sessions are closed, a procedure we call *typed closure*. We formally define typed closure below.

Definition 6 (Typed Closure). *Let $\Gamma \vdash P$. Then, the typed closure of P, denoted by $\mathsf{tclose}(P)$, is the process $\mathcal{C}[P]$ where*

$$\mathcal{C}[\cdot] = (\boldsymbol{\nu} \widetilde{xy})\big(\; [\cdot] \mid \prod_{\forall x_i : T_i \in \Gamma} (\!|T_i|\!)^{y_i}_{\emptyset} \big)$$

Above, all x_i in $\widetilde{xy}$ correspond exactly to the domain of Γ. Typed closure is the identity for closed processes, since those are typed with empty environments.

Example 2. Consider the previous open process P in (1):

$$P = x!\langle \mathsf{unit} \rangle.x?(z).\mathbf{0}$$

P can be typed with environment $\Gamma = x : !\mathbf{1}.?\mathbf{1}.\mathsf{end}$. Its typed closure is then:

$$\mathsf{tclose}(P) = (\boldsymbol{\nu}xy)(P \mid y?(w).y!\langle \mathsf{unit} \rangle.\mathbf{0}) \qquad \square$$

Typed closure preserves typability:

Proposition 1 (Closure preserves typability). *If $\Gamma \vdash P$ then $\emptyset \vdash \mathsf{tclose}(P)$.*

We are now going to present one of the major properties in our technical development, which will be crucial in establishing our main results: a process has progress if and only if its typed closure reduces to terms where actions at the top level can always be matched with their respective co-actions in a parallel subterm. Intuitively, this is because the catalysers in the typed closure of a process are exactly all those ones needed for further reducing the process as required by the definition of progress (Definition 5).

Lemma 1 (From Closure to Progress). *Let P be well-typed. Then, P has progress if and only if $\mathsf{tclose}(P) \rightarrow^* \mathcal{E}[R]$ (where R is an input or an output process) implies that there exist $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{E}[R] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu}xy)$ is a restriction in $\mathcal{E}[R]$.*

Thanks to Lemma 1, we are able to establish that checking progress for a process P is equivalent to checking the progress property for its closure:

Theorem 3 (Closure Progress ⇔ Progress). *If P is well-typed then $\mathsf{tclose}(P)$ has progress if and only if P has progress.*

We are finally able to link progress and lock-freedom in the general case of open processes: a well-typed process has progress if and only if its typed closure is lock-free. This is an immediate consequence of Theorem 2 and Theorem 3.

Theorem 4 (Progress ⇔ Closed Lock-Free). *If P is well-typed then P has progress if and only if $\mathsf{tclose}(P)$ is lock-free.*

4 Untyped Closure

4.1 Definitions

So far, we have investigated the notion of progress and its connection with lock-freedom by building on top of the typing discipline for the π-calculus with sessions. Typing is useful for defining the adequate contexts for checking progress, namely our catalysers. In this section, we show that adequate contexts can be defined without the need for a typing discipline. Such contexts are based solely on the structure of processes, and lead to a more general notion of progress. Below, we introduce the notion of *co-process*:

Definition 7 (Co-Process). *Given a process P, its* co-process $\mathsf{co}[P]_f$ *is inductively defined as:*

$$\mathsf{co}[x?(y).P]_f = \begin{cases} \mathsf{co}[P]_f & \textit{if } x \notin \mathsf{dom}(f) \\ (\boldsymbol{\nu} zw)(f_x!\langle z\rangle.\mathsf{co}[P]_{f,y\mapsto w}) & \textit{if } x \in \mathsf{dom}(f),\ y \textit{ is a channel, } z, w \textit{ fresh} \\ f_x!\langle \mathsf{unit}\rangle.\mathsf{co}[P]_f & \textit{otherwise} \end{cases}$$

$$\mathsf{co}[x!\langle v\rangle.P]_f \ = \begin{cases} \mathsf{co}[P]_f & \textit{if } x \notin \mathsf{dom}(f) \\ f_x?(y).\mathsf{co}[P]_f & \textit{otherwise} \end{cases}$$

$$\mathsf{co}[(\boldsymbol{\nu} xy)P]_f = \mathsf{co}[P]_f \quad \big(\textit{if } x, y \notin \mathsf{dom}(f)\big) \qquad\qquad \mathsf{co}[X]_f = X$$

$$\mathsf{co}[\mathbf{rec}X.P]_f = \mathbf{rec}X.\mathsf{co}[P]_f \qquad \mathsf{co}[P \mid Q]_f = \mathsf{co}[P]_f \mid \mathsf{co}[Q]_f \qquad \mathsf{co}[\mathbf{0}]_f = \mathbf{0}$$

$$\mathsf{co}[x \triangleright \{l_i : P_i\}_{i\in I}]_f = \begin{cases} f_x \triangleleft \{l_i : \mathsf{co}[P_i]_f\}_{i\in I} & \textit{if } x \in \mathsf{dom}(f) \\ \sqcup\, \mathsf{co}[P_i]_f & \textit{otherwise} \end{cases}$$

$$\mathsf{co}[x \triangleleft \{l_i.P_i\}_{i\in I}]_f \ = \begin{cases} f_x \triangleright \{l_i : \mathsf{co}[P_i]_f\}_{i\in I} & \textit{if } x \in \mathsf{dom}(f) \\ \sqcup\, \mathsf{co}[P_i]_f & \textit{otherwise} \end{cases}$$

Roughly, the co-process $\mathsf{co}[P]_f$ of a process P is P with all its actions replaced with respective compatible co-actions. The function f is a renaming for variables. Intuitively, we use it for mapping free variables in P, which identify the open communication endpoints in P, to their respective co-variables in $\mathsf{co}[P]_f$. For an input $x?(y).P$, its co-process is: the co-process of the continuation P if x is not in f; the output of a fresh variable z if y is used as a channel in P (we distinguish channels in inputs using standard sorting from the π-calculus, omitted here); the output of a unit value otherwise. The rule for outputs is similar. For a restriction $(\boldsymbol{\nu} xy)P$, we check that the restricted names are not in f since their actions are already matched inside P. The cases of recursion, parallel, and the terminated process are simply homomorphisms. We assume that in $\mathsf{co}[P]_f$, any occurrence of recursion calls not guarded by actions, e.g., $\mathbf{rec}X.X$, are replaced with $\mathbf{0}$. Branching and selection are defined similarly to inputs and outputs whenever the subject of the communication is in f. Otherwise, since we cannot predict which choice will be made at run-time, we make use of the auxiliary operator $\sqcup$ to *merge* the behaviours in the different branches. We formally define $\sqcup$ below.

Definition 8 (Merge). *The merge operator $\sqcup$ is defined by the equations below.*

$$x \triangleright \{\widetilde{l : P}, \widetilde{l' : P'}\} \sqcup x \triangleright \{\widetilde{l : Q}, \widetilde{l'' : P''}\} = x \triangleright \{\widetilde{l : P \sqcup Q}, \widetilde{l' : P'}, \widetilde{l'' : P''}\}$$

$$x \triangleleft \{\widetilde{l : P}, \widetilde{l' : P'}\} \sqcup x \triangleleft \{\widetilde{l : Q}, \widetilde{l'' : P''}\} = x \triangleleft \{\widetilde{l : P \sqcup Q}\}$$

$$P \sqcup Q = P \quad \textit{if } P \equiv Q$$

We say that P and Q are mergeable, written $P\clubsuit Q$, whenever $P \sqcup Q$ is defined.

Using co-processes, we can define a new closure independent from types.

Definition 9 (Untyped Closure). *The untyped closure of* P, $\mathsf{uclose}(P)$, *is:*

$$(\boldsymbol{\nu}\widetilde{xf_x})(P \mid \mathsf{co}[P]_f)$$

where $\mathsf{dom}(f) = \mathsf{fn}(P)$.

Example 3. Untyped closure is not always defined. For example,

$$P = (\boldsymbol{\nu}xx')\big(x \triangleright \{l_1 : y \triangleleft l_3, l_2 : y!\langle v\rangle\} \mid x' \triangleleft \{l_1 : y' \triangleright l_3, l_2 : y'?(z)\}\big)$$

cannot be expressed as $P \equiv (\boldsymbol{\nu}\widetilde{xy})(Q \mid \mathsf{co}[Q]_f)$ because the merge operation given in Definition 8 cannot be defined. This is because y and y' perform once a selection and once an output, which cannot be merged together. □

For well-typed processes, untyped closure preserves typability:

Proposition 2. *If* P *is well-typed, then* $\mathsf{uclose}(P)$ *is well-typed.*

4.2 Adequacy of untyped closure

We conclude this section by showing that untyped closure is a conservative extension of typed closure, i.e., it preserves the same connection between lock-freedom and progress for well-typed processes. Technically, for well-typed processes, untyped closure and typed closure have equivalent behaviours. First, we show that for a typed process, the reductions of its untyped closure can mimic the reductions of its typed closure and vice versa. Below, we denote with $\mathsf{tclose}_0(P)$ the typed closure of P generated using the simplest output typing of P, namely if $\Gamma \vdash_0 P$ then all carried types in the output types of Γ are equal to end.

Lemma 2. *Let* P *be well-typed. Then,* $\mathsf{uclose}(P) \to \clubsuit\mathsf{uclose}(P')$ *iff* $\mathsf{tclose}_0(P) \to \mathsf{tclose}_0(P')$.

As a consequence of Lemma 2, we obtain that the untyped closure of a well-typed process is lock-free if and only if its typed closure is lock-free.

Theorem 5. *Let* P *be well-typed.* $\mathsf{uclose}(P)$ *is lock-free iff* $\mathsf{tclose}(P)$ *is lock-free.*

Proof (Sketch). By Lemma 2, we observe that if two processes can be merged then they are related by a strong typed bisimulation (cf. [5]). Then, the thesis follows by observing that $\mathsf{tclose}(P)$ is closed under reductions, and $\mathsf{uclose}(P)$ is closed under reductions up-to strong bisimulation (merging). □

From Theorem 4 in § 3.2, and Theorem 5 we conclude:

Corollary 1. *Let* P *be well-typed. If* $\mathsf{uclose}(P)$ *is lock-free, then* P *has progress.*

5 Progress through static analysis for lock-freedom

Our technical development reduced the problem of checking whether a process has progress to the problem of checking whether its closure (typed or untyped) is lock-free. A direct consequence of this result is that static analysis for lock-freedom can be lifted to static analysis for progress. In this section we show an example of how to apply this methodology, by using the typing discipline for lock-freedom in the standard π-calculus by Kobayashi [15].

The π-calculus. We report the syntax of the standard π-calculus [18] where standard choice is replaced by the $\mathbf{case}\, v\, \mathbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i\in I}$ constructor:

$$\begin{array}{rl} P, Q ::= & x!\langle\tilde{v}\rangle.P \quad | \quad x?(\tilde{y}).P \quad | \quad P \mid Q \quad | \quad \mathbf{0} \quad | \quad (\boldsymbol{\nu} x)P \\ & | \ \mathbf{case}\, v\, \mathbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i\in I} \quad | \quad X \quad | \quad \mathbf{rec} X.P \\ v ::= & x \quad | \quad \mathsf{unit} \quad | \quad l\_v \end{array}$$

The differences wrt to the syntax of the π-calculus with sessions are that restriction is now on a single variable and that there are no constructs for branching and selection. Values include variables and the unit value, as in the π-calculus with sessions, and also the labelled values l_v, used in the **case** process.

We report below the main reduction rules:

$$\begin{array}{ll} (\mathrm{R}\pi\text{- }\textsc{Com}) & x!\langle\tilde{v}\rangle.P \mid x?(\tilde{z}).Q \to P \mid Q[\tilde{v}/\tilde{z}] \\ (\mathrm{R}\pi\text{- }\textsc{Case}) & \mathbf{case}\, l_j\_v\, \mathbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i\in I} \to P_j[v/x_j] \quad j \in I \end{array}$$

The main difference wrt the π-calculus with sessions is that communications happen when sending and receiving actions have the same subject (the variable used as a channel for sending or receiving a value), and not when the two actions in question have different subjects that were linked by a shared restriction. Moreover, the communicating channels need not be restricted. For simplicity, we omit all the other rules, as well as the definition of the structural congruence relation $\equiv$ between standard π-calculus processes.

Kobayashi's Typing for Lock-Freedom. We briefly introduce Kobayashi's type system for checking lock-freedom in the standard π-calculus, from [15]. The syntax of types is defined as.

$$\begin{array}{lrl} \text{(actions)} & \alpha ::= & ? \mid ! \\ \text{(usage types)} & U ::= & \mathbf{0} \ | \ \alpha^o_c.U \ | \ U_1 \mid U_2 \ | \ U_1 \& U_2 \ | \ \mathbf{t} \ | \ \mu\mathbf{t}.U \\ \text{(channel types)} & T ::= & [\widetilde{T}]\ U \quad | \quad \langle l_i : T_i\rangle_{i\in I} \quad | \quad \mathbf{1} \end{array}$$

Types T include channel types $[\widetilde{T}]\ U$, the variant type $\langle l_i : T_i\rangle_{i\in I}$, and the unit type $\mathbf{1}$. In a channel type $[\widetilde{T}]\ U$, $\widetilde{T}$ are the types of the values transmitted over the channel and U is a usage type, describing how the channel is used. Usage types are similar to session types. Usage $\mathbf{0}$ describes a channel that cannot be used anymore (we will often omit it when not necessary); usage $\alpha^o_c.U$ describes a channel used for an input action (when $\alpha = ?$) or output action (when $\alpha = !$), and then used according to U. The annotations o and c, called tags, are natural numbers that indicate respectively the *obligation* and *capability* of an action, described below. Usage $U_1 \mid U_2$ describes a channel used according to U_1 and U_2 in parallel. Usage $U_1 \& U_2$ describes a channel used according to either U_1 or U_2. Usages $\mu \mathbf{t}.U$ and $\mathbf{t}$ indicate standard recursive types.

We describe the intuition behind reasoning with tags in usage types (see [15] for a full description). The tags o and c are abstract representations of time steps and describe dependencies between the usage of channels, corresponding to how actions on channels are interleaved in processes. Intuitively, an obligation o denotes a guarantee that its action will become available at most in time o, while a capability c denotes a requirement that a compatible co-action becomes available at most in time c. This information is crucial to ensure that processes do not get stuck, and it is checked to be consistent by Kobayashi's typing rules. As an example, we consider the rules for typing input and restriction:

$$\frac{\mathit{\Gamma}, \widetilde{y} : \widetilde{T} \vdash_{\mathtt{LF}} P}{x : [\widetilde{T}]\ ?^{0}_{c}\ ; \mathit{\Gamma} \vdash_{\mathtt{LF}} x?(\tilde{y}).P}\ (\text{LF-In}) \qquad \frac{\mathit{\Gamma}, x : [\widetilde{T}]\ U \vdash_{\mathtt{LF}} P \quad rel(U)}{\mathit{\Gamma} \vdash_{\mathtt{LF}} (\boldsymbol{\nu} x)P}\ (\text{LF-Res})$$

Rule (LF-In) states that the $x?(\tilde{y}).P$ is well-typed if x is a channel used in input with obligation 0. Moreover, the operator ; raises (increases by one) the obligations of the other channels in $\mathit{\Gamma}$ in the conclusion of the rule, in order to reflect that actions inside process P are prefixed by an input action and will thus become available later. Rule (LF-Res) is the key rule for establishing lock-freedom; it states that the restriction of a name x in process P is well-typed if x is used *reliably* in P. The notion of reliability of a usage is as follows. A usage U is said to be reliable, denoted with $rel(U)$, if after any step, whenever it contains an action (input or output) having capability tag c, it also contains the co-action with an obligation tag *at most* c. This means that the guarantee that the action will become available is at most the requirement for its availability (we refer the reader to [15] for the formal definition of $rel(U)$).

Kobayashi's type system guarantees lock-freedom:

Theorem 6 (Lock-Freedom [15]). *If* $\mathit{\Gamma} \vdash_{\mathtt{LF}} P$ *and* $rel(\mathit{\Gamma})$*, then* P *is lock-free.*

Above, $rel(\mathit{\Gamma})$ checks $rel(U)$ for all the usage types in $\mathit{\Gamma}$.

From the above theorem, we immediately get the following corollary:

Corollary 2. $\emptyset \vdash_{\mathtt{LF}} P$ *implies that* P *is lock-free.*

Encoding. Processes in the π-calculus with sessions can be translated to equivalent processes in the standard π-calculus, using the encoding $[\![-]\!]_f$ presented in [9]. Intuitively, such encoding transforms each action on sessions in the original process into an action on a linear channel in the standard π-calculus. We report a selection of the rules defining $[\![-]\!]_f$ in Fig. 5.

The parameter f renames the variables involved in a communication in order to simulate the structure of sessions using linear channels that are used exactly once. For example, in (E-Output) a new channel c is created and sent along

$$
\begin{array}{rll}
[\![x!\langle v\rangle.P]\!]_f = & (\boldsymbol{\nu} c)f_x!\langle v,c\rangle.[\![P]\!]_{f,x\mapsto c} & \text{(E-Output)}\\
[\![x?(y).P]\!]_f = & f_x?(y,c).[\![P]\!]_{f,x\mapsto c} & \text{(E-Input)}\\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f = & f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\ \{l_i\_(c) \triangleright [\![P_i]\!]_{f,x\mapsto c}\}_{i\in I} & \text{(E-Branching)}\\
[\![(\boldsymbol{\nu} xy)P]\!]_f = & (\boldsymbol{\nu} c)[\![P]\!]_{f,x\mapsto c,y\mapsto c} & \text{(E-Res)}
\end{array}
$$

Fig. 5. π-calculus with sessions, encoding to standard π-calculus.

with the original value v. The function f is then updated by mapping x to the new channel c, which is used in the continuation process. On the other hand, the process produced by rule (E-Input) performs the dual action by receiving the value of the communication and the new channel. (E-Branching) encodes the branching process by using the **case** process, after the guard of the **case** is received in input. Rule (E-Res) encodes the restriction $(\boldsymbol{\nu} xy)$ as $(\boldsymbol{\nu} c)$.

The encoding $[\![-]\!]_f$ is semantically correct:

Theorem 7 (Operational Correspondence [9]). *Let P be a process in the π-calculus with sessions. Then:*

- *If $P \to P'$ then $\exists Q$ such that $[\![P]\!]_f \to Q$ and $Q \hookrightarrow [\![P']\!]_f$, where $\hookrightarrow$ denotes a structural congruence extended with a* case normalisation*;*
- *If $[\![P]\!]_f \to\equiv Q$ then, $\exists\ P'$ such that $(\boldsymbol{\nu} xy)P \to (\boldsymbol{\nu} xy)P'$ and $Q \to^n\equiv [\![P']\!]_{f'}$, where $f_x = f_y$, $n \in \{1,2\}$ and f' is f updated after the reduction.*

From lock-freedom in the π-calculus to progress for sessions. We can finally present how to use our results in combination with Kobayashi's typing system for lock-freedom. First, from Theorem 7 we get that:

Corollary 3. *P in the π-calculus with sessions is lock-free iff $[\![P]\!]_f$ is lock-free.*

From our Corollaries 1 and 3, we can lift Kobayashi's analysis to progress in the π-calculus with sessions:

Theorem 8 (Typing Progress). *Let P be a well-typed process in the π-calculus with sessions. If $\emptyset \vdash_{\mathsf{LF}} [\![\mathsf{uclose}(P)]\!]_f$, then P has progress.*

Comparison. We conclude this section by comparing our approach with other static analysis for guaranteeing the progress property for session-based calculi in the literature [11, 4, 22]. For readability reasons, we omit some empty processes and restrictions of unused channels.

Example 4. The following process is lock-free and has progress:

$$(\boldsymbol{\nu} a_1a_2)\Big(\ a_1!\langle\mathsf{unit}\rangle\ \mid\ (\boldsymbol{\nu} b_1b_2)\big(\ b_1!\langle\mathsf{unit}\rangle\ \mid\ b_2?(y).a_2?(z)\ \big)\Big)$$

However, it is rejected by [22], since the type system presented therein does not distinguish between obligation and capability tags, but uses a single tag instead. If we consider its encoding in the π-calculus, we obtain the following process

$$(\boldsymbol{\nu} a)\Big(\ a!\langle\mathsf{unit}\rangle\ \mid\ (\boldsymbol{\nu} b)\big(\ b!\langle\mathsf{unit}\rangle\ \mid\ b?(y).a?(z)\ \big)\ \Big)$$

This process is accepted by Kobayashi's type system with types $a : !^0_1 \mid ?^1_0$ and $b : !^0_0 \mid ?^0_0$ and therefore our initial process has progress. □

Example 5. Consider the session process

$$(\boldsymbol{\nu} a_1 a_2)(\boldsymbol{\nu} b_1 b_2)\ \big(\, a_1?(x).\ b_1!\langle x\rangle.\ b_1?(y).\ a_1!\langle y\rangle \mid a_2!\langle \mathsf{unit}\rangle.\ b_2?(z).\ b_2!\langle \mathsf{unit}\rangle.\ a_2?(z)\,\big)$$

This process satisfies the progress property, but it is rejected by the type systems in [1] and [4]. This is because, in the two processes in parallel, there is a circular dependency between channels that such type systems cannot handle. Let us now consider its encoding in the π-calculus, given as the process:

$$(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)\left(\begin{array}{l} a?(x, c_1).\ (\boldsymbol{\nu} c_2)\Big(\ b!\langle x, c_2\rangle.\ c_2?(y).\ c_1!\langle y\rangle\ \Big) \mid \\ (\boldsymbol{\nu} c_1)\Big(a!\langle \mathsf{unit}, c_1\rangle.\ b?(z, c_2).\ c_2!\langle \mathsf{unit}\rangle.\ c_1?(z)\Big) \end{array}\right)$$

This process is correctly recognised as having progress by our technique, since it is well-typed in Kobayashi's type system. □

6 Conclusions and Future Work

In this paper we studied the relationship between the notions of progress and lock-freedom in the π-calculus with sessions, proving that they are strongly linked: progress can be thought of as a generalisation of lock-freedom to open processes. We have shown how to characterise progress using session types (typed closure) or the structure of processes (untyped closure). Our results can be used to lift static analyses for lock-freedom to the progress property. For example, we showed that reusing Kobayashi's type system [15] captures new interesting cases of processes that have progress that could not be recognised by previous work.

Future Work. As future work, we plan to extend our approach to multiparty sessions [14, 7]. For the multiparty setting, we need to investigate an extension of the encoding in [9] to a setting where sessions are established between more than two peers and messaging is asynchronous. It is not clear whether Kobayashi's usage types are expressive enough for handling such situations.

The works in [2, 26] use linear logic to type processes in the π-calculus with sessions. While these works guarantee lock-freedom, we conjecture that their techniques can be reused for progress, similarly to what we have done with Kobayashi's type system. We leave such an investigation as future work.

Kobayashi's type system comes with the reference implementation TyPiCal [24]. We are currently implementing a tool that allows to write processes in the π-calculus with sessions, checks that they are well-typed, and then uses the encoding in [9] for generating π-calculus code that can be analysed in TyPiCal.

References

1. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.

2. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
3. M. Carbone, O. Dardha, and F. Montesi. Progress as compositional lock-freedom, 2014. `http://www.dcs.gla.ac.uk/~ornela/my_papers/CDM-Extended.pdf`.
4. M. Carbone and S. Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE'10*, pages 13–27, 2010.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
6. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
7. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Global progress for dynamically interleaved multiparty sessions (long version), 2008. `http://www.di.unito.it/~dezani/papers/cdy12.pdf`.
8. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures of Computer Science*, To Appear.
9. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In D. D. Schreye, G. Janssens, and A. King, editors, *PPDP*, pages 139–150. ACM, 2012.
10. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *Proc. of POPL*, pages 435–446. ACM, 2011.
11. M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC*, volume 4912 of *LNCS*, pages 257–275. Springer, 2007.
12. S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, Nov. 2005.
13. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, Heidelberg, Germany, 1998. Springer-Verlag.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.
15. N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.
16. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, pages 233–247, 2006.
17. N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.
18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, Sept. 1992.
19. F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.
20. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
21. OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.
22. L. Padovani. From lock freedom to progress using session types. In Proc. of PLACES, 2013.
23. B. C. Pierce. *Types and programming languages.* MIT Press, 2002.
24. TYPICAL. Type-based static analyzer for the pi-calculus. `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`.
25. V. T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.
26. P. Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.

A Proof of Theorem 2 (Lock-freedom ⇔ Closed Progress)

Let P be a well-typed closed process. Then, P is lock-free if and only if P has progress.

We split the proof in two Lemmas, one for each direction in the Theorem.

Lemma 3 (Lock-freedom ⇒ Progress). *Let P be a well-typed closed process. Then, P lock-free implies that P has progress.*

Proof. The proof proceeds by contradiction. Let us assume that P does not have progress. Formally, it means that: there exists $\mathcal{C}[\cdot]$ such that $\mathcal{C}[P]$ is well-typed, $\mathcal{C}[P] \rightarrow^* \mathcal{E}[R]$ where R is an input or an output, and for all $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' it holds that $\mathcal{C}'[\mathcal{E}[R]] \not\rightarrow^* \mathcal{E}'[R][R']$ such that $R \bowtie_{\{x,y\}} R'$ where x and y are such that $(\boldsymbol{\nu} xy)$ is a restriction in $\mathcal{C}'[\mathcal{E}[R]]$. Instead, to reach a contradiction, we show that there exists $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{C}'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ such that $R \bowtie_{\{x,y\}} R'$. Since P is closed it means that it does not communicate with any context $\mathcal{C}[\cdot]$ it is inserted in. Hence, $\mathcal{C}[P] \rightarrow^* \mathcal{E}[R]$ means that reductions have occurred either in the catalyser $\mathcal{C}$ or in P, separately. Let now $\mathcal{C}'[\cdot] = [\cdot]$. We show that $\mathcal{E}[R] \rightarrow^* \mathcal{E}'[R][R']$. Since P is lock-free, by definition $P \rightarrow^* P_i$ and for some $n \geq i$, $P_i \rightarrow^* P_n$ and P_n has both action and co-action on some channels $(\boldsymbol{\nu} x'y')$. Notice that P_i is a subprocess of $\mathcal{E}[R]$ and hence P_n is a subprocess of $\mathcal{E}'[R][R']$ where R and R' are the action and co-action that have come up at the top level in the reduction under the restriction $(\boldsymbol{\nu} x'y')$ and let $x = x', y = y'$.

Lemma 4 (Progress ⇒ Lock-freedom). *Let P be a well-typed closed process. Then, P has progress implies P lock-free.*

Proof. From the definition of progress we know that for all catalysers $\mathcal{C}[\cdot]$, $\mathcal{C}[P] \rightarrow^* \mathcal{E}[R]$. In particular, this holds also for the empty catalyser $[\cdot]$. Hence, $P \rightarrow^* \mathcal{E}[R] \equiv P_i$. Here we can assume, without any loss of generality (the other cases are trivial) that R is an input or an output process. Furthermore, we know that there exist $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{C}'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu} xy)$ is a restriction in $\mathcal{C}'[\mathcal{E}[R]]$. Since P is closed, P_i is also closed and this means that it does not communicate with any catalyser it is inserted in. Hence, $\mathcal{C}'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ means that reductions have occurred either in the catalyser $\mathcal{C}'$ or in $\mathcal{E}[R]$, separately. Notice that R is part of $\mathcal{E}[R] \equiv P_i$, and since R occurs in the reductum $\mathcal{E}'[R][R']$ together with its counterpart R', it means that $P_i \rightarrow^* P_n$ where P_n is a subprocess of $\mathcal{E}'[R][R']$, and the communication occurs over $(\boldsymbol{\nu} xy)$. We conclude by applying the definition of lock-freedom.

B Proof of Lemma 1 (From Closure to Progress)

Let P be well-typed. Then, P has progress if and only if $\mathsf{tclose}(P) \to^* \mathcal{E}[R]$ *(where R is an input or an output process) implies that there exist $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{E}[R] \to^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu}xy)$ is a restriction in $\mathcal{E}[R]$.*

Proof. We split the proof into two cases.

only if. Follows immediately by the definitions of progress and $\mathsf{tclose}(P)$.

if. Let $\mathcal{C}[\cdot]$ be a catalyser such that $\mathcal{C}[P]$ is well-typed. Intuitively, any catalysers can be written by splitting the processes put in parallel with P in two: the ones that implement and the ones that do not implement the counterparts of sessions in P; formally:

$$\begin{aligned}
&\mathcal{C}[\cdot] \equiv (\boldsymbol{\nu}\widetilde{xy})\big([\cdot] \mid Q_1 \mid Q_2\big)\\
&Q_1 = \textstyle\prod_{x_j:T_j \notin \Gamma} (\!|T_j|\!)^{y_j}_g\\
&Q_2 = \textstyle\prod_{x_i:T_i \in \Gamma'} (\!|T_i|\!)^{y_i}_g \qquad \text{where } \Gamma' \subseteq \Gamma
\end{aligned}$$

Moreover, from the definition of $\mathsf{tclose}(P)$ we know that:

$$\begin{aligned}
&\mathsf{tclose}(P) \equiv (\boldsymbol{\nu}\widetilde{xy}')\big(P \mid Q_2 \mid Q_3\big)\\
&Q_3 = \textstyle\prod_{x_i:T_i \in \Gamma \setminus \Gamma'} (\!|T_i|\!)^{y_i}_g
\end{aligned}$$

Since $\mathcal{C}[P]$ is well-typed, we know that Q_1 cannot interact with P nor with Q_2; therefore we have only three possible cases for the derivation of $\mathcal{C}[P] \to^* \mathcal{E}''[R]$: (i) $P \to^* P'$; (ii) $(\boldsymbol{\nu}\widetilde{xy})Q_1 \to^* (\boldsymbol{\nu}\widetilde{xy})Q_1'$; (iii) $(\boldsymbol{\nu}\widetilde{xy})(P \mid Q_2) \to^* (\boldsymbol{\nu}\widetilde{xy})(P' \mid Q_2')$. We discuss the cases.

(i) For this case, we know that $\mathsf{tclose}(P) \to^* \mathsf{tclose}(P') \equiv \mathcal{E}[R]$ and $\mathcal{C}[P] \to^* \mathcal{C}[P']$. We now choose tclose as catalyser for $\mathcal{C}[P']$; therefore: $\mathsf{tclose}(\mathcal{C}[P']) \equiv \mathcal{C}[\mathsf{tclose}(P')] \equiv (\boldsymbol{\nu}\widetilde{xy}'')(\boldsymbol{\nu}\widetilde{xy})\big(P' \mid Q_1 \mid Q_2 \mid Q_3\big)$ where $\widetilde{xy}''$ are the free names in the typing of $\mathcal{C}[P']$. Since, by hypothesis, $\mathsf{tclose}(P') \to^* \mathcal{E}'[R][R']$ we also know that: $\mathsf{tclose}(\mathcal{C}[P']) \to^* \mathcal{C}[\mathcal{E}'[R][R']]$ and the thesis follows.

(ii) $(\boldsymbol{\nu}\widetilde{xy})Q_1 \to^* (\boldsymbol{\nu}\widetilde{xy})Q_1'$. This means that $\mathcal{C}[P] \to^* \mathcal{C}'[P]$, since only the catalyser reduces. We now choose tclose as catalyser for $\mathcal{C}'[P]$; therefore: $\mathsf{tclose}(\mathcal{C}'[P]) \equiv \mathcal{C}'[\mathsf{tclose}(P)] \to^* \mathcal{C}'[\mathcal{E}[R]]$ and the thesis follows by applying the hypothesis for $\mathsf{tclose}(P)$.

(iii) $(\boldsymbol{\nu}\widetilde{xy})(P \mid Q_2) \to^* (\boldsymbol{\nu}\widetilde{xy})(P' \mid Q_2')$. This means that $\mathcal{C}[P] \to^* \mathcal{C}'[P'] \equiv (\boldsymbol{\nu}\widetilde{xy})\big(P' \mid Q_1 \mid Q_2'\big)$, since both the catalyser and P reduce. By hypothesis, $\mathsf{tclose}(P) \to^* \mathcal{E}[R]$ and since the closure gives to P its missing counterpart, it means that P and Q_2 can communicate, hence $\mathcal{E}[R] \equiv (\boldsymbol{\nu}\widetilde{xy}')\big(P' \mid Q_2' \mid Q_3\big)$. We know that $\mathcal{E}[R] \to^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu}xy)$ is a restriction in $\mathcal{E}[R]$. Let $\mathcal{C}''[\cdot]$ be the catalyser for $\mathcal{C}'[P']$ defined as: $\mathcal{C}''[\cdot] \equiv (\boldsymbol{\nu}\widetilde{xy}')\big([\cdot] \mid Q_3\big)$. Then $\mathcal{C}''[\mathcal{C}'[P']] \equiv (\boldsymbol{\nu}\widetilde{xy}')(\boldsymbol{\nu}\widetilde{xy})\big(P' \mid Q_1 \mid Q_2' \mid Q_3\big)$and the thesis follows by applying the hypothesis for $\mathsf{tclose}(P)$. □

C Proof of Lemma 2

Let P be well-typed. Then, $\mathsf{uclose}(P) \to \clubsuit\mathsf{uclose}(P')$ *if and only if* $\mathsf{tclose}_0(P) \to \mathsf{tclose}_0(P')$*.*

Proof. We observe that if the reduction is such that $P \to P'$ then the thesis follows trivially. Hence, we only consider the case in which P actually interacts with $\mathsf{co}[P]_f$/catalyser. We need to show that

$$(\boldsymbol{\nu}\widetilde{xf_x})(P \mid \mathsf{co}[P]_f) \;\to\; (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}rs)(P' \mid \mathsf{co}[P']_{f,r\mapsto s})$$

if and only if

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})\big(\; P \mid (\!|T|\!)^{f_y}_g \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g\big) \;\to\\ (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}rs)\big(\; P' \mid (\!|T'|\!)^{f_y}_g \mid \;R \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g\big)\end{array}$$

where $T \;\to T'$. The proof proceeds by structural induction on P. Below, we report the most relevant cases.

- **Case** $P = (\boldsymbol{\nu}zw)Q$, only if). For this case, our *assumption* is

  $$(\boldsymbol{\nu}\widetilde{xf_x})((\boldsymbol{\nu}zw)Q \mid \mathsf{co}[Q]_f) \;\to\; (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}rs)((\boldsymbol{\nu}zw)Q' \mid \mathsf{co}[Q']_f)$$

  where $z, w \not\in f$, noting that since $P \neq P'$, it must be the case that $P' = (\boldsymbol{\nu}zw)Q'$ for some Q' with the same free names as Q. On the other hand, our *thesis* states that

  $$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})\big(\;(\boldsymbol{\nu}zw)Q \mid (\!|T|\!)^{f_y}_g \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g\big) \;\to\\ (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}rs)\big(\;(\boldsymbol{\nu}zw)Q' \mid (\!|T'|\!)^{f_y}_g \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g \;\mid R\big)\end{array}$$

  Our *induction hypothesis* (**IH**) says that if:

  $$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}zz')(\boldsymbol{\nu}ww')(Q \mid \mathsf{co}[Q]_{f,z\mapsto z',w\mapsto w'}) \;\to\\ \quad(\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu}zz')(\boldsymbol{\nu}ww')(\boldsymbol{\nu}rs)(Q' \mid \mathsf{co}[Q']_{f,z\mapsto z',w\mapsto w'})\end{array} \tag{2}$$

  then

  $$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x}, zz', ww')(Q \mid (\!|T_z|\!)^{z'}_g \mid (\!|T_w|\!)^{w'}_g \mid (\!|T|\!)^{f_y}_g \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g) \;\to\\ \quad(\boldsymbol{\nu}\widetilde{xf_x}, zz', ww')(Q' \mid (\!|T_z|\!)^{z'}_g \mid (\!|T_w|\!)^{w'}_g \mid (\!|T'|\!)^{f_y}_g \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)^{f_{x_i}}_g \;\mid R')\end{array} \tag{3}$$

  Our assumption implies the reduction (2) since such reduction is not on the pair rs, and therefore, by applying the induction hypothesis, (3) holds.
  Now, we can observe that the reduction in (3) must be a synchronisation on yf_y between Q (owning y) and the catalyser $(\!|T|\!)^{f_y}_g$: this follows from the semantics of our calculus and typing. Therefore, such a reduction must also be possible for the term where we pair r and s with a restriction, that is the case of our thesis.

- **Case** $P = (\boldsymbol{\nu} zw)Q$, if). This case is similar to the previous one, just by inverting hypotheses and theses.

- **Case** $P = y!\langle v\rangle.Q$. For this case, we need to show that

$$(\boldsymbol{\nu}\widetilde{xf_x})(y!\langle v\rangle.Q \mid f_y?(z).\mathsf{co}[Q]_f) \;\rightarrow\; (\boldsymbol{\nu}\widetilde{xf_x})(Q \mid \mathsf{co}[Q]_f)$$

  if and only if

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})\big(\; y!\langle v\rangle.Q \mid (\!|T|\!)_g^{f_y} \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)_g^{f_{x_i}}\big) \;\rightarrow\\ (\boldsymbol{\nu}\widetilde{xf_x})\big(\; Q \mid (\!|T'|\!)_g^{f_y} \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)_g^{f_{x_i}}\big)\end{array}$$

  We can see that this case is trivial by definition of catalyser.

- **Case** $P = y?(z).Q$. If y is a channel, then

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})(y?(z).Q \mid (\boldsymbol{\nu} rs)(f_y!\langle r\rangle.\mathsf{co}[Q]_f \mid \mathsf{co}[Q]_{z\mapsto s})) \;\rightarrow\\ (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu} rs)(Q[r/z] \mid \mathsf{co}[Q]_f \mid \mathsf{co}[Q]_{z\mapsto s})\end{array}$$

  if and only if

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{xf_x})\big(\; y?(z).Q \mid (\boldsymbol{\nu} rs)(f_y!\langle r\rangle.((\!|T'|\!)_g^y \mid (\!|T''|\!)_g^s)) \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)_g^{f_{x_i}}\big) \;\rightarrow\\ (\boldsymbol{\nu}\widetilde{xf_x})(\boldsymbol{\nu} rs)\big(\; Q[r/z] \mid (\!|T'|\!)_g^y \mid (\!|T''|\!)_g^s \mid \prod_{x_i:T_i\in\Gamma}(\!|T_i|\!)_g^{f_{x_i}}\big)\end{array}$$

  If y is not a channel, then it is trivial.

□