A deadlock free type system for a calculus of services and sessions*

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 different instances of the same services, and to group together basic units of work. Recently, SCC has been proposed as a calculus centered around the concept of a dyadic session, where service interaction protocols and service orchestration can be conveniently expressed. In this paper we propose a type system to guarantee deadlock freedom within sessions, i.e., a key conformance requirement between service invocations and service definitions. The type system is based on previous researches focussed on 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 oriented architectures is attracting the interest of both industry and academy. On the one hand, important standardization 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 centered around typical usage patterns: it focuses on technical details of the implementation, but leaves several ambiguities on the overall methodology. On the other hand, several efforts are now posed on mathematical foundations, by developing formal languages and models tailored to services. The related documentation is aimed in first place to provide current standards with unambiguous semantics, but also to lay the basis for sound service orchestration, tackling the scenarios from a maybe more abstract perspective. Within this research thread, building on the experience maturated in the area of concurrency, many proposals of service oriented calculi have emerged [1, 4–9, 13, 20, 21] (just to cite a few), where well-studied process calculi have been enhanced with service-domain primitives. Interestingly, the pragmatic and the foundational perspective have been able to exert some mutual influence with benefits for both. For example, several formal encodings of WS-BPEL are now available and the design of WS-CDL has been strongly influenced by pi-calculus.

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

In this paper we focus on behavioral types for one such service oriented calculus, aiming to understand the syntactic restrictions needed to grant a fundamental property when combining services, i.e. that sessions are deadlock free.

We study a language derived from SCC [4] in which service invocation encompasses *one-way* and *request-response* [10–12, 3] but also allows for more sophisticated message exchanges, according to some fixed interaction protocol exposed by the service. The key feature is considering such messages 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 [15, 16, 22, 6] and also from languages based on correlation sets [21], here the programmers should not bother with the manipulation of sessions, as SCC chooses to instantiate sessions in an automatic way, upon service invocation. Automatic instantiation requires a mechanism to permit passing values outside a session, i.e., having a way for producing some values for the parent session as an outcome of a nested service invocation (a session can create its own sub sessions, resulting in an arbitrary session nesting). SCC allows mobility of service names [2], but not of session names. With respect to pi-calculus, in SCC the channel name expressing the communication media for exchanging messages is always implicit and determined by the context surrounding abstractions and concretions: communication is associated with the enclosing session/ pipeline.

For the typing, we consider the fragment of SCC where service definitions are always at the top level and persistent (i.e., replicated), an assumption which is fair enough w.r.t. current technologies. Moreover we think that enforcing service persistence to the language is not only a matter of taste. Without persistence sure the client can still check its specific interaction protocol against the single service specification, for example when the programmer statically designs its own service oriented application, but with service persistence the language can offer stronger global guaranties on the entire system, as our typing will demonstrate.

Even with service persistence, the resulting language is somehow too permissive to be dealt with session types [15, 22, 16] directly. Session types permit to correctly typecheck safety for a session if a process has some constraints, and one important constraint is that each input in a session must be matched by only an output. This condition is violated if, for example, in the presence of an input we introduce parallel outputs. Our type system extends ordinary session types to work correctly with our language. Differently from [17], this permits for each variable to have a simple way for either statically knowing its type in the case of the basic values or the service type in the case of service name mobility. Also we investigate the possibility of assigning obligations and capability levels (in the Kobayashi style [19, 18]) to our types, so that all session opened by well-typed processes are guaranteed to be deadlock free. The result is that exponential complexity of reachability analysis in Petri Nets is replaced here by a syntactic comparison of session usages. A similar type system exists for SSCC (a session language with streams) [20] that guarantees session safety. Streams are an alternative method to the return construct: they imply a sort of global buffer and permit only a single type of value for each stream.

Outline of the paper. Section 2 fixes the syntax and semantics of our SCC-like calculus. Section 3 presents the type system and the subject reduction result. Section 4 presents a new type system with obligations and capability level annotations able to guarantees that all session interactions are deadlock-free. Moreover it is shown that the new type systems essentially coincides with the simpler one in Section 3, in the sense that it is always possible to assign level annotations to the types of well-typed processes. Section 5 summarizes the results and points out directions for further work. For referees' convenience, omitted proofs are in Appendix.

2 Session Centered Calculus

Before the formal presentation of our language we illustrates a simple example.

2.1 Example

Consider the calculator service *calc*

$$calc.\Big(\ \texttt{(}sum\texttt{).(x,y).}\langle\texttt{x+y}\rangle\ +\ \texttt{(}inc\texttt{).(x).}\langle\texttt{x+1}\rangle\ \Big)$$

calc offers two operations, *sum* or *inc*, depending on the client choice. If the client after an invocation $\overline{calc}$ to *calc* sends the label *sum* to the service then the service waits for a tuple of two integers x and y, and after receiving them, sends the sum back to the client. Here the operator `+` is a function only available on service side either directly implemented or built with other services (the reader might think of `+` to be replaced by an operation that requires either an access to local data or a computational power available only at the service side). Functions available to the service together with the base type are left unspecified, because we want to model the entire system with modularity, in the case of *calc* service, for example we want to hide the sum implementation. Assume the client

$$\overline{calc}\texttt{.if (}test\texttt{) then }\langle inc\rangle\texttt{.}\langle\texttt{1}\rangle\texttt{.(x).return x}$$
$$\texttt{else }\langle sum\rangle\texttt{.}\langle\texttt{1,2}\rangle\texttt{.(y).return y}$$

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

$$(\nu\ r)\ \Big(\ r\triangleright\langle\texttt{1,2}\rangle\texttt{.(y).return y}\ \mid\ r\triangleright\texttt{(x,y).}\langle\texttt{x+y}\rangle\ \Big)$$

where the client protocol is running on the left and the service protocol on the right. Abstractions and concretions running on the opposite sides of the same session can exchange data, thus leading to

$$(\nu\ r)\ \Big(\ r\triangleright\texttt{(y).return y}\ \mid\ r\triangleright\ \langle\texttt{1+2}\rangle\ \Big).$$

After another interaction the client side ($r \triangleright \texttt{return}\ 3$) can return the result outside of r (to the parent session, if any). We shall see in Section 3 that thanks to session types, the type system collects the usage in each session and compares it with the correspondent dual usage to guarantee sound session interactions.

$$
\begin{array}{lll}
P,Q ::= & \mathbf{0} & \text{(nil)} \\
\mid & s.P & \text{(service definition)} \\
\mid & \overline{e}.P & \text{(invocation)} \\
\mid & \texttt{if } x = v \texttt{ then } P \texttt{ else } Q & \text{(if-then-else)} \\
\mid & (\tilde{x}).P & \text{(tuple input)} \\
\mid & \langle \tilde{v} \rangle.P & \text{(value output)} \\
\mid & \Sigma_{i=1}^{n}(l_i).P_i & \text{(label guarded sum)} \\
\mid & \langle l \rangle.P & \text{(label choice)} \\
\mid & \texttt{return } \tilde{v}.P & \text{(return)} \\
\mid & P|Q & \text{(parallel)} \\
\mid & P > Q & \text{(pipe)} \\
\mid & (\nu m)P & \text{(site/session restriction)} \\
\mid & r \triangleright P & \text{(session)}
\end{array}
$$

Fig. 1. Syntax of our service calculus.

2.2 Syntax

The set of processes is inductively defined by the grammar in Figure 1. We let P, Q range over the processes, s over service names, r over session names, m over both session and site names, l over labels, x over variables (over session names, data and labels), e over both service names and service variables, and v over values, which include an elsewhere specified set of basic data values and expressions (possibly with variables) together with service names.

As usual $[\tilde{\cdot}]$ identifies a tuple, $\mathbf{0}$ is the nil process, the trailing of $\mathbf{0}$ is often omitted, parallel composition is denoted by $P|Q$ and restriction $(\nu m)P$. Here restriction can be applied to services and sessions, but we assume only fresh service specification are available to the programmer, while programmers should not exploit the primitive $r \triangleright P$. A process without subterms $r \triangleright P$ is called *session-free*. Differently from other languages that provide primitives for explicit session naming and creation, here all sessions should be built only automatically, resulting in a more elegant and disciplined style of writing processes. In fact, two 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 $s.Q$, that share the same fresh name r. Then, input and output values communication in P and Q are uniquely directed toward the dual session labeled with the same name r. Also labels can be exchanged as values, but they are just a way 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 $P > Q$ is a construct for in-session communications i.e. for propagating values in the same side of a session. Pipe is inspired by ORC [13] to activate a fresh instance of Q on any value produced by P.

Processes are taken up to alpha-equivalence, i.e.

$$
\begin{array}{ll}
(\tilde{x}).P = (\tilde{y}).P[{}^{\tilde{y}}/_{\tilde{x}}] & \text{if } \tilde{y} \notin \mathsf{fn}(P) \\
(\nu x)P = (\nu y)P[{}^{y}/_{x}] & \text{if } y \notin \mathsf{fn}(P)
\end{array}
$$

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$
$$(\nu n)\mathbf{0} \equiv \mathbf{0} \qquad (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \qquad ((\nu n)P)|Q \equiv (\nu n)(P|Q) \text{ if } n \notin \mathsf{fn}(Q)$$
$$(\nu n)P > Q \equiv (\nu n)(P > Q) \text{ if } n \notin \mathsf{fn}(P) \qquad r \triangleright (\nu n)P \equiv (\nu n)(r \triangleright P) \text{ if } r \neq n$$
$$\mathbf{0} > R \equiv \mathbf{0} \qquad (P|Q) > R \equiv (P > R)|(Q > R)$$
$$(r \triangleright \mathbf{0}) > R \equiv r \triangleright \mathbf{0} \qquad r_1 \triangleright (r_2 \triangleright \mathbf{0})|Q \equiv r_1 \triangleright Q|r_2 \triangleright \mathbf{0} \qquad (\nu r)(r \triangleright \mathbf{0}|r \triangleright \mathbf{0}) \equiv \mathbf{0}$$

Fig. 2. Structural equivalence.

where the notion of free names $\mathsf{fn}(P)$ and bound names $\mathsf{bn}(P)$ are defined as usual, the operation of substitution $P[^{\tilde{v}}/_{\tilde{x}}]$, which replaces a tuple of variables with a tuple of values, considers that variables cannot appear in certain positions (i.e. that $x \triangleright P$ and $x.P$ are forbidden by the syntax). The effect of the substitution over the other processes is the standard capture avoiding substitution (when considering (νx) and $(\tilde{x}).P$ as binders).

Each service definition is persistent (i.e., not consumed after an invocation) and must appear at the top level. For this reason their protocols are not supposed to return any value to the parent in the type system given in Section 3.

In conclusion a process P is *well-formed* if: it is generated from the previous grammar, for each service name restriction there must be a relative service definition, all service definitions are at the top level and for each session name r appearing in P, there are exactly two (disjoint) session ends $r \triangleright P_1$ and $r \triangleright P_2$ that occur under the binding (νr), and that are active (in the sense of not occurring underneath a service definition, or service invocation, or input/ output/ return/ branch/ choice prefix, or the right-hand side of a pipeline). In our language restriction has a double role: it is used to model the visibility of a service and it is a sort of manifest for the service (like a WSDL) with respect to the considered analysis (in our case for example restriction annunciates the type of the session associated to the respective service invocation).

2.3 Operational Semantics

We describe the semantics of our language by means of an LTS that exploits a structural equivalence. The structural equivalence $\equiv$ is the least relation defined by the equations in Figure 2 (together with alpha equivalence). The structural congruence includes ordinary axioms about parallel and restriction, together with distributivity of parallel over pipes, and a few axioms for garbage collecting terminated session ends $r \triangleright \mathbf{0}$.

Our transition system exploits the labels λ in Figure 3. The session name r is bound in $s \Rightarrow (r)$ and $s \Leftarrow (r)$, and the tuple $\tilde{v}$ can contain free and bound names. We write $(s)\lambda$ to mean the label λ where the name s becomes bound. We write $\leftrightarrow$ to mean either $\leftarrow$ or $\rightarrow$.

The semantics is 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 com-

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

Fig. 3. Labels of the transition system.

municate 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. (Close) permits both communication of basic values and service names with the help of rule (Open). (Pipe) creates a new concurrent copy of process Q' (process Q without the initial input) together with the residual $P' > Q$ in the case that P outputs a value and (PipePass) makes a move in P if the action is not an output. The proofs of the lemmas below are by induction on the length of the computation.

Lemma 1. *Let P a well-formed process. Each process Q reachable from P via a sequence of τ transitions is well-formed.*

Lemma 2. *Let P a session-free process. For each session name r and each process Q reachable from P, if $r \triangleright R$ appears in Q then $r \notin \mathsf{fn}(R)$.*

3 Typing

In this section we define a first type system to check session safety. The set of types is defined by the grammar in Figure 5, building on a given set of data types D. (We shall assume that $int \in D$.) Types express sequences of typed tuples of input and output and square brackets pin session border. We do not need nested session types in our type system, even if the grammar permits that, because for this specific language we only need to check locally in-session communications to guarantee global properties. Also we do not consider those types in which basic types appear in a session without usage, e.g. $[int]$ instead of $[\mathtt{O}_{int}]$. Internal choice $\oplus$ records all the possible process choices in a certain point of a session, generated by means of an `if` instruction, like in the calculator example above.

Our set of type judgments is in Figure 6. Type judgments take the form $\Gamma \vdash P : S[T]$ where the type S represents the outputs of P to the parent session, while T is the type of current session usage. The type of $\mathbf{0}$ in (Tzero) is $[\mathtt{end}]$ since no action are performed neither in the current nor towards the parent session. The typing rule (Tnew) infers the right type of a service inserting it in the assumptions. Rule (Tdef) constrains the protocol of the service to be the same as the body process P and in the final type completely ignores all the returns

$$\text{(In)}\quad (\tilde{x}).P \xrightarrow{\leftarrow \tilde{v}} P[^{\tilde{v}}/_{\tilde{x}}] \qquad \text{(Out)}\quad \langle \tilde{v} \rangle.P \xrightarrow{\rightarrow \tilde{v}} P$$

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

$$\text{(Branch)}\quad \Sigma_{i=0}^{n}(l_i).P_i \xrightarrow{\rightarrow l_i} P_i \qquad \text{(Choice)}\quad \langle l \rangle.P \xrightarrow{\leftarrow l} P$$

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

$$\text{(Def)}\quad \frac{r \notin \mathsf{fn}(s.P)}{s.P \xrightarrow{s \Rightarrow (r)} r \triangleright P | s.P} \qquad \text{(Inv)}\quad \frac{r \notin \mathsf{fn}(\overline{s}.P)}{\overline{s}.P \xrightarrow{s \Leftarrow (r)} r \triangleright P}$$

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

$$\text{(Ret)}\quad \texttt{return}\ \tilde{v}.P \xrightarrow{\uparrow \tilde{v}} P \qquad \text{(SessionOut)}\quad \frac{P \xrightarrow{\uparrow \tilde{v}} P'}{r \triangleright P \xrightarrow{\rightarrow \tilde{v}} r \triangleright P'}$$

$$\text{(NestInv)}\quad \frac{P \xrightarrow{s \Leftarrow (r')} P' \quad r \neq r'}{r \triangleright P \xrightarrow{s \Leftarrow (r')} r \triangleright P'}$$

$$\text{(Close)}\quad \frac{P \xrightarrow{r:\leftarrow \tilde{v}} P' \quad Q \xrightarrow{r:\rightarrow \tilde{v}} Q' \quad \tilde{s} = \mathsf{bn}(\tilde{v}) \quad \tilde{s} \cap \mathsf{fn}(P) = \emptyset}{P|Q \xrightarrow{r\tau} (\nu \tilde{s})(P'|Q')}$$

$$\text{(Res)}\quad \frac{P \xrightarrow{\lambda} P' \quad s \notin \mathsf{n}(\lambda)}{(\nu s)P \xrightarrow{\lambda} (\nu s)P'} \qquad \text{(SessRes)}\quad \frac{P \xrightarrow{r\tau} P'}{(\nu r)P \xrightarrow{\tau} (\nu r)P'}$$

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

$$\text{(Pipe)}\quad \frac{P \xrightarrow{\rightarrow \tilde{v}} P' \quad Q \xrightarrow{\leftarrow \tilde{v}} Q' \quad \tilde{s} = \mathsf{bn}(\tilde{v}) \quad \tilde{s} \cap \mathsf{fn}(Q) = \emptyset}{P > Q \xrightarrow{\tau} Q'|(P' > Q)}$$

$$\text{(PipePass)}\quad \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \rightarrow \tilde{v} \quad \mathsf{bn}(\lambda) \cap \mathsf{fn}(Q) = \emptyset}{P > Q \xrightarrow{\lambda} P' > Q}$$

$$\text{(IfL)}\quad \frac{P \xrightarrow{\lambda} P'}{\texttt{if}\ v = v\ \texttt{then}\ P\ \texttt{else}\ Q \xrightarrow{\lambda} P'} \qquad \text{(IfR)}\quad \frac{w \neq v \quad Q \xrightarrow{\lambda} Q'}{\texttt{if}\ v = w\ \texttt{then}\ P\ \texttt{else}\ Q \xrightarrow{\lambda} Q'}$$

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

Fig. 4. Operational semantics.

$$
\begin{array}{rll}
T, S ::= & \mathtt{end} & \text{(no action)} \\
| & \mathtt{I}_{T_1,\ldots,T_n}.T & \text{(input of a tuple)} \\
| & \mathtt{O}_{T_1,\ldots,T_n}.T & \text{(output of a tuple)} \\
| & T[S] & \text{(session with returns)} \\
| & [S] & \text{(session without returns)} \\
| & \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(external choice)} \\
| & \oplus\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(internal choice)} \\
| & D & \text{(basic expression types)}
\end{array}
$$

Fig. 5. Syntax of types.

in S since the services must be defined at the top level by well-formedness. In fact returning a value in a service side is only useful to modify the internal state of the service, a feature which is not considered in the current analysis. (Tinv) checks that service behaves in the dual manner with respect to the client. Here the dual of T, written $\overline{T}$ is inductively defined as:

$$
\begin{array}{l}
\overline{\mathtt{I}_{\tilde{T}}.T'} = \mathtt{O}_{\tilde{T}}.\overline{T'} \qquad \overline{\mathtt{O}_{\tilde{T}}.T'} = \mathtt{I}_{\tilde{T}}.\overline{T'} \qquad \overline{\mathtt{end}} = \mathtt{end} \\
\overline{\&\{l_1 : T_1, \ldots, l_n : T_n\}} = \oplus\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\} \\
\overline{\oplus\{l_1 : T_1, \ldots, l_n : T_n\}} = \&\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\}
\end{array}
$$

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 specified but can be inferred thanks to session type. Rule (Tbranch) considers in the final type only the effective branches used by all the dual sessions in the rule (Tselect). 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. 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. it has a type [end]. Note instead that both P and Q are allowed to produce values upwards, in which case the operation $\circ$ translates parallel outputs of the same type in a sequence and is formally defined as $\mathtt{O}_{\tilde{T}} \circ \mathtt{O}_{\tilde{T}} = \mathtt{O}_{\tilde{T}}.\mathtt{O}_{\tilde{T}}$ and inductively extended to $S \circ S'$. This operation is always possible because tail outputs of parallel values of the same type at type system level are not observable. (Tpipe) uses the function $\mathsf{pipe}(S[T], S'[T']) = S''[T'']$ defined as

$$
\begin{array}{lll}
\mathsf{pipe}(S[\mathtt{end}], S'[T']) & = S[\mathtt{end}] & \\
\mathsf{pipe}(S[\mathtt{O}_{S'''}.T], S'[T']) & = S'' \circ S'[T^r] \ \text{ if } & T' = \mathtt{I}_{S'''}.T'' \quad \wedge \\
& & \mathsf{pipe}(S[T], S'[T']) = S''[T'''] \ \wedge \\
& & T'' = \mathtt{end} \vee T''' = \mathtt{end} \quad \wedge \\
& & T^r = \begin{cases} T''' & \text{if } \ T'' = \mathtt{end} \\ T'' & \text{if } \ T''' = \mathtt{end} \end{cases}
\end{array}
$$

Intuitively pipe constrains $P > Q$ in the current session to allow a single output P whenever the type of Q is different a single input or vice versa Q to be a single input whenever P is different from a single output. Consequently if

$$\text{(Env)}\ \Gamma, m : T \vdash m : T \qquad \text{(Tzero)}\ \Gamma \vdash \mathbf{0} : [\mathtt{end}] \qquad \text{(Tnew)}\ \frac{\Gamma, s : T' \vdash P : T}{\Gamma \vdash (\nu s)P : T}$$

$$\text{(Tdef)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash s : [T]}{\Gamma \vdash s.P : [\mathtt{end}]} \qquad \text{(Tinv)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash e : [T'] \quad \overline{T} = T'}{\Gamma \vdash \overline{e}.P : [S]}$$

$$\text{(Tif)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash Q : S[T]}{\Gamma \vdash \mathtt{if}\ x = v\ \mathtt{then}\ P\ \mathtt{else}\ Q : S[T]}$$

$$\text{(Tin)}\ \frac{\Gamma, \tilde{x} : \tilde{T}' \vdash P : S[T]}{\Gamma \vdash (\tilde{x}).P : S[\mathtt{I}_{\tilde{T}'}.T]} \qquad \text{(Tout)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash \tilde{v} : \tilde{T}'}{\Gamma \vdash \langle \tilde{v} \rangle.P : S[\mathtt{O}_{\tilde{T}'}.T]} \qquad \text{(Tret)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash \tilde{v} : \tilde{T}'}{\Gamma \vdash \mathtt{return}\ \tilde{v}.P : \mathtt{O}_{\tilde{T}'}.S[T]}$$

$$\text{(Tbranch)}\ \frac{I \subseteq \{1, \ldots, n\} \quad \forall i \in I \quad \Gamma \vdash P_i : S_i[T_i]}{\Gamma \vdash \Sigma_{i=0}^{n}(l_i).P_i : \&\{l_i : S_i\}[\&\{l_i : T_i\}]_{i \in I}}$$

$$\text{(TChoice)}\ \frac{l = l_i \in \{l_1, \ldots, l_n\} \quad \Gamma \vdash P : S_i[T_i]}{\Gamma \vdash \langle l \rangle.P : \oplus\{l_1 : S_1, \ldots, l_n : S_n\}[\oplus\{l_1 : T_1, \ldots, l_n : T_n\}]}$$

$$\text{(TparL)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash Q : S'[\mathtt{end}]}{\Gamma \vdash P|Q : S \circ S'[T]} \qquad \text{(TparR)}\ \frac{\Gamma \vdash P : S[\mathtt{end}] \quad \Gamma \vdash Q : S'[T]}{\Gamma \vdash P|Q : S \circ S'[T]}$$

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

$$\text{(Tses)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash r : [T]}{\Gamma \vdash r \triangleright P : [S]} \qquad \text{(TsesI)}\ \frac{\Gamma \vdash P : S[T] \quad \Gamma \vdash r : [T'] \quad \overline{T} = T'}{\Gamma \vdash \overline{r} \triangleright P : [S]}$$

Fig. 6. Typing rules.

$\Gamma \vdash Q : S[\mathtt{I}_{\tilde{T}'}]$ and $S = (\mathtt{O}_{\tilde{T}''})^k$ (a sequence of k outputs of type $\mathtt{O}_{\tilde{T}''}$) then it must be the case that $\Gamma \vdash P : (\mathtt{O}_{\tilde{T}''})^m[(\mathtt{O}_{\tilde{T}'})^n]$ for suitable n and m in order to type $P > Q$. (Tses) and (TsesI) are similar to service definition and invocation even if they are not used to write programs, they are needed for proving the subject reduction theorem. Notice that in the rule (TsesI) the name of the resource is overlined to distinguish inverse session side. This operation is not necessary in the language because we are always able to identify if we are in a session or in its dual by simply checking the assumptions in the typing environment.

Now we show some examples of typing,

Example 1. Let us take the calculator example. the type $\mathtt{O}_{int}[\mathtt{O}_{int,int}.\mathtt{I}_{int}]$ expresses the client usage: the output of two integers 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 $[\mathtt{I}_{int,int}.\mathtt{O}_{int}]$ to ensure that the invocation is sound. Below we report the typing proof, where we let $P = \langle inc \rangle.\langle 1 \rangle.(x).\mathtt{return}\ x$, $Q = \langle sum \rangle.\langle 1,2 \rangle.(x).\mathtt{return}\ x$, $T = \oplus\{sum : \mathtt{O}_{int,int}.\mathtt{I}_{int}, inc : \mathtt{O}_{int}.\mathtt{I}_{int}\}$ and $\Gamma = calc : \&\{sum : \mathtt{I}_{int,int}.\mathtt{O}_{int}, inc : \mathtt{I}_{int}.\mathtt{O}_{int}\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \langle 1 \rangle.(x).\mathtt{return}\ x : int[\mathtt{O}_{int}.\mathtt{I}_{int}]}{\Gamma \vdash \langle inc \rangle.\langle 1 \rangle.(x).\mathtt{return}\ x : int[T]} \quad \dfrac{\Gamma \vdash \langle 1,2 \rangle.(x).\mathtt{return}\ x : int[\mathtt{O}_{int,int}.\mathtt{I}_{int}]}{\Gamma \vdash \langle sum \rangle.\langle 1,2 \rangle.(x).\mathtt{return}\ x : int[T]}}{\Gamma \vdash \mathtt{if}\ (test)\ \mathtt{then}\ P\ \mathtt{else}\ Q : int[T]}$$

Moreover we could safely replace the service definition with

$$calc.\Big(\mathtt{(}sum\mathtt{).(x,y).}\langle \mathtt{x+y} \rangle\ +\ \mathtt{(}inc\mathtt{).(x).}\langle \mathtt{x+1} \rangle\ +\ \mathtt{(}sub\mathtt{).(x,y).}\langle \mathtt{x-y} \rangle\Big)$$

that extends the previous version of the service with additional behaviors. In fact, our type system can statically exclude the non used branch *inc* when the previous client is typed.

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

$$fatt\mathtt{.(n).\ if\ (n==0)\ then}\ \overline{\langle 1 \rangle}\ \mathtt{else}\ \overline{fatt}.\langle \mathtt{n-1} \rangle\mathtt{.(x).return\ x} > \mathtt{(x).}\langle \mathtt{x*n} \rangle$$

Notice that in this case we are able to express the factorial thanks to service persistence. This guarantees a separation between each invocation and the entire well typed program by type checking only a single session instance. Below we report the typing proof, where we let $P = \overline{fatt}.\langle n-1 \rangle.(x).\mathtt{return}\ x$, $Q' = (x).\langle x * n \rangle$, $Q = P > Q'$ and $\Gamma = fatt : \mathtt{I}_{int}.\mathtt{O}_{int}, n : int$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \langle n-1 \rangle.(x).\mathtt{return}\ x : \mathtt{O}_{int}[\mathtt{O}_{int}.\mathtt{I}_{int}]}{\Gamma \vdash \overline{fatt}.\langle n-1 \rangle.(x).\mathtt{return}\ x : \mathtt{O}_{int}} \quad \Gamma \vdash (x).\langle x * n \rangle : \mathtt{I}_{int}.\mathtt{O}_{int}}{\Gamma \vdash P > Q' : \mathtt{O}_{int}} \quad \emptyset \vdash \langle 1 \rangle : \mathtt{O}_{int}}{\Gamma \vdash \mathtt{if}\ (n == 0)\ \mathtt{then}\ \langle 1 \rangle\ \mathtt{else}\ Q : \mathtt{O}_{int}}}{fatt : \mathtt{I}_{int}.\mathtt{O}_{int} \vdash (n).\mathtt{if}\ (n == 0)\ \mathtt{then}\ \langle 1 \rangle\ \mathtt{else}\ Q : \mathtt{I}_{int}.\mathtt{O}_{int}}}{fatt : \mathtt{I}_{int}.\mathtt{O}_{int} \vdash fatt.(n).\mathtt{if}\ (n == 0)\ \mathtt{then}\ \langle 1 \rangle\ \mathtt{else}\ Q : [\mathtt{end}]}$$

Example 3. Beyond the basic type, expressions may use the name of a service as 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.

$$(\nu a)\ (\nu b)\ \Big(\ \mathit{loadbalance}.\langle \mathit{choose}\ (a,b)\rangle\ \ |\ a.P\ \ |\ \ b.P\ \Big)$$

Here the function *choose* is a basic expression 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 invocation of the service.

$$\overline{\mathit{loadbalance}}.\texttt{(x).return x > (x).}\overline{x}.Q$$

It is a nice exercise to verify that the ensemble of the above processes is well-typed (assuming P and Q are compatible).

The type system enjoys subject congruence and subject reduction. In their proof we need some auxiliary lemmas.

Lemma 3 (Weakening). *if* $\Gamma \vdash P : T$ *and* $n \notin \mathsf{fn}(P)$ *then* $\Gamma, n : S \vdash P : T$.

Proof. Straightforward induction on the derivation of the typing judgment, applying α-conversion when needed. □

Lemma 4 (Strengthening). *if* $\Gamma, n : S \vdash P : T$ *and* $n \notin \mathsf{fn}(P)$ *then* $\Gamma \vdash P : T$.

Proof. Straightforward induction on the derivation of the typing judgment. □

Lemma 5 (Subject Congruence). *If* $\Gamma \vdash P : T$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : T$

Lemma 6 (Value substitution). *Let* $\Gamma, y : S' \vdash P : T$. *If* $\Gamma \vdash w : S'$ *then* $\Gamma \vdash P[^{w}/_{y}] : T$.

In the proof of Subject reduction we shall use Lemma 2 together with the weakening lemma to remove an assumption about a session from Γ and strengthening lemma to reinsert a session with modified type. We also use the strengthening lemma to replace Γ' with Γ every time that $\Gamma' \subseteq \Gamma$ holds, and weakening lemma to replace Γ' with Γ every time that $\Gamma \subseteq \Gamma'$ holds.

Theorem 1 (Subject reduction). *If* $\Gamma \vdash R : T$ *and* $P \xrightarrow{\lambda} Q$ *then* $\exists \Gamma', T'$ *s.t.* $\Gamma' \vdash Q : T'$

4 Deadlock free sessions

In this section we show that every well typed program is deadlock free, in the sense that certain communications will eventually succeed (unless the process diverges). We show this by assigning suitable capability and obligation levels in the Kobayashi style [19, 18] to each well typed process. In pi-calculus-like languages, a session usage can depend both by its dual counterpart and by its level of nesting. We show that in our case we can abstract away from the

session nesting because if all inputs and outputs are balanced within a session, then each return always succeeds making the outputs available one level up. In the following lemma the concept of balanced means that each action has a correspondent coaction in the dual session usage. We will show that verifying such condition is the same as verifying syntactic equality among a session type and its dual usage.

Lemma 7 (Balancing). *If each session r in P is balanced then each return succeeds unless the entire process diverges.*

The previous lemma shows that we can consider each session as separated by the others, hence deadlock freedom (which include the possibility of a process to diverge) can be verified just with the help of session usages. In turn this implies that each session must be well typed in the Kobayashi type system style. Following this intuition the index for deadlock freedom can be assigned with the help of the following two rules:

A An obligation of level n must be fulfilled by using only capabilities of level less than n.
B For an action of capability level n, there must exist a co-action of obligation level less than or equal to n.

Originally, rule **A** was used to capture the sequentiality of communications among different channels in each subprocess, but here intuitively all the communications within a session are along the same channel and isolated as the previous lemma shows. Even if rule **A** is here useless for the original purpose, we enforce its usage, for future extensions of the language. Previous lemma in fact is optimistic and does not hold if for example we admit session communication.

Next we are going to show how to assign capability and obligation levels to each session to ensure that Kobayashi condition of *reliability*, written *Rel*, and that rule **A** always holds. The *Rel* condition checks that for each possible evolution of a session rule **B** holds. For example if $\mathcal{T}$ is $\mathtt{O}_1|\mathtt{O}_2 \parallel \mathtt{I}_1.\mathtt{I}_2$ depending of which output among $\mathtt{O}_1$ or $\mathtt{O}_2$ is performed first, it can evolve to either $\mathtt{O}_2 \parallel \mathtt{I}_2$ or $\mathtt{O}_1 \parallel \mathtt{I}_2$. The $\parallel$ separates two sides of a session and is different from the $|$ among types which expresses parallel in-session usage. The evaluation relation on types $\rightarrow$ defined in Figure 7, uses $\preceq$ which is not symmetric. Here $\mathcal{T}_1 \equiv \mathcal{T}_2$ indicates that both $\mathcal{T}_1 \preceq \mathcal{T}_2$ and $\mathcal{T}_2 \preceq \mathcal{T}_1$ hold, α indicates a generic action either input or output and $\star$ both either external or internal choices. Note that we define such relation without caring about the square brackets that may appear in the type because it is used only for checking *Rel* within a session. We shall also omit $\mathtt{I}/\mathtt{O}$ subscripts, as they can be recovered from the type system in Section 3.

Next we define two functions ob and cap defining the obligation and the capability level of a process, respectively.

$$\mathsf{ob}(\alpha^{t_o}_{t_c}.\mathcal{T}) = t_o \qquad \mathsf{ob}(\mathcal{T}_1|\mathcal{T}_2) = \mathbf{min}(\mathsf{ob}(\mathcal{T}_1), \mathsf{ob}(\mathcal{T}_2))$$
$$\mathsf{ob}(\star\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\}) = \mathbf{min}(\mathsf{ob}(\mathcal{T}_1), \ldots, \mathsf{ob}(\mathcal{T}_n))$$

$$\mathsf{cap}(\alpha^{t_o}_{t_c}.\mathcal{T}) = t_c \qquad \mathsf{cap}(\mathcal{T}_1|\mathcal{T}_2) = \mathbf{min}(\mathsf{cap}(\mathcal{T}_1), \mathsf{cap}(\mathcal{T}_2))$$
$$\mathsf{cap}(\star\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\}) = \mathbf{min}(\mathsf{cap}(\mathcal{T}_1), \ldots, \mathsf{cap}(\mathcal{T}_n))$$

$$\mathcal{T}_1|\mathcal{T}_2 \equiv \mathcal{T}_2|\mathcal{T}_1 \qquad (\mathcal{T}_1|\mathcal{T}_2)\mathcal{T}_3 \equiv \mathcal{T}_1|(\mathcal{T}_2|\mathcal{T}_3)$$

$$\star\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\} \parallel \overline{\star}\{l_1 : \mathcal{S}_1, \ldots, l_n : \mathcal{S}_n\} \preceq \mathcal{T}_i \parallel \mathcal{S}_i \qquad \frac{\mathcal{T}_1 \preceq \mathcal{T}_1' \quad \mathcal{T}_1' \preceq \mathcal{T}_2}{\mathcal{T}_1 \preceq \mathcal{T}_2}$$

$$\alpha_{t_c}^{t_o}.\mathcal{T}_1 \parallel \overline{\alpha}_{t_c'}^{t_o'}.\mathcal{T}_2 \to \mathcal{T}_1 \parallel \mathcal{T}_2 \qquad \frac{\mathcal{T}_1 \preceq \mathcal{T}_1' \quad \mathcal{T}_1' \to \mathcal{T}_2' \quad \mathcal{T}_2' \preceq \mathcal{T}_2}{\mathcal{T}_1 \to \mathcal{T}_2}$$

Fig. 7. Relation of evaluation on types.

These two simple functions are useful to describe the semantics of each type operator w.r.t. obligation/ capability levels. Differently from [19] here the obligation level of external choice is the minimum among branches because it is always uniquely compared with the correspondent internal choice from its counterpart.

Definition 1. *$Rel(\mathcal{T} \parallel \mathcal{T}')$ is coinductively defined to hold either if $\mathcal{T} = \mathcal{T}' =$ end or if $\mathsf{ob}(\mathcal{T}) \leq \mathsf{cap}(\mathcal{T}')$ and $\mathsf{ob}(\mathcal{T}') \leq \mathsf{cap}(\mathcal{T})$ and for each $\mathcal{T}''$ and $\mathcal{T}'''$ such that $\mathcal{T} \parallel \mathcal{T}' \to \mathcal{T}'' \parallel \mathcal{T}'''$ then $Rel(\mathcal{T}'' \parallel \mathcal{T}''')$ holds.*

Lemma 8. *If Rel holds for a session side of type $\mathcal{T}$ and its counterpart of type $\mathcal{T}'$ then that session is deadlock free*

Proof. Let $\alpha_{t_c}^{t_o}$ one of the action with minimum capability annotation performed in a session of type $\mathcal{T}$. Since $Rel(\mathcal{T} \parallel \mathcal{T}')$ holds then the dual session usage $\mathcal{T}'$ has $\mathsf{ob}(\mathcal{T}') \leq t_c$. Moreover in $\mathcal{T}'$ there must be a cousage with capability annotation, say t_c'. Since initially we choose the usage with the minimum annotation it must be $t_c' \leq t_o$. In a few word we have $\alpha_{t_c}^{\leq t_c'}$ and $\overline{\alpha}_{t_c'}^{\leq t_c}$ that ensures that there are at least two coactions that are waiting each others. Previous proof does not need rule **A** to hold, but if the system permits interactions between different sessions it ensures that α and its cousage $\overline{\alpha}$ are actions on the same session i.e. there are not action on other sessions blocking them. □

The type system with annotated levels $\Gamma \vdash_{oc} P : \mathcal{T}$ differs from the previous one only in the rules in Figure 8 and because the definition of pipe uses $|$ instead of $\circ$. The condition in the rules (Sin), (Sout) and (Sret) on the obligation level enforces rule **A**.

Example 4. $\overline{\mathtt{O}_T.\mathtt{O}_T.\mathtt{O}_T.\mathtt{O}_T} = \mathtt{I}_T.\mathtt{I}_T.\mathtt{I}_T\mathtt{I}_T$ and $Rel(\mathtt{O}_{t_{c1}}^{t_{o1}}.\mathtt{O}_{t_{c2}}^{t_{o2}}|\mathtt{O}_{t_{c3}}^{t_{o3}}|\mathtt{O}_{t_{c4}}^{t_{o4}} \parallel \mathtt{I}_{s_{c1}}^{s_{o1}}.\mathtt{I}_{s_{c2}}^{s_{o2}}.\mathtt{I}_{s_{c3}}^{s_{o3}}.\mathtt{I}_{s_{c4}}^{s_{o4}})$. By rule **A** (implicitly enforced by the typing rules):

$$s_{c1} < s_{o2} \quad s_{c2} < s_{o3} \quad s_{c3} < s_{o4} \quad t_{c1} < t_{o2}$$

and because of *Rel* the following inequalities are required:

$$\begin{array}{ll} \mathbf{min}(t_{o1}, t_{o3}, t_{o4}) \leq s_{c1} & s_{o1} \leq \mathbf{min}(t_{c1}, t_{c3}, t_{c4}) \\ \mathbf{min}(t_{o1}, t_{o2}, t_{o3}, t_{o4}) \leq s_{c2} & s_{o2} \leq \mathbf{min}(t_{c1}, t_{c2}, t_{c3}, t_{c4}) \\ \mathbf{min}(t_{o1}, t_{o2}, t_{o3}, t_{o4}) \leq s_{c3} & s_{o3} \leq \mathbf{min}(t_{c1}, t_{c2}, t_{c3}, t_{c4}) \\ \mathbf{min}(t_{o2}, t_{o3}, t_{o4}) \leq s_{c4} & s_{o4} \leq \mathbf{min}(t_{c2}, t_{c3}, t_{c4}) \end{array}$$

$$\text{(SInv)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad \Gamma \vdash_{oc} s : [\mathcal{T}'] \quad Rel(\mathcal{T} \parallel \mathcal{T}')}{\Gamma \vdash_{oc} \overline{s}.P : [\mathcal{S}]}$$

$$\text{(SparL)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad \Gamma \vdash_{oc} Q : \mathcal{S}'[\mathtt{end}]}{\Gamma \vdash_{oc} P|Q : \mathcal{S}|\mathcal{S}'[\mathcal{T}]} \qquad \text{(SparR)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathtt{end}] \quad \Gamma \vdash_{oc} Q : \mathcal{S}'[\mathcal{T}]}{\Gamma \vdash_{oc} P|Q : \mathcal{S}|\mathcal{S}'[\mathcal{T}]}$$

$$\text{(SsesI)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad \Gamma \vdash_{oc} r : [\mathcal{T}'] \quad Rel(\mathcal{T} \parallel \mathcal{T}')}{\Gamma \vdash_{oc} \overline{r} \triangleright P : [\mathcal{S}]}$$

$$\text{(Sin)}\ \frac{\Gamma, \tilde{x} : \tilde{T}' \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad t_c < \mathsf{ob}(\mathcal{T})}{\Gamma \vdash_{oc} (\tilde{x}).P : \mathcal{S}[\mathtt{I}^{t_o}_{t_c}.\mathcal{T}]} \qquad \text{(Sout)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad \Gamma \vdash_{oc} \tilde{v} : \tilde{\mathcal{T}}' \quad t_c < \mathsf{ob}(\mathcal{T})}{\Gamma \vdash_{oc} \langle \tilde{v} \rangle.P : \mathcal{S}[\mathtt{O}^{t_o}_{t_c}.\mathcal{T}]}$$

$$\text{(Sret)}\ \frac{\Gamma \vdash_{oc} P : \mathcal{S}[\mathcal{T}] \quad \Gamma \vdash_{oc} \tilde{v} : \tilde{\mathcal{T}}' \quad t_c < \mathsf{ob}(\mathcal{S})}{\Gamma \vdash_{oc} \mathtt{return}\ \tilde{v}.P : \mathtt{O}^{t_o}_{t_c}.\mathcal{S}[\mathcal{T}]}$$

Fig. 8. Typing rules with deadlock indexes annotation.

Then $Rel(\mathtt{O}^0_4.\mathtt{O}^5_6|\mathtt{O}^0_6|\mathtt{O}^0_6 \parallel \mathtt{I}^0_1.\mathtt{I}^2_3.\mathtt{I}^4_5.\mathtt{I}^6_6)$ holds.

This example also shows that the index assignments to the first output of each parallel subprocess must account for all the actions of the other parallel processes: the capability level of the first output is 4, the same as the obligation level of the third input, indicating the presence of other two parallel outputs.

Here we discuss an algorithmic way to assign index annotations such that if $T = \overline{T'}$ holds in a premise of a rule for $\vdash$ then $Rel(\mathcal{T} \parallel \mathcal{T}')$ holds in the premise of the corresponding rule for $\vdash_{oc}$. Therefore it is not necessary to define a translation function from one type system to other, because if some hypotheses hold we are able to type a session with indexes, and later we just discover that the necessary hypotheses are only those that we implicitly assume when syntactically compare a type with the dual type of the cousage. There are two main cases to consider:

1. $T = \alpha_1 \ldots \alpha_n.(\underbrace{\mathtt{O} \ldots \mathtt{O}}_{m_1} | \ldots | \underbrace{\mathtt{O} \ldots \mathtt{O}}_{m_k})$ and $T' = \overline{\alpha_1} \ldots \overline{\alpha_n}.\underbrace{\mathtt{I} \ldots \ldots \mathtt{I}}_{m}$ where $m = \sum_{i=m_1}^{m_k} m_i$. Then $Rel(T, T')$ holds when we assign indexes as shown below, with $t_c(i) = n + 1 + 2 * (m - i)$ (for readability reasons we assume that level enumeration starts from 0):

$$\begin{array}{ll} & \alpha_1{}^0_1 \ldots \alpha_n{}^{n-1}_n.(\underbrace{\mathtt{O}^{n+1}_{t_c(m_1)} \ldots \mathtt{O}^{n+2*(m-1)}_{t_c(1)}}_{m_1} | \ldots | \underbrace{\mathtt{O}^{n+1}_{t_c(m_k)} \ldots \mathtt{O}^{n+2*(m-1)}_{t_c(1)}}_{m_k}) \\ \parallel & \overline{\alpha_1}{}^0_1 \ldots \overline{\alpha_n}{}^{n-1}_n.\mathtt{I}_1{}^{n+1}_{n+2} \ldots \mathtt{I}_m{}^{n+m-1}_{n+m} \end{array}$$

Now since the parallel outputs have the same type this is the same to syntactically compare a type with the dual of its cousage.

2. The two types $T = \alpha_1{}_1^0 \ldots \alpha_n{}_n^{n-1}.\star\{l_1 : \mathcal{T}_1{}_{n+2}^{n+1}, \ldots, l_n : \mathcal{T}_n{}_{n+2}^{n+1}\}$ and $T' = \overline{\alpha_1}{}_1^0 \ldots \overline{\alpha_n}{}_n^{n-1}.\overline{\star}\{l_1 : \mathcal{S}_1{}_{n+2}^{n+1}, \ldots, l_n : \mathcal{S}_n{}_{n+2}^{n+1}\}$ involve different choices, where $\mathcal{T}_1{}_{t_c}^{t_o}$ indicates the inductive indexes assignment for type $\mathcal{T}$ starting from t_o and t_c. Here we have assumed that the two types at the end of a dual sequence perform a dual choice and in each branch of the choice all the hypotheses we have assumed must hold. Still, we have that the hypotheses hold if a type matches with the dual of its cousage.

This show that we can forget the indexes and parallel annotation and use the original type system instead.

Theorem 2. *If $\emptyset \vdash P : [T]$ then P is deadlock free*

Proof. The type system only checks that deadlock cannot happen in the middle of a session. The others two cases that we need to consider are the service call and the pipe. The service call cannot fail because by assumption we type only closed services which in turn implies a corresponded service definition for the well-formedness and each service definition is persistent. The case of pipe is accounted for by the definition of pipe. In $P > Q$ if P produces one output then P has a type $S[\mathtt{0}_T]$ which is consumed by the first input of Q. Both these input and output are removed from the final type. If instead P produces many outputs then the only input in Q consumes all of them. The final type considers a parallel execution of the returns of P with the returns of Q one for each output in P. □

5 Conclusion

We have studied a service language with sessions. Differently from [15, 16, 22, 6] our language builds sessions automatically on each service invocation. For this language we have defined a type system that has the property to typecheck deadlock free programs. For proving this we have proposed another type system with indexes annotation in the Kobayashi type system style in which some numeric inequalities must hold for ensuring deadlock freedom. At the end we give an algorithmic solution to such inequalities for each well typed process in the original type system.

Future work will address the enhancement of the type system with a notion of subtyping so that different usage of the same service can be typed consistently. Moreover we plan to extend the language with recursion processes and see if regular μ-types [14] can be accommodated for in the type system. Furthermore we want to relax some requirements on parallel usages, admit session passing and extend the type system to multiparty session languages like [6].

References

1. L. Acciai and M. Boreale. Xpi: a typed process calculus for xml messaging. In *Proceedings of FMOODS'05*, volume 3535 of *Lect. Notes in Comput. Sci.*, pages 47–66. Springer Verlag, 2005.

2. BEA Systems, IBM, Microsoft, SAP, and Sun. *Web Services Addressing (WS-Addressing)*, 2004. http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/.
3. D. Booth and C. Liu. *Web Services Description Language (*WSDL*) Version 2.0 Part 0: Primer*, 2006. http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327.
4. 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 M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proceedings of WS-FM'06*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer Verlag, 2006.
5. A. L. Brown, C. Laneve, and L. G. Meredith. Piduce: A process calculus with native xml datatypes. In *Proceedings of EPEW'05/WS-FM'05*, volume 3670 of *Lect. Notes in Comput. Sci.*, pages 18–34. Springer Verlag, 2005.
6. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in soc. In *Proceedings of WS-FM'07*, 2007. To appear.
7. M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2007.
8. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Sock: a calculus for service oriented computing. In A. Dan and W. Lamersdorf, editors, *Proceedings of ICSOC'06*, volume 4294 of *Lect. Notes in Comput. Sci.*, pages 327–338. Springer Verlag, 2006.
9. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web serices. In *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 2–17. Springer Verlag, 2007.
10. R. Chinnici, H. Haas, A. Lewis, J.-J. Moreau, et al. *Web Services Description Language (*WSDL*) Version 2.0 Part 2: Adjuncts*, 2006. http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327.
11. R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. *Web Services Description Language (*WSDL*) Version 2.0 Part 1: Core Language*, 2006. http://www.w3.org/TR/2006/CR-wsdl20-20060327.
12. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (*WSDL*) 1.1*, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.
13. W. Cook, D. Kitchin, and J. Misra. A language for task orchestration and its semantic properties. In C. Baier and H. Hermanns, editors, *Proceedings of CONCUR'06*, volume 4137 of *Lect. Notes in Comput. Sci.*, pages 477–491. Springer Verlag, 2006.
14. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.
15. S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inform.*, 42(2):191–225, 2005.
16. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Proceedings of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer Verlag, 1998.
17. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.
18. N. Kobayashi. Typical: Type-based static analyzer for the pi-calculus. Tool available at http://www.kb.ecei.tohoku.ac.jp/~koba/typical/.

19. N. Kobayashi. New type system for deadlock-free processes. In C. Baier and H. Hermanns, editors, *Proceedings of CONCUR'06*, volume 4137 of *Lect. Notes in Comput. Sci.*, pages 233–247. Springer Verlag, 2006.
20. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of SEFM'07*. IEEE Computer Society Press, 2007. To appear.
21. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer Verlag, 2007.
22. N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Appendix

Lemma (Subject Congruence). *If $\Gamma \vdash P : T$ and $P \equiv Q$ then $\Gamma \vdash Q : T$*

Proof. By induction on the derivation of $P \equiv Q$, with a case analysis on the last rule used. The inductive cases are the congruence rules, and are straightforward. Of the other cases we show the two most significant.
Case for $((\nu x)P)|Q \equiv (\nu x)(P|Q)$ if $x \notin \mathsf{fn}(Q)$, we have:

$$* \quad \dfrac{\dfrac{\Gamma, x : S''' \vdash P : S'[T]}{\Gamma \vdash (\nu x)P : S'[T]}\,(\text{Tnew}) \qquad \Gamma \vdash Q : S''[\mathtt{end}]}{\Gamma \vdash (((\nu x))P)|Q : S' \circ S''[T]}\,(\text{TparL})$$

which can be rearranged to give:

$$** \quad \dfrac{\Gamma, x : S''' \vdash P : S'[T] \qquad \Gamma, x : S''' \vdash Q : S''[\mathtt{end}]}{\dfrac{\Gamma, x : S''' \vdash (P|Q) : S' \circ S''[T]}{\Gamma \vdash (\nu x)(P|Q) : S' \circ S''[T]}\,(\text{Tnew})}\,(\text{TparL})$$

The two proofs * and ** can be read from * to ** to have the only if side in which we use the weakening lemma to type Q with assumption x and from ** to * for the if side in which we use the strengthening lemma to type Q without the assumption x. The case in which (TparR) is used is similar.
Case for $r_1 \triangleright (r_2 \triangleright \mathbf{0})|Q \equiv r_1 \triangleright Q|r_2 \triangleright \mathbf{0}$, we have:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathbf{0} : [\mathtt{end}] \qquad \Gamma \vdash r_2 : [\mathtt{end}]}{\Gamma \vdash r_2 \triangleright \mathbf{0} : [\mathtt{end}]}\,(\text{Tses}) \qquad \Gamma \vdash Q : S[T]}{\Gamma \vdash (r_2 \triangleright \mathbf{0})|Q : S[T]}\,(\text{Tpar1}) \qquad \Gamma \vdash r_1 : [T]}{\Gamma \vdash r_1 \triangleright (r_2 \triangleright \mathbf{0})|Q : [S]}\,(\text{Tses})$$

which can be rearranged to give:

$$\dfrac{\dfrac{\Gamma \vdash Q : S[T] \qquad \Gamma \vdash r_1 : [T]}{\Gamma \vdash r_1 \triangleright Q : [S]}\,(\text{Tses}) \qquad \dfrac{\Gamma \vdash \mathbf{0} : [\mathtt{end}] \qquad \Gamma \vdash r_2 : [\mathtt{end}]}{\Gamma \vdash r_2 \triangleright \mathbf{0} : [\mathtt{end}]}\,(\text{Tses})}{\Gamma \vdash r_1 \triangleright Q|r_2 \triangleright \mathbf{0} : [S]}\,(\text{Tpar})$$

Since the two proofs use the same sets of assumptions we can conclude the hypothesis.
Case for $(P|Q) > R \equiv (P > R)|(Q > R)$:

$$\dfrac{\dfrac{\Gamma \vdash P : S[\mathtt{O}_{\tilde{T_1}}] \qquad \Gamma \vdash Q : S'[\mathtt{end}]}{\Gamma \vdash (P|Q) : S \circ S'[\mathtt{O}_{\tilde{T_1}}]}\,(\text{TparR}) \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T_1}}.T]}{\Gamma \vdash (P|Q) > R : S \circ S' \circ S''[T]}\,(\text{Tpipe})$$

which can be arranged to give:

$$\dfrac{\dfrac{\Gamma \vdash P : S[\mathtt{0}_{\tilde{T}_1}] \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T}_1}.T]}{\Gamma \vdash (P > R) : S \circ S''[T]}\ (\text{Tpipe}) \qquad \dfrac{\Gamma \vdash Q : S'[\mathtt{end}] \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T}_1}.T]}{\Gamma \vdash Q > R : S'[\mathtt{end}]}\ (\text{Tpipe})}{\Gamma \vdash (P > R)|(Q > R) : S \circ S' \circ S''[T]}\ (\text{TparR})$$

The mirror case which uses (TParL) is similar. The are also two other cases in which the process R can make only one input.

$$\dfrac{\dfrac{\Gamma \vdash P : S[(\mathtt{0}_{\tilde{T}})^n] \qquad \Gamma \vdash Q : S'[\mathtt{end}]}{\Gamma \vdash (P|Q) : S \circ S'[(\mathtt{0}_{\tilde{T}})^n]}\ (\text{TparR}) \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T}}]}{\Gamma \vdash (P|Q) > R : S \circ S' \circ (S'')^n[\mathtt{end}]}\ (\text{Tpipe})$$

which can be arranged to give:

$$\dfrac{\dfrac{\Gamma \vdash P : S[(\mathtt{0}_{\tilde{T}})^n] \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T}}]}{\Gamma \vdash (P > R) : S \circ (S'')^n[\mathtt{end}]}\ (\text{Tpipe}) \qquad \dfrac{\Gamma \vdash Q : S'[\mathtt{end}] \qquad \Gamma \vdash R : S''[\mathtt{I}_{\tilde{T}}]}{\Gamma \vdash Q > R : S'[\mathtt{end}]}\ (\text{Tpipe})}{\Gamma \vdash (P > R)|(Q > R) : S \circ S' \circ (S'')^n[\mathtt{end}]}\ (\text{TparR})$$

□

Lemma (Value substitution). *Let $\Gamma, y : S' \vdash P : T$. If $\Gamma \vdash w : S'$ then $\Gamma \vdash P[^w/_y] : T$.*

Proof. By induction on the typing judgment.
Without losing the essence of the argument, we consider only 1-arity.
Case for (Tzero): straightforward.
Cases for (Tin) and (Tnew) : follow directly from the induction hypothesis and the fact that variable under the binder cannot be chosen neither as a object nor as a subject of the substitution.
Case for (Tout): the only interesting case is when $y = v$, we have:

$$\dfrac{\overset{(\text{Tout})}{\Gamma', v : S', w : S' \vdash P : S[T]} \quad \Gamma', v : S', w : S' \vdash v : S'}{\Gamma', v : S', w : S' \vdash \langle v \rangle.P : S[\mathtt{0}_{S'}.T]}$$

by induction hypothesis $\Gamma', w : S' \vdash P[^w/_v] : S[T]$ so (Tout) gives $\Gamma', w : S' \vdash \langle w \rangle.P[^w/_v] : S[\mathtt{0}_{S'}.T]$ which is the required judgment.
Case for (Tret): the only interesting case is when $y = v$, we have:

$$\dfrac{\overset{(\text{Tret})}{\Gamma', v : S', w : S' \vdash P : S[T]} \quad \Gamma', v : S', w : S' \vdash v : S'}{\Gamma', v : S', w : S' \vdash \mathtt{return}\ v.P : \mathtt{0}_{S'}.S[T]}$$

by induction hypothesis $\Gamma', w : S' \vdash P[^w/_v] : S[T]$ so (Tret) gives $\Gamma', w : S' \vdash \mathtt{return}\ w.P[^w/_v] : \mathtt{0}_{S'}.S[T]$ which is the required judgment.
Case for (Tinv) the only interesting case is when $y = x$, we have:

$$\dfrac{\overset{(\text{Tinv})}{\Gamma', x : [S''], w : [S''] \vdash P : S[T]} \quad \Gamma', x : [S''], w : [S''] \vdash x : [S''] \quad \overline{T} = S''}{\Gamma', x : [S''], w : [S''] \vdash \overline{x}.P : [S]}$$

by induction hypothesis $\Gamma', w : [S''] \vdash P[^{w}/_{x}] : S[T]$ and $\overline{T} = S''$ so (Tinv) gives $\Gamma', w : [S''] \vdash \overline{w}.P[^{w}/_{x}] : S[T]$ which is the required judgment.
Cases for (Tchoice), (Tbranch), (Tses), (Tses1) and (Tdef) follow directly from the induction hypothesis.
Cases for (Tpar), (Tpar1), (Tif) and (Tpipe) follow directly from the distributivity of substitution with respect to $|$, `if` and $>$ and the fact that `pipe` and $\circ$ work only on the resulting type. □

Theorem (Subject reduction). *If $\Gamma \vdash R : T$ and $P \xrightarrow{\lambda} Q$ then $\exists \Gamma', T'$ s.t. $\Gamma' \vdash Q : T'$*

Proof. By induction on the derivation with a case analysis on the last rule used.
Case for (In): We have $\Gamma \vdash (\tilde{x}).P' : S[\mathtt{I}_{\tilde{T'}}.T]$ and $(\tilde{x}).P' \xrightarrow{\leftarrow \tilde{v}} P'[^{\tilde{v}}/_{\tilde{x}}]$. We also know that $\Gamma, \tilde{x} : \tilde{T} \vdash P' : S[T]$, and by the application of the substitution lemma we have $\Gamma \vdash P'[^{\tilde{v}}/_{\tilde{x}}] : S[T]$ which is the desiderated judgment.
Case for (Branch) We have $\Gamma \vdash \Sigma_{i=1}^{n}(l_i).P_i : \&\{l_i : S_i\}[\&\{l_i : T_i\}]_{i \in I}$ and $I \subseteq \{1, \ldots, n\}$. The desiderated judgment follows because $\Gamma \vdash P_i : S_i[T_i]$.
Case for (Choice): We have $\Gamma \vdash \langle l \rangle.P : \oplus\{l_1 : S_1 \ldots l_n : S_n\}[\oplus\{l_1 : T_1 \ldots l_n : T_n\}]$ and $l = l_i$. The desiderated judgment follows because $\Gamma \vdash P : S_i[T_i]$.
Case for (Select): We have $\Gamma \vdash (P|Q) : [S]$. By inductive hypothesis we know $\Gamma \vdash P' : S$ and $\Gamma \vdash Q' : [\mathtt{end}]$. Applying (TparL) we obtain $\Gamma \vdash (P'|Q') : [S]$ which is the desiderated judgment.
Case for (Out): We have $\Gamma', \tilde{v} : \tilde{T'} \vdash \langle \tilde{v} \rangle.P' : S[\mathtt{O}_{\tilde{T'}}.T]$ and $\langle \tilde{v} \rangle.P' \xrightarrow{\rightarrow \tilde{v}} P'$. We also know that $\Gamma', \tilde{v} : \tilde{T'} \vdash P' : S[T]$ which is the desiderated judgment.
Case for (Ret): We have $\Gamma', \tilde{v} : \tilde{T'} \vdash \mathtt{return}\ \tilde{v}.P' : \mathtt{O}_{\tilde{T'}}.S[T]$ and $\mathtt{return}\ \tilde{v}.P' \xrightarrow{\uparrow \tilde{v}} P'$. We also know that $\Gamma', \tilde{v} : \tilde{T'} \vdash P' : S[T]$ which is the desiderated judgment.
Case for (Def): We have $\Gamma', s : [T] \vdash s.P' : [\mathtt{end}]$ and $s.P' \xrightarrow{s \Rightarrow r} r \triangleright P'|s.P'$. We also know that $\Gamma', s : [T] \vdash P' : S[T]$ and that the session r has type $[T]$ which is the same as that of s. Thanks to (TSes) we can conclude that $\Gamma', s : [T], r : [T] \vdash r \triangleright P' : [S]$ and finally with (TparL) $\Gamma', s : [T], r : [T] \vdash r \triangleright P'|s.P' : [S]$ which is the desiderated judgment.
Case for (Inv): We have $\Gamma', s : [T] \vdash s.P' : [S]$ and $\overline{s}.P' \xrightarrow{s \Leftarrow r} \overline{r} \triangleright P'$. We also know that $\Gamma', s : [T] \vdash P : S[T']$ with $\overline{T} = T'$ and that the session $\overline{r}$ has a type $[T']$ (because initially the session is not used at all). Thanks to (TSesI) we can conclude that $\Gamma', s : [T], r : [T] \vdash \overline{r} \triangleright P' : [S]$ which is the desiderated judgment.
Case for (SCom): We have $\Gamma \vdash (P|Q) : [S \circ S']$. By inductive hypothesis we also know $\Gamma, r : [T] \vdash P' : [S]$ and $\Gamma, r : [T] \vdash Q' : [S']$ and by applying either (TParL) or (TparR) and (Tnew) we obtain either $\Gamma \vdash (\nu r)(P'|Q') : [S \circ S']$ or $\Gamma \vdash (\nu r)(P'|Q') : [S' \circ S]$ which are the required judgments because $S' \circ S = S \circ S'$ for definition.
Case for (SessionOut): We have $\Gamma', r : [\mathtt{I}_{\tilde{T_1}}.T] \vdash r \triangleright P : [\mathtt{O}_{\tilde{T_1}}.T']$ with $\overline{\mathtt{O}_{\tilde{T_1}}.T'} = \mathtt{I}_{\tilde{T_1}}.T$. By inductive hypothesis we know that $\Gamma'', r : [T] \vdash P' : T'[T'']$ where $\Gamma'' \subseteq \Gamma'$. Thanks to (TsesI) we can conclude $\Gamma', r : [T] \vdash r \triangleright P' : [T']$ which is the desiderated judgment. In the other case we have, $\Gamma', r : [\mathtt{O}_{\tilde{T_1}}.T] \vdash r \triangleright P : [\mathtt{O}_{\tilde{T_1}}].T$ and it is similar.

Case for (Session): We have $\Gamma', r : [T] \vdash r \triangleright P : [T]$. By inductive hypothesis we know $\Gamma'', r : [T] \vdash P' : T[S]$ where either $\Gamma'' \subseteq \Gamma'$ (for output) or $\Gamma' \subseteq \Gamma''$ (for input) or $\Gamma' = \Gamma''$ (depending of the performed action) and thanks to (Tses) we can conclude $\Gamma', r : [T] \vdash r \triangleright P' : [T]$ which is the desiderated judgment.
Case for (NestInv). We have $\Gamma', r : [T], s : [T'] \vdash r \triangleright P : [T]$. By inductive hypothesis we know $\Gamma', r : [T], s : [T'], r' : [T'] \vdash P' : T[T']$. Thanks to (Tses) we can conclude $\Gamma', r : [T], s : [T'], r' : [T'] \vdash r \triangleright P' : [T]$.
Case for (Close): We have $\Gamma', r : [\mathtt{0}_{T'}.T] \vdash (P|Q) : [S]$ (resp. $r : [\mathtt{I}_{T'}.T)$. By inductive hypothesis we know $\Gamma'', r : [T] \vdash P' : S$ where $\Gamma'' \subseteq \Gamma$ (resp. $\Gamma \subseteq \Gamma''$) and $\Gamma''', r : [T] \vdash Q' : [\mathtt{end}]$ where $\Gamma \subseteq \Gamma'''$ (resp. $\Gamma''' \subseteq \Gamma$) with $\overline{T''} = T$. Applying (TparL) we obtain $\Gamma', r : [T] \vdash (P'|Q') : [S]$ which is the desiderated judgment. The other cases one where P is located on client side and one where Q performs returns are similar.
Case for (Res): We have $\Gamma \vdash P : S[T]$ and by inductive hypothesis $\Gamma, x : [T] \vdash P' : S[T]$. Applying (Tnew) we obtain $\Gamma \vdash P : S[T]$ which is the desiderated judgment.
Cases for (SessRes) and (Open) similar to the previous case.
Case for (Par): We have $\Gamma \vdash P|Q : S \circ S'[T]$ and by inductive hypothesis $\Gamma \vdash P' : S''[T'']$. Applying (TParL) we have $\Gamma \vdash P'|Q : S'' \circ S'[T']$. The other case with $P : S''[\mathtt{end}]$ is similar.
Case for (Pipe): We have $\Gamma \vdash P > Q : S''[T'']$ and by inductive hypothesis we also have $\Gamma' \vdash P' : S[\mathtt{end}]$ where $\Gamma' \subseteq \Gamma$ and $\Gamma \vdash Q' : S'[T'']$ with $S'' = S \circ S'$ and the desiderated judgment follows by applying (TParR). In the other case we have $\Gamma \vdash P > Q : S''[\mathtt{end}]$ by inductive hypothesis we have $\Gamma' \vdash P' : S[T]$ where $\Gamma' \subseteq \Gamma$ and $\Gamma'' \vdash Q' : S'[\mathtt{end}]$ where $\Gamma \subseteq \Gamma''$. By applying (TPipe) we have $\Gamma \vdash P' > Q : S'''[\mathtt{end}]$ and with (TParR) finally we can conclude $\Gamma \vdash (P' > Q)|Q' : S''[\mathtt{end}]$ where $S'' = S \circ S'''$
Case for (PipePass) we have $\Gamma \vdash P > Q : S''[T'']$ and by inductive hypothesis we also have $\Gamma' \vdash P' : S[T]$ where $\Gamma \subseteq \Gamma'$ and by applying (TPipe) $\Gamma \vdash P' > Q : S''[T'']$ where $S'' = S \circ S'$ for some type S'
Cases for (IfL) and (IfR): Follow directly by inductive hypothesis.
Case for (Equiv): Follow directly by inductive hypothesis and the subject congruence lemma. □

Lemma (Balancing). *If each session r in P is balanced then each return succeeds unless the entire process diverges.*

Proof. By induction on the level of session nesting. In the base case we have a session with no inner session that performs returns to the outermost session. In this case we know that each input and output in a session is consumed so each return is performed as soon as it is encountered. In the inductive case we have a session which depends on the returns of n-nested sub session. By inductive hypothesis we know that all the returns at level $n-1$ succeed and can be considered as outputs in the current session. Since inputs and outputs are balanced by hypothesis, even the current session performs all of its returns. The entire process may however diverges due some in-session service invocation □