Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines*

Roberto Bruni[1] and Leonardo Gaetano Mezzina[2]

[1] Computer Science Department, University of Pisa, Italy
bruni@di.unipi.it
[2] IMT Lucca, Institute for Advanced Studies, Italy
leonardo.mezzina@imtlucca.it

Abstract. The notion of a session is fundamental in service-oriented applications, as it serves to separate interactions between clients and different instances of the same service, and to group together logical units of work. Recently, the Service Centered Calculus (SCC) has been proposed as a process calculus designed around the concept of a dyadic session between a service side and an invoker side, where interaction protocols and service orchestration can be conveniently expressed. In this paper we propose a generic type system to collect services' behaviours and then we fix a class of well-typed processes that are guaranteed to be deadlock free, in the sense that they either diverge by invoking new service instances or reach a normal form. The type system is based on previous research on traditional mobile calculi, here conveniently extended and simplified thanks to the neat discipline imposed by the linguistic primitives of SCC.

1 Introduction

The success of service orientation is attracting the interest of both industry and academy. On the one hand, important standardisation bodies and industrial consortia are developing the WS-* stack, targeting the engineering of web services technologies from a pragmatic perspective. The related documentation is often centred around common programming patterns: it is more focused on technical details of some case-studies, than on the overall methodology, leaving many ambiguities open. On the other hand, several efforts are posed on mathematical foundations, by developing formal languages and models tailored to service-oriented architectures. The main aim is to provide current standards with unambiguous semantics, but hopefully, tackling the scenarios from a more abstract perspective, the formalisation can lay the basis for sound service orchestration methodologies. Within this research thread, many process calculi have emerged ([6, 5, 17, 4, 2, 16], to cite a few), that are enhanced with service-specific primitives.

The aim of this paper is to study a type system for one of the above proposals, called Service Centered Calculus (SCC) [2]. More precisely, we study a calculus derived from SCC and from its refined variant CaSPiS [3] in which service invocation encompasses *one-way* and *request-response* protocols available in current WS-technology and allows for more sophisticated message exchanges, according to the protocol exposed by

* Research supported by the EU within the FET-GC II Integrated Project IST-2005-016004 Sensoria and by the Italian FIRB Project Tocai.it.

the service. The key feature is considering the messages exchanged between caller and callee as correlated, enclosed in special units of work, called *sessions*, and isolated from messages belonging to different invocations to the same service. Differently from other session languages inspired by pi-calculus [11, 12, 19, 8], object-orientation [9, 7] and correlation sets [17], here the programmers should not bother with the manipulation of sessions: they are created automatically, in a transparent manner, upon service invocation. In particular, in SCC the communication media for exchanging messages is always implicit and determined by the context surrounding active abstractions and concretions. For this reason, we allow service name mobility, but not session name mobility.

The automatic teller machine example in [12] can serve well to illustrate our approach to the typing of SCC. The ATM offers three options to choose from: deposit, withdraw and balance. Once the user chooses one option the ATM establishes a new direct connection with the bank to account for the operation. Afterwards, the result is returned by the ATM to the user who can choose another option. At the type system level, even if the connection with the bank is reiterated each time the user chooses an option, it is only necessary to check a single instance to guarantee safety of the communication, because each interaction belongs to a distinct session. For SCC we show that, by constraining the communication activities, well-typedness not only guarantees safety but a much stronger property such as deadlock freedom.

Since sessions can be nested but cannot be addressed explicitly in communication primitives, the language is endowed with children-to-parent communications and with in-session communications. These two communication patterns not only are expressive enough to encode lazy λ-calculus [2] but also makes it possible that typing a single instance of a session suffices to guarantee deadlock freedom of recursive processes such as the factorial service (see Example 2). Another feature of SCC is the presence of a pipe construct, inspired by Orc [6], an elegant language for structured orchestration. Pipelines offer a basic mechanism for composing processes: it is more general and better suited w.r.t. concurrency than sequential composition and it does not require the improper use of channels for a task that pertains to orchestration. As far as we know, our type system is the first one to address the direct typing of such a pipe primitive.

The resulting language is somehow too permissive to be dealt with using session types [11, 19, 12] directly, as they would require, e.g., each input in a session to be matched by only an output. This condition is violated (and consequently subject reduction does not hold) if, for example, in the presence of an input we introduce parallel outputs of different types. Our type system extends ordinary session types to work correctly with our language. Differently from [13], this permits each variable to be statically assigned a basic type or a service type (in the case of service name mobility).

The type system characterises a subclass of SCC processes typical of the service-oriented scenarios (e.g., service declarations are top level and replicated) for which we show the main theorem of this paper: we prove that these processes are deadlock free in the sense that they either diverge by invoking service instances and opening new sessions or reach a normal form in which only service declarations remain; that is, every client terminates its computation unless someone diverges, but the entire system cannot block on pending communications. The proof technique makes an extensive use of types to limit the number of possible cases, resulting less error prone.

A similar type system exists for SSCC (a variant of SCC based on named streams instead of pipelines) [16] that guarantees session safety. Streams introduce some sort of global buffers for extra-session communication and permit only a single type of values for each stream. Likewise, the type system described in [1] resembles ours but it deals asymmetrically with services and clients, by guaranteeing only client progress. Concerning deadlock freedom we use definitions in [14, 15] as main references for our definition, which is slightly different because tailored to the service-oriented scenario.

Synopsis. Section 2 introduces our SCC-like calculus. Section 3 presents the type system and the subject reduction result. Section 4 defines the class of initial processes and proves the main theorem: every well-typed initial process is deadlock free. Section 5 summarises the results and points out directions for further work. Due to space limitation proofs are just sketched or omitted.

2 Session Centered Calculus

2.1 SCC Overview: Room Reservation Example

Consider the following reservation service *reserve* for hotel rooms:

$$R \;=\; \mathit{reserve}.\Big(\; (\mathsf{double}).\texttt{(x,y)}.\langle\texttt{code(x,y)}\rangle \;+\; (\mathsf{single}).\texttt{(x)}.\langle\texttt{code(x,"")}\rangle \;\Big)$$

reserve offers two kinds of rooms, double or single, depending on the client choice. If the client after an invocation $\overline{\mathit{reserve}}$ sends the label double to the service then the service waits for a pair of names x and y (both of type *str*), and after receiving them, generate a numeric reservation code (type *int*) derived from x and y that is sent back to the client. Here code : $str \times str \rightarrow int$ is a function only available on service side.

$$\begin{array}{rcl} C & = & \overline{\mathit{reserve}}.\texttt{if}(\mathit{test})\ \texttt{then}\ \langle\mathsf{single}\rangle.\langle\texttt{"Bob"}\rangle.\texttt{(x).return x} \\ & & \qquad\qquad\quad\ \texttt{else}\ \langle\mathsf{double}\rangle.\langle\texttt{"Bob","Leo"}\rangle.\texttt{(y).return y} \end{array}$$

The above client, after invoking $\overline{\mathit{reserve}}$ and depending on some condition *test*, chooses between the two available options. The situation of the freshly established session r after the client's choice of double is the following:

$$(\nu\ r)\ \Big(\ r^{-} \triangleright \langle\texttt{"Bob","Leo"}\rangle.\texttt{(y).return y}\ \mid\ r^{+} \triangleright \texttt{(x,y)}.\langle\texttt{code(x,y)}\rangle\ \mid\ R\ \Big)$$

where the client protocol is running on the left (the session side r^{-} with negative polarity) and the service protocol on the right (the session side r^{+} with positive polarity). Abstractions (e.g., (x, y)) and concretions (e.g., $\langle$"Bob", "Leo"$\rangle$) running on opposite sides (r^{+} and r^{-}) of the same session (bound name r) can exchange data, leading to

$$(\nu\ r)\ \Big(\ r^{-} \triangleright \texttt{(y).return y}\ \mid\ r^{+} \triangleright \langle\texttt{code("Bob","Leo")}\rangle\ \mid\ R\ \Big)$$

Say code("Bob", "Leo") evaluates to 556047. Then, after another interaction, the client side ($r^{-} \triangleright$ return 556047) can return the result outside of r (to the parent session, if any). For example $\overline{\mathit{reserve}}.\langle\mathsf{single}\rangle.\langle\texttt{"Bob"}\rangle.(x).\texttt{return}\ x > y > Q$ invokes *reserve* and delivers the result in y to Q. As many instances of Q are spawn as the number of values that are issued. Since the execution of an invocation prefix opens a nested session, the conjunct use of return and pipe is the easiest way to continue the computation within the pre-existing session.

$$
\begin{array}{rlr}
P,Q,R ::= & \mathbf{0} & \text{(nil)} \\
\mid & s.P & \text{(service definition)} \\
\mid & \overline{v}.P & \text{(invocation)} \\
\mid & \texttt{if}\ v = v_1\ \texttt{then}\ P\ \texttt{else}\ Q & \text{(if-then-else)} \\
\mid & (\tilde{x}).P & \text{(tuple input)} \\
\mid & \langle\tilde{v}\rangle.P & \text{(values output)} \\
\mid & \Sigma_{i=1}^{n}(l_i).P_i & \text{(label-guarded sum)} \\
\mid & \langle l\rangle.P & \text{(label choice)} \\
\mid & \texttt{return}\ \tilde{v}.P & \text{(return)} \\
\mid & (\nu m)P & \text{(restriction)} \\
\mid & r^p \triangleright P & \text{(session)} \\
\mid & P > \tilde{x} > Q & \text{(pipe)} \\
\mid & P|Q & \text{(parallel)}
\end{array}
\qquad
\begin{array}{rlr}
v ::= & \mathsf{f}(\tilde{v}) & \text{(function call)} \\
\mid & x & \text{(variable)} \\
\mid & m & \text{(service/session)} \\
\mid & b & \text{(basic data value)} \\
\\
p,q ::= & + \mid - & \text{(polarities)}
\end{array}
$$

Fig. 1. Syntax of our service calculus

2.2 Syntax

The set of processes is defined by the grammar in Fig. 1. We let P, Q, R range over processes, s over service names, r over session names, m over both session and service names, l over labels, x over variables (for service names and data), and v over values, which include an elsewhere specified set of basic data and expressions (possibly with names, variables and functions). Tuples are denoted by $\tilde{\cdot}$. Operators are listed in Fig. 1 in decreasing order of precedence, e.g., $r^- \triangleright P > \tilde{x} > Q|R$ reads $((r^- \triangleright P) > \tilde{x} > Q)|R$.

As usual, $\mathbf{0}$ is the nil process, the trailing of $\mathbf{0}$ is often omitted, parallel composition is denoted by $P|Q$ and restriction by $(\nu m)P$. The construct $r^p \triangleright P$ indicates a generic session side with polarity p (taking values in $\{+, -\}$). Sessions are mostly intended as run-time syntax. In fact, differently from other languages that provide primitives for explicit session naming and creation, here all sessions could be built automatically, resulting in a more elegant and disciplined style of writing processes. A fresh session name r and two polarised session ends $r^- \triangleright P$ and $r^+ \triangleright Q$ are generated (on client and service sides, respectively) upon each service invocation $\overline{s}.P$ of the service $s.Q$. We say $r^- \triangleright P$ is the dual session side of $r^+ \triangleright Q$ and vice versa. As P and Q share a session, their I/O communications are directed toward the dual session side. We let p, q range over polarities and $\overline{p}, \overline{q}$ are the opposite polarities of p and q, where $\overline{+} = -$ and $\overline{-} = +$.

Labels l allow for expressing a choice on one side among a set of available options at the other side. The primitive `return` is used to output values to the parent session and the pipe $P > \tilde{x} > Q$ is a construct for on-side communications, i.e., for propagating values in the same side of a session. Pipe is inspired by Orc [6] to activate a fresh instance of Q on any value produced by P.

Processes are taken up to α-equivalence considering $(\tilde{x}).Q$ and $P > \tilde{x} > Q$ as binders for variables $\tilde{x}$ in Q and $(\nu m)P$ as the binder for m in P. The set $\mathsf{fn}(P)$ of free names of P is defined as expected. It is worth noting that the standard capture avoiding substitution $P[{}^{\tilde{v}}/_{\tilde{x}}]$, which replaces a tuple of variables with a tuple of values, assumes that variables cannot appear in certain positions (i.e., $x \triangleright P$ and $x.P$ are forbidden by the syntax).

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$
$$(\nu m_1)(\nu m)P \equiv (\nu m)(\nu m_1)P \qquad ((\nu m)P)|Q \equiv (\nu m)(P|Q) \;\text{ if } m \notin \mathsf{fn}(Q)$$
$$((\nu m)P) > \tilde{x} > Q \equiv (\nu m)(P > \tilde{x} > Q) \;\text{ if } m \notin \mathsf{fn}(Q) \qquad r^p \triangleright (\nu m)P \equiv (\nu m)(r^p \triangleright P) \;\text{ if } r \neq m$$
$$\mathbf{0} > \tilde{x} > P \equiv \mathbf{0} \qquad (P|Q) > \tilde{x} > R \equiv (P > \tilde{x} > R)|(Q > \tilde{x} > R)$$
$$(r^p \triangleright \mathbf{0}) > \tilde{x} > R \equiv r^p \triangleright \mathbf{0} \qquad r_1^p \triangleright (Q|r_2^q \triangleright \mathbf{0}) \equiv r_1^p \triangleright Q|r_2^q \triangleright \mathbf{0} \qquad (\nu r)(r^+ \triangleright \mathbf{0}|r^- \triangleright \mathbf{0}) \equiv \mathbf{0}$$

Fig. 2. Structural congruence

Each service definition is persistent (i.e., not consumed after an invocation) and available at top level (see Definition 4). For this reason, in the type system we shall give in Section 3, their protocols are not supposed to return any value to the parent.

2.3 Operational Semantics

We describe the semantics of our language by means of an LTS that exploits the structural congruence $\equiv$, which is the least one defined by the equations in Fig. 2. They include ordinary axioms about parallel and restriction, together with distributivity of parallel over pipes, and a few axioms for garbage collecting terminated session ends $r^p \triangleright \mathbf{0}$. We say that Q is at the *top level* in P if $P \equiv (\nu\tilde{m})Q|R$ for some $\tilde{m}$ and R.

Our transition system exploits the labels λ in Fig. 3. We write $\leftrightarrow$ to mean either $\leftarrow$ or $\rightarrow$. We write $(\tilde{m})\lambda$ to mean the label λ where the names $\tilde{m}$ become bound. The notions of bound names $\mathsf{bn}(\lambda)$, free names $\mathsf{fn}(\lambda)$ and names $\mathsf{n}(\lambda)$ of a label λ are defined as expected. We remark that α-conversion is not applicable to labels.

The semantics is given in the early style, which guesses the values and labels in the rules (In) and (Branch), respectively. Rule (Def) shows the replicated nature of the service and together with (Inv) creates two processes which are ready to communicate after that (SCom) creates a new shared common session. (SessionOut) accounts for the return of a value, which is converted in an output out of the current session when the session construct is traversed. (Session) marks with the name of the exchanging session each operation in that session. (Comm) permits both communication of basic values and service names. Extrusion is handled by (Open) and (Par), but thanks to (Equiv) restricted names can be moved to the top before communication and a closure rule is not necessary. On the other hand, side condition of rule (Par) is useful for session floating since r is bound in labels for the service invocation. Rule (Pipe) creates a new concurrent copy of process Q together with the residual $P' > \tilde{x} > Q$ in the case that P outputs a value. Rule (PipePass) makes a move in P if the action is not an output.

We shall write $P \xrightarrow{\lambda}$ if there is a Q such that $P \xrightarrow{\lambda} Q$.

3 Typing

The set of session types, $U, T, \ldots$, is defined by the grammar in Fig. 5. Session types express sequences of typed tuples of input and output. Internal choice $\oplus$ records all the choices at a certain point of a session, requested in the branches of a conditional process. External choice & records the types of all offered options.

$$
\begin{array}{rrcll}
\lambda ::= & s \Rightarrow r & \mid & s \Leftarrow r & \text{(service invocation / definition)} \\
\mid & \rightarrow \tilde{v} & \mid & \leftarrow \tilde{v} & \text{(value production / consumption)} \\
\mid & r^p :\rightarrow \tilde{v} & \mid & r^p :\leftarrow \tilde{v} & \text{(value production / consumption within } r\text{)} \\
\mid & \rightarrow l & \mid & \leftarrow l & \text{(choice selection / branching)} \\
\mid & r^p :\rightarrow l & \mid & r^p :\leftarrow l & \text{(choice selection / branching within } r\text{)} \\
\mid & \tau & \mid & r\tau & \text{(silent steps)} \\
\mid & \uparrow \tilde{v} & \mid & (m)\lambda & \text{(value return / extrusion)}
\end{array}
$$

Fig. 3. Labels of the transition system

(In) $(\tilde{x}).P \xrightarrow{\leftarrow \tilde{v}} P[^{\tilde{v}}/_{\tilde{x}}]$

(Out) $\langle \tilde{v} \rangle.P \xrightarrow{\rightarrow \tilde{v}} P$

(Comm) $$\frac{P \xrightarrow{r^p:\leftarrow \tilde{v}} P' \quad Q \xrightarrow{r^{\overline{p}}:\rightarrow \tilde{v}} Q'}{P|Q \xrightarrow{r\tau} (P'|Q')}$$

(Session) $$\frac{P \xrightarrow{\lambda} P' \quad \lambda \in \{\leftrightarrow \tilde{v}, \leftrightarrow l\}}{r^p \triangleright P \xrightarrow{r^p:\lambda} r^p \triangleright P'}$$

(Branch) $\Sigma_{i=0}^{n}(l_i).P_i \xrightarrow{\leftarrow l_i} P_i$

(Choice) $\langle l \rangle.P \xrightarrow{\rightarrow l} P$

(Select) $$\frac{P \xrightarrow{r^p:\rightarrow l} P' \quad Q \xrightarrow{r^{\overline{p}}:\leftarrow l} Q'}{P|Q \xrightarrow{r\tau} (P'|Q')}$$

(SessionPass) $$\frac{P \xrightarrow{\lambda} P' \quad \lambda \in \{\tau, r'\tau, r'^{p'} : \lambda'\}}{r^p \triangleright P \xrightarrow{\lambda} r^p \triangleright P'}$$

(Def) $$\frac{r \notin \mathsf{fn}(s.P)}{s.P \xrightarrow{(r)s \Leftarrow r} r^+ \triangleright P|s.P}$$

(Inv) $$\frac{r \notin \mathsf{fn}(\overline{s}.P)}{\overline{s}.P \xrightarrow{(r)s \Rightarrow r} r^- \triangleright P}$$

(SCom) $$\frac{P \xrightarrow{(r)s \Rightarrow r} P' \quad Q \xrightarrow{(r)s \Leftarrow r} Q'}{(P|Q) \xrightarrow{\tau} (\nu r)(P'|Q')}$$

(Ret) $\mathtt{return}\ \tilde{v}.P \xrightarrow{\uparrow \tilde{v}} P$

(SessionOut) $$\frac{P \xrightarrow{\uparrow \tilde{v}} P'}{r^p \triangleright P \xrightarrow{\rightarrow \tilde{v}} r^p \triangleright P'}$$

(NestInv) $$\frac{P \xrightarrow{(r')s \Rightarrow r'} P' \quad r \neq r'}{r^p \triangleright P \xrightarrow{(r')s \Rightarrow r'} r^p \triangleright P'}$$

(Res) $$\frac{P \xrightarrow{\lambda} P' \quad s \notin \mathsf{n}(\lambda)}{(\nu s)P \xrightarrow{\lambda} (\nu s)P'}$$

(Open) $$\frac{P \xrightarrow{\lambda} P' \quad \lambda \in \{\rightarrow \tilde{v}, r^p :\rightarrow \tilde{v}\} \wedge s \in \mathsf{n}(\tilde{v})}{(\nu s)P \xrightarrow{(s)\lambda} P'}$$

(SessRes) $$\frac{P \xrightarrow{r\tau} P'}{(\nu r)P \xrightarrow{\tau} (\nu r)P'}$$

(Pipe) $$\frac{P \xrightarrow{\rightarrow \tilde{v}} P'}{P > \tilde{x} > Q \xrightarrow{\tau} Q[^{\tilde{v}}/_{\tilde{x}}]|(P' > \tilde{x} > Q)}$$

(PipePass) $$\frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \rightarrow \tilde{v}}{P > \tilde{x} > Q \xrightarrow{\lambda} P' > \tilde{x} > Q}$$

(IfL) $$\frac{P \xrightarrow{\lambda} P'}{\mathtt{if}\ v = v\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\lambda} P'}$$

(IfR) $$\frac{v_1 \neq v \quad Q \xrightarrow{\lambda} Q'}{\mathtt{if}\ v = v_1\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\lambda} Q'}$$

(Par) $$\frac{P \xrightarrow{\lambda} P' \quad \mathsf{bn}(\lambda) \cap \mathsf{fn}(Q) = \emptyset}{P|Q \xrightarrow{\lambda} P'|Q}$$

(Equiv) $$\frac{P \equiv Q \quad Q \xrightarrow{\lambda} Q' \quad Q' \equiv P'}{P \xrightarrow{\lambda} P'}$$

Fig. 4. Operational semantics

$$
\begin{array}{rll}
T, U ::= & \mathtt{end} & \text{(no action)} \\
\mid & ?(S_1, \ldots, S_n).T & \text{(input of a tuple)} \\
\mid & !(S_1, \ldots, S_n).T & \text{(output of a tuple)} \\
\mid & \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(external choice)} \\
\mid & \oplus\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(internal choice)}
\end{array}
\qquad
\begin{array}{rll}
S ::= & [T] & \text{(session)} \\
\mid & \mathcal{B} & \text{(basic data types)}
\end{array}
$$

Fig. 5. Syntax of types

Sorts S can be either $[T]$ which represents a session with session type T or an elements of a given set of basic data types $\mathcal{B}$. By convention we denote $\tau_b \in \mathcal{B}$ the type of the basic value b. We shall assume that $int, str \in \mathcal{B}$.

Our set of typing rules is in Fig. 6. Type judgements for values take the form $\Gamma \vdash v : S$ where S is the sort of v. Type judgements for processes take the form $\Gamma \vdash P : U[T]$, where the type U represents the outputs of P to the parent session, while T is the type of admissible interactions for the current session. We shall refer to such types as *usages*. Sometimes we write $[T]$ as a shorthand for $\mathtt{end}[T]$. The *type environment* Γ is a finite partial mapping from variables and services to sorts and function types. The empty environment is annotated $\emptyset$. When $x \notin dom(\Gamma)$ we write $\Gamma, x : S$ for the environment obtained by extending Γ with the binding of x to S (the same holds for $m \notin dom(\Gamma)$).

The first four rules for values are standard and the signature of each used external function must be inserted in the environment as a functional type (rule (FuncV)) because they are not bound by processes.

The type of $\mathbf{0}$ in (Tzero) is $\mathtt{end}[\mathtt{end}]$ since no action is performed neither in the current session nor towards the parent session. Rule (Tdef) constrains the protocol of the service to be the same as that of the body process P and rule (Tinv) checks that the invoked service behaves in the dual manner with respect to the client. Here the dual of T, written $\overline{T}$ is inductively defined as:

$$
\begin{array}{lll}
\overline{\mathtt{end}} = \mathtt{end} & \overline{?(\tilde{S}).T} = !(\tilde{S}).\overline{T} & \overline{\&\{l_1 : T_1, \ldots, l_n : T_n\}} = \oplus\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\} \\
 & \overline{!(\tilde{S}).T'} = ?(\tilde{S}).\overline{T'} & \overline{\oplus\{l_1 : T_1, \ldots, l_n : T_n\}} = \&\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\}
\end{array}
$$

It can be readily observed that duality exchanges the role of ! with ? and of & with $\oplus$.

Rules (Tin), (Tout) and (Tret) insert the usage type in the correct place. The type for the input variable $\tilde{x}$ in rule (Tin) is not declared in the syntax, but it can be inferred with the help of the algorithm described in [18]. Rule (Tbranch) allows for considering a subset of possible branches; this subset can be chosen in a minimal way by letting it include the branches used by the dual sessions in the rule (Tchoice). In fact, the `if` construct allows to choose between many branches at the same time and also different clients can invoke the same service making their own choices, and rule (Tif) force all of them to agree on the least common set of choices that must be available at the other side. The two rules for parallel composition (TparL) and (TparR) allow parallel composition of two processes only if at least one does not have any action in the current session, i.e. if it has type $U[\mathtt{end}]$. Note instead that both P and Q are allowed to produce values upwards, in which case the operation $U \circ U'$ is defined only if all the atomic parts in U and U' are of the same kind, say $!(\tilde{S})$, and in that case $U \circ U' = U.U' = U' \circ U$.

$$\text{(Service)}\ \Gamma, s : S \vdash s : S \qquad \text{(Var)}\ \Gamma, x : S \vdash x : S \qquad \text{(BasV)}\ \frac{\tau_b \in \mathcal{B}}{\Gamma \vdash b : \tau_b} \qquad \text{(Func)}\ \frac{\Gamma \vdash v_1 : S_1 \ldots \Gamma \vdash v_n : S_n \quad \tau_b \in \mathcal{B}}{\Gamma, \mathsf{f} : S_1 \times \ldots \times S_n \to \tau_b \vdash \mathsf{f}(v_1, \ldots, v_n) : \tau_b}$$

$$\text{(Tzero)}\ \Gamma \vdash \mathbf{0} : \mathsf{end}[\mathsf{end}] \qquad \text{(Tdef)}\ \frac{\Gamma \vdash P : \mathsf{end}[T] \quad \Gamma \vdash s : [T]}{\Gamma \vdash s.P : \mathsf{end}[\mathsf{end}]} \qquad \text{(Tinv)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash v : [\overline{T}]}{\Gamma \vdash \overline{v}.P : \mathsf{end}[U]}$$

$$\text{(Tin)}\ \frac{\Gamma, \tilde{x} : \tilde{S} \vdash P : U[T]}{\Gamma \vdash (\tilde{x}).P : U[?(\tilde{S}).T]} \qquad \text{(Tout)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \langle\tilde{v}\rangle.P : U[!(\tilde{S}).T]} \qquad \text{(Tret)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma \vdash \mathtt{return}\ \tilde{v}.P : !(\tilde{S}).U[T]}$$

$$\text{(Tbranch)}\ \frac{I \subseteq \{1, \ldots, n\} \quad \forall i \in I \quad \Gamma \vdash P_i : U[T_i]}{\Gamma \vdash \Sigma_{i=0}^{n}(l_i).P_i : U[\&\{l_i : T_i\}]_{i \in I}} \qquad \text{(TparL)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma \vdash Q : U'[\mathsf{end}] \quad U'' = U \circ U'}{\Gamma \vdash P|Q : U''[T]}$$

$$\text{(TChoice)}\ \frac{l = l_i \in \{l_1, \ldots, l_n\} \quad \Gamma \vdash P : U[T_i]}{\Gamma \vdash \langle l\rangle.P : U[\oplus\{l_1 : T_1, \ldots, l_n : T_n\}]} \qquad \text{(TparR)}\ \frac{\Gamma \vdash P : U[\mathsf{end}] \quad \Gamma \vdash Q : U'[T] \quad U'' = U \circ U'}{\Gamma \vdash P|Q : U''[T]}$$

$$\text{(Tpipe)}\ \frac{\Gamma \vdash P : U[T] \quad \Gamma, \tilde{x} : \tilde{S} \vdash Q : U'[T'] \quad \mathsf{pipe}(U[T], U'[T'], \tilde{S}) = U''[T'']}{\Gamma \vdash P > \tilde{x} > Q : U''[T'']}$$

$$\text{(Tses)}\ \frac{\Gamma \vdash P : U[T]}{\Gamma, r : [T] \vdash r^{+} \rhd P : \mathsf{end}[U]} \qquad \text{(TsesI)}\ \frac{\Gamma \vdash P : U[T]}{\Gamma, r : [\overline{T}] \vdash r^{-} \rhd P : \mathsf{end}[U]}$$

$$\text{(Tnew)}\ \frac{\Gamma, m : S \vdash P : U[T] \quad \mathsf{exists}(m, P)}{\Gamma \vdash (\nu m)P : U[T]} \qquad \text{(Tif)}\ \frac{\Gamma \vdash v_i : S_i \quad i = 1, 2 \quad \Gamma \vdash P : U[T] \quad \Gamma \vdash Q : U[T]}{\Gamma \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : U[T]}$$

Fig. 6. Typing rules

This operation is sound because tail outputs of parallel values of the same type are not observable at the type system level. Rule (Tpipe) uses the function defined as

$$\begin{array}{lll} \mathsf{pipe}(U[\mathsf{end}], U'[T'], \tilde{S}) = U[\mathsf{end}] & & \text{(no pipe activation)} \\ \mathsf{pipe}(U[!(\tilde{S})], U'[T], \tilde{S}) = U \circ U'[T] & & \text{(pipe activated once)} \\ \mathsf{pipe}(U[!(\tilde{S})^k], U'[\mathsf{end}], \tilde{S}) = U \circ U'^k[\mathsf{end}] & k > 1 & \text{(multiple pipe activation)} \end{array}$$

where U'^k is a sequence of $k > 1$ output usages of the same type. If no value is passed to the pipe, then it is inessential. If it can be activated once, then its instance will act as a continuation for the current session. If multiple activations are considered, which will run in parallel, then each instance usage in the session must be end. Intuitively pipe constrains $P > \tilde{x} > Q$ in the current session to allow a single output P whenever the type of Q is different from a single input or vice versa Q to be a single input whenever the type of P is $!(\tilde{S})^k$ *a sequence of k-times* $!(\tilde{S})$. In this case the result is visible upward, repeating U' for k times. The first case of pipe is necessary to guarantee the subject reduction. (Tses) and (TsesI) are similar to service definition and invocation rules but r is removed from the environment to forbid the nesting of the same session name.

With respect to the two type systems presented in [19] ours is more similar to the one with balanced typing. In case the bound name m is a service, rule (TNEW) checks the existence of a corresponding service definition by means of $\mathsf{exists}(m, P)$. Function exists ensures that the process P declares the announced service (its inductive definition is as expected). Finally, rule (TIF) handles conditionals in the usual way.

Typing rules allow a deduction for processes like $r :!(int) \vdash r^+ \triangleright \langle 1\rangle | r^+ \triangleright \langle 2\rangle$ which do not preserve session linearity. We will exclude such processes by inroducing the notions of balanced and initial processes (see Definitions 1 and 4, and Theorem 1).

Example 1. Let us take the reservation example. The type $!(int)[!(str, str).?(int)]$ expresses the following client usage: the output of two strings is followed by the reading of the result and an integer is returned outside the session (the first output out of the square brackets indicates a return action, that is an output out of the current session). Previous usage is compared with the session usage $?(str, str).!(int)$ to ensure that the invocation is sound. Below we report the typing proof for the client, where we let $\Gamma = reserve : [\&\{\mathsf{double} :?(str, str).!(int), \mathsf{single} :?(str).!(int)\}]$, $P = \langle\text{"}\mathtt{Bob}\text{"}\rangle.(x).\mathtt{return}\ x$, $Q = \langle\text{"}\mathtt{Bob}\text{"}, \text{"}\mathtt{Leo}\text{"}\rangle.(x).\mathtt{return}\ x$ and $T = \oplus\{\mathsf{double} :!(str, str).?(int), \mathsf{single} :!(str).?(int)\}$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash P :!(int)[!(str).?(int)]}\,(\textsc{Tout})}{\Gamma \vdash \langle\mathsf{single}\rangle.P :!(int)[T]}\,(\textsc{Tchoice}) \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash Q :!(int)[!(str, str).?(int)]}\,(\textsc{Tout})}{\Gamma \vdash \langle\mathsf{double}\rangle.Q :!(int)[T]}\,(\textsc{Tchoice})}{\Gamma \vdash \mathtt{if}\ (test)\ \mathtt{then}\ \langle\mathsf{single}\rangle.P\ \mathtt{else}\ \langle\mathsf{double}\rangle.Q :!(int)[T]}\,(\textsc{Tif})$$

Moreover, we could safely replace the service definition with

$$reserve.\Big((\mathsf{double}).\mathtt{(x,y)}.\langle\mathtt{code(x,y)}\rangle\ +\ (\mathsf{single}).\mathtt{(x)}.\langle\mathtt{code(x,"")}\rangle\ +\ (\mathsf{suite}).R\Big)$$

(which extends the previous version of the service with additional behaviours) and still we correctly type check the client. In fact, our type system can statically exclude the new branch suite when the client is typed.

Example 2. Let us consider the factorial service $fatt$, defined by:

$$\begin{array}{l} fatt.\mathtt{(n).\ if\ (n=0)\ then}\ \langle 1\rangle \\ \qquad\qquad\qquad\quad \mathtt{else}\ \overline{fatt}.\langle\mathtt{n-1}\rangle.\mathtt{(x).return\ x} > \mathtt{x} > \langle\mathtt{mul(x,n)}\rangle \end{array}$$

Notice that in this case we are able to express the factorial thanks to service persistence, which guarantees a separation between each invocation. The entire program is well-typed by type checking only a single session instance. As the Theorem 2 will show, this check suffices to ensure that $fatt$ is deadlock free. The typing proof is below, where we recall that $\mathsf{end}[!(int)] = [!(int)]$ and let $P' = \langle n-1\rangle.(x).\mathtt{return}\ x$, $P = \overline{fatt}.P'$, $Q' = \langle\mathtt{mul}(x, n)\rangle$, $Q = P > x > Q'$ and $\Gamma = fatt : [?(int).!(int)], n : int, \mathtt{mul} : int \times int \to int$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\emptyset \vdash \langle 1\rangle : [!(int)]} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash P' :!(int)[!(int).?(int)]}}{\Gamma \vdash P : [!(int)]} \qquad \dfrac{\vdots}{\Gamma, x : int \vdash Q' : [!(int)]}}{\Gamma \vdash P > x > Q' : [!(int)]}}{\Gamma \vdash \mathtt{if}\ (n = 0)\ \mathtt{then}\ \langle 1\rangle\ \mathtt{else}\ Q : \mathsf{end}[!(int)]}}{fatt : [?(int).!(int)], \mathtt{mul} : \ldots \vdash (n).\mathtt{if}\ (n = 0)\ \mathtt{then}\ \langle 1\rangle\ \mathtt{else}\ Q : \mathsf{end}[?(int).!(int)]}}{fatt : [?(int).!(int)], \mathtt{mul} : \ldots \vdash fatt.(n).\mathtt{if}\ (n = 0)\ \mathtt{then}\ \langle 1\rangle\ \mathtt{else}\ Q : \mathsf{end}[\mathsf{end}]}$$

Example 3. Beyond basic types, expressions may take the name of a service as a parameter. Take a load balancing service that is called to discover, at each invocation, which service between a and b is more reliable for executing P.

$$(\nu\ a\ b)\ \Big(\mathit{loadbalance}.\mathtt{if}\ \mathtt{choose}(a,b)=1\ \mathtt{then}\ \langle a\rangle\ \mathtt{else}\ \langle b\rangle\ \mid\ a.P\ \mid\ b.P\Big)$$
$$\mid\ \overline{\mathit{loadbalance}}.(x).\mathtt{return}\ x > x > \overline{x}.Q$$

Here the function choose is a basic expression of type $[T]\times[T]\rightarrow \mathit{int}$ and uses the names of the two services as parameters. The client after receiving the name of the reliable service can substitute it for x for all future invocations of the service. It is a nice exercise to verify that the ensemble of the above processes is well-typed under the assumption that P and Q have types $[T]$ and $[\overline{T}]$ respectively.

The type system enjoys subject congruence and subject reduction. In their proofs we need some auxiliary lemmas that are proved by straightforward induction on the derivation of typing judgements.

Lemma 1 (Weakening). *If $\Gamma \vdash P : U[T]$ and $m \notin \mathsf{fn}(P)$ then $\Gamma, m : S \vdash P : U[T]$.*

Lemma 2 (Strengthening). *If $\Gamma, m : S \vdash P : U[T]$ and $m \notin \mathsf{fn}(P)$ then $\Gamma \vdash P : U[T]$.*

Proposition 1 (Subject Congruence). *If $\Gamma \vdash P : U[T]$ and $P \equiv Q$ then $\Gamma \vdash Q : U[T]$*

The following substitution lemma is needed in the proof of subject reduction for dealing with rules (In) and (Pipe).

Lemma 3 (Substitution). *Let $\Gamma, x : S \vdash P : U[T]$. If $\Gamma \vdash v : S$ then $\Gamma \vdash P[^v/_x] : U[T]$.*

Definition 1 (Balanced Process). *A process P is* balanced *if $\Gamma \vdash P : U[T]$ for some Γ, U, T and for each session name r in P, each of r^+ and r^- appears exactly once in P.*

Theorem 1 (Subject Reduction). *Let P be a balanced process and $\sqsubset$ be the smallest relation such that: $T \sqsubset ?(\tilde{S}).T$, $T \sqsubset !(\tilde{S}).T$, $T_i \sqsubset \&\{l_i : T_i\}_{i\in I}$ and $T_i \sqsubset \oplus\{l_i : T_i\}_{i\in I}$.*

1. *If $P \xrightarrow{\lambda} Q$ with $\lambda \in \{\tau, r\tau\}$, then Q is balanced (session linearity).*
2. *If $\Gamma, r : [T] \vdash P : U[T'']$ and $P \xrightarrow{r\tau} Q$ then $\Gamma, r : [T'] \vdash Q : U[T'']$ where $T' \sqsubset T$.*
3. *If $\Gamma \vdash P : U[T]$ and $P \xrightarrow{\tau} Q$ then $\Gamma \vdash Q : U[T]$.*

The proof is by induction on the derivation of the transition. Lemmas 1 and 2 serve to insert in /remove from Γ assumptions about session names. Session linearity guarantees that only "safe" programs are produced starting from balanced processes, i.e., that situation like $r^+ \triangleright \langle 1\rangle | r^+ \triangleright \langle 1\rangle$ (where r^+ appears twice) cannot arise at run-time.

4 Deadlock Freedom and Normal Form

Hereafter, we let $\omega, \gamma \ldots$ range over possibly empty sequences of labels τ, $r\tau$ and $r\iota$, where $r\iota$ will be introduced later by the rule (SCom'). We let $\xrightarrow{\omega}{}^{*}$ represents the reflexive and transitive closure of $\xrightarrow{\tau} \cup \xrightarrow{r\tau} \cup \xrightarrow{r\iota}$ (for all session names r). We say that a process is deadlock free if it cannot be blocked waiting a synchronisation unless it reaches the *normal form*. Normal form means that all the possible communications are exhausted and only service definitions remain.

Definition 2 (Normal form). *A process P is in normal form if there exist service names $s_1, ..., s_n$ and processes $Q_1, ..., Q_n$ such that $P \equiv (\nu s_1) \ldots (\nu s_n)(s_1.Q_1| \ldots |s_n.Q_n)$.*

Definition 3 (Deadlock free). *A process P is deadlock free if for each Q s.t. $P \xrightarrow{\omega}{}^{*} Q$ then either Q reaches a normal form or $Q \xrightarrow{\tau}$.*

Since deadlock freedom is a strong property we need to focus on a specific set of processes, called initial processes.

Definition 4 (Initial process). *A process P is initial if it does not contain session constructs, all service definitions are at the top level and $\emptyset \vdash P : \mathtt{end}[\mathtt{end}]$.*

Note that initial processes are also balanced. Our main theorem shows that all initial processes are deadlock free (see Theorem 2). Proving deadlock freedom involves the possibility of exhibiting a τ reduction after an arbitrary number of evaluation steps. However, to characterise the next admissible reduction in a constructive way, we need to argue about some specific session. To observe the name of the session in which a synchronisation is taking place we need a mild modification to the rule (SCom) of the transition system. The basic idea is to remove the binders for session names, i.e., to consider P' instead of P, for $P \equiv (\nu\tilde{r})P'$ where P' has no binder on session names. The revised rule (SCom') does not restrict the fresh session with a binder and uses the already mentioned label $r\iota$ to identify the created session.

$$\text{(SCom')}\quad \frac{P \xrightarrow{(r)s \Rightarrow r} P' \quad Q \xrightarrow{(r)s \Leftarrow r} Q'}{(P|Q) \xrightarrow{r\iota} (P'|Q')}$$

Another subtle aspect is that now session r is bound in the label $r\iota$, because it must be fresh w.r.t. all the other pre-existing session names: we omit parentheses in favour of a lighter syntax. On the other hand, the type environment Γ for closed processes can now contain assumptions about session names. The modification of the rule is sound since we are considering only processes that are reachable from initial processes. In fact, any initial process will produce for each new session a corresponding binder to restrict the session. For this reason the LTS with rule (SCom') has essentially the same behaviour of the previous LTS with rule (SCom) (just read both labels $r\iota$ and $r\tau$ as τ).

Lemma 4. *If P is initial and $P \xrightarrow{\omega}{}^{*} Q$, then there exists a typing environment Γ for session names in ω such that $\Gamma \vdash Q : \mathtt{end}[\mathtt{end}]$.*

To reason inductively on the way sessions are nested we introduce some convenient notation for contexts $\mathbb{C}[\![\cdot]\!]$ and $\mathbb{C}_{r^p}[\![\cdot]\!]$, parent/children sessions relation $\prec_P$ and session ancestors relation $<_P$. The set of contexts is defined by the grammar:

$$\mathbb{C} \ ::= \ [\![\cdot]\!] \mid \mathbb{C}|P \mid r^p \triangleright \mathbb{C} \mid (\nu s)\mathbb{C} \mid \mathbb{C} > \tilde{x} > P \qquad \mathbb{C}_{r^p} \ ::= \ r^p \triangleright ([\![\cdot]\!]|P)$$

As usual $\mathbb{C}[\![P]\!]$ and $\mathbb{C}_{r^p}[\![P]\!]$ are the processes obtained by filling the holes with P.

Definition 5 (Session nesting relation). *Let $r_1 \prec_P r_2$ iff $P \equiv \mathbb{C}[\![\mathbb{C}_{r_1^p}[\![r_2^q \triangleright Q]\!]]\!]$ for some contexts $\mathbb{C}, \mathbb{C}_{r_1^p}$, session r_2^q, and process Q. We let $<_P$ be the transitive closure of $\prec_P$.*

The relation $<_P$ for initial processes is also acyclic ($<_P$ is irreflexive), it is preserved by τ reductions and it holds that if $P \xrightarrow{r\iota} Q$ then $<_Q = <_P \cup \{(x, r) \mid \exists r_1 . r_1 \prec_P r \wedge x <_P r_1\}$.

The next proposition is a sort of progress property valid for the outermost (in terms of $\prec$-relation) active sessions. In fact, if one of such sessions has a pending action enabled then it is either guaranteed that after a finite number of steps a suitable synchronisation is accomplished or a service invocation can open a new nested session.

Proposition 2. *Let P be an initial process. If $P \xrightarrow{\omega}{}^* Q$, then for any session name r in Q if $\nexists \mathbb{C}, \mathbb{C}_{r^p}, v'$ and R such that $Q \equiv \mathbb{C}[\![\mathbb{C}_{r^p}[\![\mathtt{return}\ \tilde{\mathtt{v}}'.\mathtt{R}]\!]]\!]$ all of the following hold:*

1. *if $Q \xrightarrow{r^p : \leftarrow \tilde{v}}$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^{\overline{p}} : \rightarrow \tilde{v}} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$*
2. *if $Q \xrightarrow{r^p : \rightarrow \tilde{v}}$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^{\overline{p}} : \leftarrow \tilde{v}} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$*
3. *if $Q \xrightarrow{r^p : \leftarrow l}$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^{\overline{p}} : \rightarrow l} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$*
4. *if $Q \xrightarrow{r^p : \rightarrow l}$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^{\overline{p}} : \leftarrow l} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$*
5. *if $Q \equiv \mathbb{C}[\![\mathbb{C}_{r^p}[\![P' > \tilde{x} > Q']\!]]\!]$ and $\Gamma_1 \vdash P' : U[!(\tilde{S}).T]$ then*
   $\mathbb{C}[\![\mathbb{C}_{r^p}[\![P']\!]]\!] \xrightarrow{\gamma}{}^* \xrightarrow{r^p : \rightarrow \tilde{v}} \vee \mathbb{C}[\![\mathbb{C}_{r^p}[\![P']\!]]\!] \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$

Proof. Take any pair (r^p, Q) satisfying the premise. The proof is by induction on the well-founded order over pairs (r^p, Q) defined as the least transitive relation such that $(r_1^{p_1}, Q_1) < (r_2^{p_2}, Q_2)$ if one of the following holds:

- $llns(r_1, Q_1) < llns(r_2, Q_2)$,
- or $llns(r_1, Q_1) = llns(r_2, Q_2)$, $Q_1 \equiv \mathbb{C}[\![\mathbb{C}_{r^p}[\![Q_1'[^{\tilde{v}}/_{\tilde{x}}]]\!]]\!]$ and $Q_2 \equiv \mathbb{C}[\![\mathbb{C}_{r^p}[\![Q_2']\!]]\!]$ and Q_1' is a sub-term of Q_2' with $\mathsf{fn}(Q_2') = \tilde{x}$ (the substitution $[^{\tilde{v}}/_{\tilde{x}}]$ is possibly empty),

where we let $llns(r, Q)$ denote the length of the longest nesting sequence induced by $\prec_Q$ and starting with r, that is of the form $r \prec_Q r_1 \prec_Q r_2 \ldots r_{n-1} \prec_Q r_n$.

We sketch the proof for case (1). By Lemma 4, we know that there is a suitable Γ such that $\Gamma \vdash Q : \mathsf{end}[\mathsf{end}]$. Depending on p, we need to prove one of cases below:

1.a if $Q \xrightarrow{r^- : \leftarrow \tilde{v}}$ and $\Gamma = \Gamma_1, r : [!(\tilde{S}).T]$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^+ : \rightarrow \tilde{v}} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$
1.b if $Q \xrightarrow{r^+ : \leftarrow \tilde{v}}$ and $\Gamma = \Gamma_1, r : [?(\tilde{S}).T]$ then $Q \xrightarrow{\gamma}{}^* \xrightarrow{r^- : \rightarrow \tilde{v}} \vee Q \xrightarrow{\gamma}{}^* \xrightarrow{r_1 \iota} Q_1$ and $r <_{Q_1} r_1$

We can read the above statements as "a session side must respect, after a finite number of steps γ, the obligation imposed by its type unless it postpones the obligation with a new service call". The fact that the type of r reflects the enabled action is a direct consequence of the subject reduction.

Case 1.a: If $r : [!(\tilde{S}).T]$ it means that $Q \equiv \mathbb{C}[\![r^+ \triangleright Q']\!]$ with $\Gamma' \vdash Q' : U[!(\tilde{S}).T]$ for suitable Q, Γ' and U. The entire proof is completely type-driven, the key idea is that we consider only instances of rules able to yield $Q' : U[!(\tilde{S}).T]$ in the conclusion. To ease readability, in the rules we use W to range over processes and Δ over environments.

$$\text{(TOUT)}\ \frac{\Delta, \tilde{v} : \tilde{S} \vdash W : U[T]}{\Gamma' \vdash \langle \tilde{v} \rangle.W : U[!(\tilde{S}).T]} \qquad \text{(TINV)}\ \frac{\Delta, s : [\overline{T'}] \vdash W : !(\tilde{S}).T[T']}{\Delta, s : [\overline{T'}] \vdash \overline{s}.W : [!(\tilde{S}).T]}$$

(a) Some rule instances considered in base cases: output and invoke

$$\text{(TPARL)}\ \frac{\Gamma' \vdash W : U_W[!(\tilde{S}).T] \quad \Gamma' \vdash R : U_R[\mathtt{end}]}{\Gamma' \vdash W|R : U_W \circ U_R[!(\tilde{S}).T]} \qquad \text{(TPIPE)}\ \frac{\Gamma' \vdash W : U_W[!(\tilde{S}')] \quad \Gamma', \tilde{x} : \tilde{S}' \vdash R : U_R[!(\tilde{S}).T]}{\Gamma' \vdash W > \tilde{x} > R : U_W \circ U_R[!(\tilde{S}).T]}$$

(b) Some rule instances considered in inductive cases: parallel and pipe

$$\text{(TSESI)}\ \frac{\Delta \vdash W : !(\tilde{S}).T[T']}{\Delta, r_1 : [\overline{T'}] \vdash r_1^- \rhd W : \mathtt{end}[!(\tilde{S}).T]} \qquad \text{(TSESI + TIN)}\ \frac{\Delta \vdash W : !(\tilde{S}).T[T'']}{\Delta, r_1 : [\overline{T'}] \vdash r_1^- \rhd (\tilde{x}).W : \mathtt{end}[!(\tilde{S}).T]}$$

(c) Some rule instances considered in inductive cases: nested session

Fig. 7. Deduction rules

Base cases: The base cases are when $llns(r, Q)$ has length 0, i.e., there is no nested session in r^+ and r^-. Since $\Gamma' \vdash Q' : U[!(\tilde{S}).T]$ we consider the instances of rules compatible with an r^+ output action. Some of them are in Fig. 7(a). If (TOUT) is used, then it means that $Q' \equiv \langle \tilde{v} \rangle.W$. Then $Q \xrightarrow{r^- : \to \tilde{v}}$ and we are done by taking γ empty. Similarly, if (TINV) is used, then $Q' \equiv \overline{s}.W$. Then $Q \xrightarrow{r_1 \iota} Q_1$ with $r \prec_{Q_1} r_1$ (by invoking s) and we are done by taking γ empty.
Inductive cases: When (TPARL) is used the thesis follows by inductive hypothesis on $\mathbb{C}[\![r^+ \rhd W]\!]$ (see Fig. 7(b)) and similarly when (TPARR) is used. For (TPIPE) we apply the inductive hypothesis on $\mathbb{C}[\![r^+ \rhd W]\!]$. By case (5), either $\mathbb{C}[\![r^+ \rhd W]\!] \xrightarrow{\gamma'}{}^* \xrightarrow{r^+ \to \tilde{v}} \mathbb{C}[\![r^+ \rhd W']\!]$ and then $\mathbb{C}[\![r^+ \rhd (W > \tilde{x} > R)]\!] \xrightarrow{\gamma'}{}^* \xrightarrow{\tau} \mathbb{C}[\![r^+ \rhd ((W' > \tilde{x} > R)|R[^{\tilde{v}}/_{\tilde{x}}])]\!]$, and then the thesis follows by inductive hypothesis on $\mathbb{C}[\![r^+ \rhd R[^{\tilde{v}}/_{\tilde{x}}]]\!]$, or instead $\mathbb{C}[\![r^+ \rhd W]\!] \xrightarrow{\gamma}{}^* \xrightarrow{r_2 \iota}$ and therefore $\mathbb{C}[\![r^+ \rhd (W > \tilde{x} > R)]\!] \xrightarrow{\gamma}{}^* \xrightarrow{r_2 \iota} Q_1$ and $r \prec_{Q_1} r_2$. If (TSESI) is used, then a nested session r_1 is present, with $r \prec_Q r_1$ (see Fig. 7(c)), and we have various cases all similar. For example, if $Q' \equiv r_1^- \rhd (\tilde{x}).W'$ then $Q \xrightarrow{r_1^- : \leftarrow \tilde{v}'}$ and by inductive hypothesis either $Q \xrightarrow{\gamma'}{}^* \xrightarrow{r_1^+ : \to \tilde{v}'}$ and then the thesis follows adding $r_1 \tau$ at the begin of the resulting sequence generated by another application of the inductive hypothesis on $\mathbb{C}[\![r^+ \rhd (r_1^- \rhd W)]\!]$, or $Q \xrightarrow{\gamma'}{}^* \xrightarrow{r_2 \iota} Q_1$ with $r_1 \prec_{Q_1} r_2$ and then the thesis follows since $r \prec_Q r_1$ and $r_1 \prec_{Q_1} r_2$.
Case 1.b is similar (we might also have $Q \equiv \mathbb{C}[\![r^- \rhd \mathtt{return}\ \tilde{v}.W]\!]$, but this case can be discarded because it contradicts the assumptions). □

We are now ready to prove the main result.

Theorem 2 (Deadlock freedom). *Let P an initial process. Then for each Q s.t. $P \xrightarrow{\omega}{}^* Q$ then either $Q \xrightarrow{\tau}$ or Q is in normal form.*

Proof. If Q is in normal form we are done. If not, by contradiction, if Proposition 2 holds then it cannot be the case Q blocked on a pending action in the middle of a session. In fact, it is always possible to accomplish the synchronisation choosing the right session that fits the proposition hypothesis since Q has type $\mathtt{end}[\mathtt{end}]$. Such a session cannot have a `return` enabled because it would be seen as an output of the parent session. Since the process is closed the type system ensures that every service call is successful (rule (TNEW)). □

Remark 1. The result can be extended to processes P that can output some values. In fact, if $\Gamma \vdash P : \mathtt{end}[!(\tilde{S})^k]$ then we can take any suitable Q (designed to work on the resulting values) such that $\Gamma \vdash P > \tilde{x} > Q : \mathtt{end}[\mathtt{end}]$, thus fitting the requirements of Proposition 2. (The simplest case is $Q \equiv \mathbf{0}$.)

Example 4. The process $P = s.(x).\overline{s}.\langle x\rangle.(y).\mathtt{return}\ y \mid \overline{s}.\langle 5\rangle.(y)$ is well-typed in the environment $s : ?(int).!(int)$ and hence is deadlock free. Notice that the input of y never succeeds but the process is deadlock free since it keeps invoking new instances of s (all nested within the first established top session).

Even if simple, our framework also correctly type-checks non-tail recursive processes. For example, the initial process $s.\overline{s}.\mathtt{return}\ 1.(x)|\overline{s}.(y)$ is well-formed and well-typed in $\Gamma = s : !(int)$ and thus it is deadlock free (in our sense). An equivalent π-calculus process is $*s(r).(\nu r')(\overline{s}(r').\overline{r}(1).r'(x))|(\nu r)\overline{s}(r).r(y)$ and in Kobayashi's type system [14] the action $\overline{r}(1)$ cannot of course be ensured to succeed (check the tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`). In fact, r and r' have the same type and hence the capability level of r equal to the obligation level of r' but rule **A'**[14] is not applicable since r is created less recently than r'.

5 Conclusion

We have studied a service language with sessions and pipelines. Differently from [11, 12, 19, 9, 8, 7] our language build sessions automatically on each service invocation and disallows session name mobility (but not service name mobility). To some extent, the simple type system we have devised is similar to that of simply typed π-calculus because it only tracks the exchanged values in each session. In fact, since sessions are developed as low level run-time primitives, the type system does not need to check session linearity. Instead we track active session usages w.r.t. the current session and the parent session. Subsequently, we restrict on the class of initial processes for which well-typedness implies a suitable notion of deadlock freedom. Together with the type interference algorithm, reported in [18], we have a simple tool to check deadlock freedom. Among the main novelties we emphasise the typing rules for pipelines and the particular well-founded order used for the induction in the proof of Proposition 2.

The full version of this work will address the enhancement of the type system with a notion of sub-typing so that different usages of the same service can be typed consistently. Moreover, recursion and regular μ-types [10] will be accounted for in the type system, even if Proposition 2 will no longer hold in the present form. To see this, think of a process making the same unbounded number of inputs and returns: if we type the process as $\mu\alpha.!(\tilde{S}).\alpha[\mu\alpha.?(\tilde{S}).\alpha]$ then different numbers of inputs and returns are allowed. As future work, we want to relax some requirements on parallel usages, admit session passing and extend the result to multiparty sessions.

Acknowledgements: We would like to thank Mariangiola Dezani-Ciancaglini, Marija Kolundzija and the anonymous referees for their helpful and detailed comments.

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Festschrift in Honour of Ugo Montanari, on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*. Springer, 2008. To appear.
2. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: A service centered calculus. In *WS-FM'06*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.
3. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *FMOODS'08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.
4. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Sock: a calculus for service oriented computing. In *ICSOC'06*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
6. W. Cook, D. Kitchin, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR'06*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006.
7. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous session types and progress for object oriented languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.
8. M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On progress for structured communications. In *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.
9. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
10. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.
11. S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inform.*, 42(2):191–225, 2005.
12. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
13. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.
14. N. Kobayashi. New type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.
15. N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. In *CAV'08*, LNCS. Springer, 2008. To appear.
16. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
17. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
18. L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 216–231. Springer, 2008.
19. N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Elect. Notes in Th. Comput. Sci.*, 171(4):73–93, 2007.