How to infer session types in a calculus of services and sessions*

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

[1] Computer Science Department, University of Pisa, Italy
[2] IMT Lucca, Institute for Advanced Studies, Italy

Abstract. Regular session types account for the sequence of typed I/O actions and internal/external choices to be carried out by (possibly recursive) protocols between two communicating entities, to ensure they are compatible and will not block because of type mismatches in the next action to perform. They have been applied to many process calculi, ranging from π-calculus variants, to object-oriented calculi and, more recently, to service-oriented ones. This paper addresses the problem of session type inference, a problem not faced before, especially for what concern the reconstruction of recursive protocols. We show that the problem is decidable for a variant of the Service Centered Calculus (SCC) and give a sound and complete type inference algorithm. The choice of the SCC as a target language gives us the possibility to have a quite simplified presentation, thanks to the well-disciplined structuring of allowed communications. Still the result can be extended to other calculi with minor reworking apart from increasing technicalities.

1 Introduction

In this paper we face the problem of the reconstruction of (recursive) session types for dyadic sessions. A dyadic session is a private *locus of conversation* between two parties, which is used to exchange messages in both directions and to take decisions. For trustworthy interaction, it is then important to guarantee that the two parties carry compatible protocols. *Session types* [12] are the most accepted paradigm to type sessions, where the sequence of (the types of) input/output actions and internal/external choices are accounted for: two protocols are then compatible if they carry *dual* session types. Since communication is private within a session, the type system must also care about the linear usage of a session, i.e. the partner side must be uniquely determined.

Session type inference looks difficult because of two main problems: relaxation of duality matches and (regular) infinite behaviours. The first problem is due to the fact that session types should not mirror exactly each action in the process. For example, a client can select just one option from those made available by a service and ignore the others, while the type of the service must consider

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

that different clients can pick different options each time. In the presence of conditional constructs (like if-then-else considered here), the static analysis must take into account that more than one selection will be possible at run time (internal choice) because of the two possible branches, even if only one is to be taken. On the other hand, the set of available options (external choices) present in a type can be a subset of all offered choices; the subset which contains only the branches that can be effectively used by available partners. The second problem concerns the reconstruction of infinite behaviors by means of regular μ-types. The problem is that the type system keeps track of session actions while descending in the proof tree, while the typing rule for recursive process variables needs to know the complete type of recursion at the leaves of the proof tree.

In this paper we propose an algorithm for session type inference where both problems are solved. The target calculus we consider is a service-oriented calculus with dyadic nested sessions, and recursive protocols, which stems from the so called SCC-family [2, 14, 3, 1, 5], but it can otherwise be seen as a suitable extension of the π-calculus. We start by introducing a type system for session types in a fragment of SCC, where the goal-oriented application of the rule is not syntax-directed, in the sense that some degree of non-determinism is present in the rules. Such system is convenient to prove subject reduction and type safety. Then, we introduce a syntax directed type system and a corresponding algorithm (implemented in Ocaml and F#) which is used to extract constraints over type variables: the derived constraints are satisfiable if and only if the process is well typed in the non-syntax-directed type system.

Our algorithm solves the first problem by computing a suitable intersection of session types and the second problem by associating fresh type variables to leaves and by closing such variables only after that all the entire recursive behaviors are collected. This technique to infer recursive behaviors resembles the one used in [13] concerning the maximal solution of equation of subusage constraints, but here we infer the body of the recursion directly by means of syntax directed typing rules. While [13] focuses on the inference of each single channel usage, here we focus on the inference of actions within sessions. A session can be thought of as a single channel used bidirectionally to exchange different kinds of data values, which is very different from the standard channel sorting discipline.

The syntactic constraints of SCC allow us to design a quite simple type system, but the main approach remains general enough to be successfully applied to other session typing systems. For example, we could have presented the approach over the calculus proposed in [12], here however the study is simplified since each process in each moment has only two open sessions (the current session and the parent session) instead of having to handle many interleaved sessions along the same thread. The same consideration apply to session delegation, where session obligations can be delegated to other parties by means of sending and receiving session names: even if here disregarded for simplicity, it can be easily accommodated for. Moreover, SCC has specific features on its own, like nested sessions.

Our type system has been strongly influenced by [12, 11]. This paper is also an enhancement of [5, 16], where recursive types were not allowed.

$$
\begin{array}{llll}
P,Q ::= & & \mathbf{0} & \text{(nil)} \\
& \mid\ s.P & \mid\ \overline{v}.P & \text{(service definition / invocation)} \\
& & \mid\ r^p \triangleright P & \text{(session)} \\
& \mid\ (\tilde{x}).P & \mid\ \langle \tilde{v} \rangle.P & \text{(polyadic input / output)} \\
& & \mid\ \mathtt{return}\ \tilde{v}.P & \text{(value return)} \\
& \mid\ \Sigma_{i=1}^{n}(l_i).P_i & \mid\ \langle l \rangle.P & \text{(label branching / selection)} \\
& & \mid\ \mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q & \text{(if-then-else)} \\
& \mid\ (\nu m)P & \mid\ P|Q & \text{(restriction / parallel)} \\
& \mid\ X & \mid\ \mathtt{rec}\ X.P & \text{(process variable / recursion)} \\
v,w ::= x & & \mid\ s & \text{(variable / service)} \\
& & \mid\ \ldots,-1,0,1,\ldots & \text{(integers)}
\end{array}
$$

Fig. 1. Syntax of our service calculus

Structure of the paper. Section 2 introduces our target calculus. Section 3 defines the (non-syntax-directed) type system, showing subject congruence and subject reduction. Section 4 illustrates the syntax-directed type system and the corresponding inference algorithm, showing their soundness and completeness w.r.t. the type system in Section 3. Some concluding remarks are in Section 5.

2 A session oriented calculus

We assume an infinite collection $r,\ldots$ of session names, an infinite collection $s,\ldots$ of service names, an infinite collection of variables $x,y,\ldots$, an infinite collection of process variables $X,Y,\ldots$ and an infinite collection $l,\ldots$ of labels. Additionally we use m to range over both service and session names. We let $\tilde{\cdot}$ denote tuples and $|\cdot|$ their cardinality. The syntax of processes $P,Q,\ldots$ is defined in Fig. 1, where values $v,w,\ldots$ are either variables x, services s or integers.

As usual $\mathbf{0}$ is the nil process (whose trailing is often omitted), parallel composition is denoted by $P|Q$, restriction by $(\nu m)P$ and recursion by $\mathtt{rec}\ X.P$. The construct $r^p \triangleright P$ indicates a generic session side with polarity $p \in \{+,-\}$. We use p,q as meta variables to range over session polarities and $\overline{p}$ is the dual polarity of p, i.e.; $\overline{+} = -$ and $\overline{-} = +$. A fresh session name r and two polarized 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 $s.Q$. Then, polyadic i/o communications in P and Q are uniquely directed toward the dual side, labeled with the same name r but with dual polarity. Communication primitives inside each session are standard for session typed calculi: input/output (i.e. abstractions $(\tilde{x})$ and concretions $\langle \tilde{v} \rangle$ prefixes) and internal/external choices (i.e. a way for expressing a choice $\langle l \rangle.P$ on one side among a set of available options $\Sigma_i(l_i).Q_i$ at the other side). Sessions can be arbitrarily nested (e.g., in $\overline{s_1}.(P|\overline{s_2}.Q)$ the session established with s_2 will run inside the session established with s_1) and the $\mathtt{return}$ primitive outputs a value to (the partner of) the parent session.

$$
\begin{array}{lll}
\text{(Inv)} & \mathbb{D}[\![\overline{s}.P, s.Q]\!] \;\rightarrow\; (\nu r)\mathbb{D}[\![r^- \rhd P, r^+ \rhd Q]\!] & (r \notin \mathsf{fn}(\mathbb{D}[\![\overline{s}.P, s.Q]\!])) \\
\text{(Com)} & \mathbb{D}_r[\![(\tilde{x}).P, \langle\tilde{v}\rangle.Q]\!] \;\rightarrow\; \mathbb{D}_r[\![P[^{\tilde{v}}/_{\tilde{x}}], Q]\!] & (|\tilde{x}| = |\tilde{v}|) \\
\text{(Lcom)} & \mathbb{D}_r[\![\Sigma_{i=1}^{n}(l_i).P_i, \langle l_k\rangle.Q]\!] \;\rightarrow\; \mathbb{D}_r[\![P_k, Q]\!] & (1 \leq k \leq n) \\
\text{(Ret)} & \mathbb{D}_{r_1}[\![(\tilde{x}).P, \mathbb{C}_{r^q}[\![\mathtt{return}\ \tilde{v}.Q]\!]]\!] \;\rightarrow\; \mathbb{D}_{r_1}[\![P[^{\tilde{v}}/_{\tilde{x}}], \mathbb{C}_{r^q}[\![Q]\!]]\!] & \\
\text{(IfT)} & \mathbb{C}[\![\mathtt{if}\ v = v\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \rightarrow \mathbb{C}[\![P]\!] & \\
\text{(IfF)} & \mathbb{C}[\![\mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \rightarrow \mathbb{C}[\![Q]\!] & (v \neq w) \\
\text{(Rec)} & \mathbb{C}[\![P[^{\mathtt{rec}\ X.P}/_X]]\!] \rightarrow P' \Rightarrow \mathbb{C}[\![\mathtt{rec}\ X.P]\!] \rightarrow P' & \\
\text{(Scop)} & P \;\rightarrow\; P' \;\Rightarrow\; (\nu m)P \rightarrow (\nu m)P' & \\
\text{(Str)} & P \equiv P' \;\wedge\; P' \rightarrow Q' \;\wedge\; Q' \equiv Q \Rightarrow\; P \rightarrow Q &
\end{array}
$$

$$
\begin{array}{ll}
\mathbb{C} \;::=\; [\![\cdot]\!] \;\mid\; \mathbb{C}|P \;\mid\; r^p \rhd \mathbb{C} & \mathbb{C}_{r^p} \;::=\; r^p \rhd ([\![\cdot]\!]|P) \\
\mathbb{D} \;::= \mathbb{C}[\![\mathbb{C}' \mid \mathbb{C}'']\!] & \mathbb{D}_r \;\;::= \mathbb{D}[\![\mathbb{C}'_{r^p}, \mathbb{C}''_{r^{\overline{p}}}]\!] \quad r \notin \mathsf{fn}(\mathbb{D})
\end{array}
$$

Fig. 2. Semantics of our service calculus

Priority of the operators in order of increasing relevance is: $|$, $\rhd$ and ν so, for example $r \rhd P|Q$ means $(r \rhd P)|Q$ and $(\nu r)P|Q$ means $((\nu r)P)|Q$. The binders for the calculus are $(\tilde{x}).P$ for $\tilde{x}$ in P and $(\nu m)P$ for m in P, and $\mathtt{rec}\ X.P$ for X in P, with standard notions of free names (fn) and bound names (bn) of a process, bound (bpv) and free process variables (fpv) and closed processes (when $\mathsf{fpv}(P) = \emptyset$). Processes are considered equivalent up to renaming of bound names and bound processes variables. The structural congruence relation $\equiv$ is the standard π-calculus like, with the additional axiom $r^p \rhd (\nu m)P \equiv (\nu m)(r^p \rhd P)$ if $m \neq r$ for the floating of restrictions w.r.t session side construct.

We present the operational semantics of the calculus by means of reduction contexts. We have four different types of contexts generated from respective grammars. The one-hole context $\mathbb{C}[\![\cdot]\!]$ is useful to insert a process P, the result being denoted $\mathbb{C}[\![P]\!]$ (process P replaces the hole inside $\mathbb{C}$), into an arbitrary nesting of sessions together with arbitrary processes in parallel. The one-hole context $\mathbb{C}_{r^p}$ allows inserting a process into the session r^p running in parallel with an arbitrary process. Contexts $\mathbb{D}$ and $\mathbb{D}_r$ are the two-holes counterparts of the previous contexts. In each context, binders cannot appear above the hole.

Rule (Inv) defines how the invocation of a service creates a new session r that puts in direct communication an instance of the service protocol Q with the client body P. The rules (Com) and (Lcom) show respectively how a tuple is transmitted between the two sides of a session and how the dual partner can select one of the options offered by the other. As usual $P[^{\tilde{v}}/_{\tilde{x}}]$ is the simultaneous substitution of each variable $\tilde{x}$ with values $\tilde{v}$ in P if $|\tilde{x}| = |\tilde{v}|$. Rule (Ret) illustrates how a nested session r^q can output a value upward to the parent session r_1^p, which is read by $(\tilde{x}).P$ in $r_1^{\overline{p}}$. Remaining rules (IfT), (IfF), (Rec), (Scop) and (Str) are standard.

Example 1. As a running example we take the process

$$
\begin{array}{l}
(\nu a)(\overline{a}.\mathtt{rec}\ X.\langle l\rangle.\Sigma_{i=1}^{4}(l_i).X \;\big|\; a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 \\
\qquad\qquad\qquad\qquad\qquad\quad \big|\; a.\mathtt{rec}\ Y_1.(l).\mathtt{if}\ \mathit{test}\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1)
\end{array}
$$

$$
\begin{array}{rll}
T ::= & \texttt{end} & \text{(no action)}\\
| & ?(S_1,\ldots,S_n).T & \text{(input of a tuple)}\\
| & !(S_1,\ldots,S_n).T & \text{(output of a tuple)}\\
| & \&\{l_1:T_1,\ldots,l_n:T_n\} & \text{(external choice)}\\
| & \oplus\{l_1:T_1,\ldots,l_n:T_n\} & \text{(internal choice)}\\
| & \mu\alpha.T & \text{(recursive behavior)}\\
| & \alpha & \text{(type variable)}
\end{array}
\qquad
\begin{array}{rll}
S ::= & \mathit{int} & \text{(basic type)}\\
| & [T] & \text{(session reference)}
\end{array}
$$

Fig. 3. Syntax of types

$$
\begin{array}{llll}
\overline{\&\{l_i:T_i\}_{i\in I}} = \oplus\{l_i:\overline{T_i}\}_{i\in I} & \overline{?(\tilde{S}).T} = !(\tilde{S}).\overline{T} & \overline{\alpha} = \alpha & \overline{\texttt{end}} = \texttt{end}\\
\overline{\oplus\{l_i:T_i\}_{i\in I}} = \&\{l_i:\overline{T_i}\}_{i\in I} & \overline{!(\tilde{S}).T'} = ?(\tilde{S}).\overline{T'} & \overline{\mu\alpha.T} = \mu\alpha.\overline{T} &
\end{array}
$$

Fig. 4. The syntactic dual of a session type

The definition of the service a is duplicated and despite of their different syntactic structure both protocols are compatible with the one of the only client (in the sense of subtyping relation, see Definition 2). They are in fact both able to recursively offer l (the first one by means of a choice which offers l *twice*) and then to select one of the labels offered by client (either l_1 or l_2 in the case of the first service definition for a, depending on which l-branch is used; either l_1 or l_3 in the case of the second definition, depending on some internal condition *test*). Each declaration of the service a is one shot and is consumed once it is used, however it could be made persistent by the help of recursion (e.g. $\texttt{rec}\ X.a.P|X$).

3 Type system

We assume an infinite collection $\alpha, \beta, \ldots$ of type variables. The set of types is defined by the grammar in Fig. 3, where we distinguish session types T, U, V and types for message contents S.

The types $?(S_1,\ldots,S_n).T$ and $!(S_1,\ldots,S_n).T$ represent respectively an input and an output of n-tuples (whose i-th element has type S_i), followed by the continuation T. The types $\&\{l_1 : T_1,\ldots,l_n : T_n\}$ and $\oplus\{l_1 : T_1,\ldots,l_n : T_n\}$ denote respectively an external and an internal choice. The external choice describes the options offered to the partner while the internal choice expresses the selections a partner may wish to perform. Type $\mu\alpha.T$ represents a recursive behavior and μ is a binder that gives rise, in the standard way, to notions of bound (bv) and free type variables (fv), closed μ-types, and equivalence of μ-types up to alpha-renaming of bound variables. We let the function $\mathsf{unfold}(T)$ defined as $\mathsf{unfold}(\mu\alpha.T) = \mathsf{unfold}(T[^{\mu\alpha.T}/_{\alpha}])$ and $\mathsf{unfold}(T) = T$ otherwise. We often write $\tilde{T}$ for a sequence $T_1,\ldots,T_n$ of types and $\diamondsuit\{l_i : T_i\}_{i\in I}$ for $\diamondsuit\{l_{i_1} : T_{i_1},\ldots,l_{i_n} : T_{i_n}\}$ where $I = \{i_1,\ldots,i_n\} \neq \emptyset$ and $\diamondsuit \in \{\oplus,\&\}$. Basic data values have type int, and $[T]$ is the type of a service whose protocol has type T.

Definition 1 (Well formed types). *A type T is well formed if it is contractive (i.e., if for any subexpression of T of the form $\mu\alpha.\mu\alpha_1 \ldots \mu\alpha_n.T$ the body T is not α, see e.g. [8]) and if the set of labels of each choice are different (i.e., in any $\Diamond\{l_i : T_i\}_{i\in I}$ if $i \neq j$ then $l_i \neq l_j$).*

For simplicity we call types only well formed types and denote with $TYPE$ the set of all (well formed) types. The fact that the type of a choice must have different labels is not valid for a process which can offer the same labels provided that each branch has the same type (see Example 1). Each session type T has a syntactic dual type $\overline{T}$, recursively defined by the equations in Fig. 4.

Example 2. Intuitively, both session types $T_{ser} = \mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha\}\}$ and $T_{ser'} = \mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha, l_4 : \alpha\}\}$ are suitable types for (the protocol of) the service a of our running example, while the session type $T_{ser''} = \mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha\}\}$ is not suitable, because it does not take into account the possible selection of l_3 in the second definition for a.

The two possible types for service a introduced in the previous example are in subtyping relation. The subtype relation for session types was introduced by Gay and Hole in [11]. Here we consider a tighter notion of subtyping since it does not allow co-inductive subtyping of the sent and received values but it requires syntactic equality instead. This allows an elegant algorithm of constraints resolution simply using the most general unifier (see Section 4.2).

Definition 2. *A relation $R \subseteq TYPE \times TYPE$ is a type simulation relation if whenever $(T, V) \in R$ all of the following conditions hold:*

- *if* $\mathsf{unfold}(T) = !(\tilde{S}).T'$ *then* $\mathsf{unfold}(V) = !(\tilde{S}).V'$ *and* $(T', V') \in R$
- *if* $\mathsf{unfold}(T) = ?(\tilde{S}).T'$ *then* $\mathsf{unfold}(V) = ?(\tilde{S}).V'$ *and* $(T', V') \in R$
- *if* $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \&\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *with* $m \leq n$ *and for all* $i \in \{1, \ldots, m\}$ *there exists a (unique)* $j \in \{1, \ldots, n\}$ *such that* $l_i = l'_j$*, and for such* j *we have* $(T_i, V_j) \in R$
- *if* $\mathsf{unfold}(T) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \oplus\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *with* $n \leq m$ *and for all* $i \in \{1, \ldots, n\}$ *there exists a (unique)* $j \in \{1, \ldots, m\}$ *such that* $l'_i = l_j$*, and for such* j *we have* $(T_i, V_j) \in R$
- *if* $\mathsf{unfold}(T) = \mathtt{end}$ *then* $\mathsf{unfold}(V) = \mathtt{end}$

The coinductive relation $\leq$ is defined by letting $T \leq V$ if there is a type simulation relation R such that $(T, V) \in R$. We write $T \lesseqgtr V$ if $T \leq V$ and $V \leq T$.

Proposition 1. *$T \leq V$ iff $\overline{V} \leq \overline{T}$.*

Proof. We just prove that $R = \{(\overline{V}, \overline{T}) | T \leq V\}$ is a type simulation relation. □

Proposition 2. *$\leq$ is a preorder (i.e. it is reflexive and transitive).*

Our type system exploit two different typing environments: the *standard typing environment Γ* is a partial mapping (with domain dom) from polarized session names, service names and variables to session types, and the *process*

$$\text{(Ser)}\ \Gamma, s : S \vdash s : S \qquad \text{(Var)}\ \Gamma, x : S \vdash x : S \qquad \text{(IntV)}\ \Gamma \vdash n : int \qquad \text{(Tnil)}\ \Gamma; \Theta \vdash \mathbf{0} : \mathsf{end}; \mathsf{end}; \emptyset$$

$$\text{(Tnew)}\ \frac{\Gamma, s : S; \Theta \vdash P : T; U; L}{\Gamma; \Theta \vdash (\nu s)P : T; U; L} \qquad \text{(TnewR)}\ \frac{\Gamma, r^+ : [T], r^- : [\overline{T}]; \Theta \vdash P : T'; U; L}{\Gamma; \Theta \vdash (\nu r)P : T'; U; L \setminus \{r^+, r^-\}}$$

$$\text{(Tif)}\ \frac{\Gamma \vdash v_i : S \quad i = 1,2 \quad \Gamma; \Theta \vdash P : T; U; L_1 \quad \Gamma; \Theta \vdash Q : T; U; L_2}{\Gamma \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : T; U; L_1 \cup L_2}$$

$$\text{(Trec)}\ \frac{\Gamma; \Theta, X : T; U \vdash P : T'; U'; \emptyset \quad T \lesseqgtr T' \quad U \lesseqgtr U'}{\Gamma; \Theta \vdash \mathtt{rec}\ X.P : T; U; \emptyset}$$

$$\text{(Tpvar)}\ \Gamma; \Theta, X : T; U \vdash X : T; U; \emptyset \qquad \text{(Tses)}\ \frac{\Gamma; \emptyset \vdash P : T; U; L}{\Gamma, r^p : [T]; \Theta \vdash r^p \rhd P : U; \mathsf{end}; L \uplus \{r^p\}}$$

$$\text{(Tdef)}\ \frac{\Gamma; \emptyset \vdash P : T; U; L \quad \Gamma \vdash s : [T]}{\Gamma; \Theta \vdash s.P : U; \mathsf{end}; L} \qquad \text{(Tinv)}\ \frac{\Gamma; \emptyset \vdash P : T; U; L \quad \Gamma \vdash v : [\overline{T}]}{\Gamma; \Theta \vdash \overline{v}.P : U; \mathsf{end}; L}$$

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

$$\text{(Tbranch)}\ \frac{\emptyset \subset J \subseteq I = \{1, \dots, n\} \quad \forall i \in I,\ \Gamma; \Theta \vdash P_i : T_i; U; L_i}{\Gamma; \Theta \vdash \Sigma_{i=1}^{n}(l_i).P_i : \&\{l_j : T_j\}_{j \in J}; U; \bigcup_{i \in I} L_i} \qquad \text{(Tchoice)}\ \frac{l = l_i \in \{l_1, \dots, l_n\} \quad \Gamma; \Theta \vdash P : T_i; U; L}{\Gamma; \Theta \vdash \langle l \rangle.P : \oplus\{l_1 : T_1, \dots, l_n : T_n\}; U; L}$$

$$\text{(TparL)}\ \frac{\Gamma; \Theta \vdash P : T; U; L_1 \quad \Gamma; \Theta \vdash Q : \mathsf{end}; \mathsf{end}; L_2}{\Gamma; \Theta \vdash P | Q : T; U; L_1 \uplus L_2} \qquad \text{(TparR)}\ \frac{\Gamma; \Theta \vdash P : \mathsf{end}; \mathsf{end}; L_1 \quad \Gamma; \Theta \vdash Q : T; U; L_2}{\Gamma; \Theta \vdash P | Q : T; U; L_1 \uplus L_2}$$

Fig. 5. Typing rules

typing environment Θ is a partial mapping from process variables to pairs $T; U$ of session types (as we shall show in Proposition 4, at the least one type of each pair in Θ is recursive). As usual, the environment $\Theta' = \Theta, X : T; U$ is defined only if $X \notin \mathsf{dom}(\Theta)$ with $\Theta'(Y) = T; U$ if $Y = X$ and $\Theta'(Y) = \Theta(Y)$ otherwise (and analogously for Γ).

The typing judgments reported in Fig. 5 have the form $\Gamma, \Theta \vdash P : T; U; L$ where T is the type of the behaviour of P w.r.t. the current session, U is the type of the behaviour of P w.r.t. the parent session and L is a set of polarized session names, which is used to check the linearity of sessions (see rules (TparL) and (TparR) which allow each session side to appear in only one place). The operation of $L_1 \uplus L_2$ is defined only if the two sets are disjoint and it is equal to $L_1 \cup L_2$.

Rule (TnewR) guesses two dual types for each session sides whilst (Tnew) guesses only the type of the service s. This choice reflects the asymmetry between client and service and the symmetry between two dual session sides. Rule (Tdef) constrains the protocol of the service to be the same as the body type of the process P and (Tinv), instead checks that the dual type of the client protocol is the same as the service protocol. Rule (TSes) is similar to (Tdef) but in addition it handles

$$
\dfrac{\dfrac{\dfrac{\dfrac{\mathit{\Gamma};\mathit{\Theta} \vdash Y_2 : T_\alpha;\texttt{end}}{\mathit{\Gamma};\mathit{\Theta} \vdash \langle l_2\rangle.Y_2 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha\};\texttt{end}} \qquad \dfrac{\mathit{\Gamma};\mathit{\Theta} \vdash Y_2 : T_\alpha;\texttt{end}}{\mathit{\Gamma};\mathit{\Theta} \vdash \langle l_1\rangle.Y_2 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha\};\texttt{end}}}{\mathit{\Gamma};\mathit{\Theta} \vdash \langle l_1\rangle.Y_2 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\texttt{end}}}{\mathit{\Gamma};\mathit{\Theta} \vdash \mathit{\Sigma}_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : \&\{l : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\}\};\texttt{end}}}{\mathit{\Gamma};\emptyset \vdash a.\texttt{rec}\ Y_2.\mathit{\Sigma}_{i=1}^{2}(l).\langle l_i\rangle.Y_2;\texttt{end};\texttt{end}}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{\Gamma};\mathit{\Theta}_1 \vdash Y_1 : T_\alpha;\texttt{end}}{\mathit{\Gamma};\mathit{\Theta}_1 \vdash \langle l_3\rangle.Y_1 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\texttt{end}} \qquad \dfrac{\mathit{\Gamma};\mathit{\Theta}_1 \vdash Y_1 : T_\alpha;\texttt{end}}{\mathit{\Gamma};\mathit{\Theta}_1 \vdash \langle l_1\rangle.Y_1 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\texttt{end}}}{\mathit{\Gamma};\mathit{\Theta}_1 \vdash \texttt{if}\ \mathit{test}\ \texttt{then}\ \langle l_1\rangle.Y_1\ \texttt{else}\ \langle l_3\rangle.Y_1 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\texttt{end}}}{\mathit{\Gamma};\mathit{\Theta}_1 \vdash (l).\texttt{if}\ \mathit{test}\ \texttt{then}\ \langle l_1\rangle.Y_1\ \texttt{else}\ \langle l_3\rangle.Y_1 : \&\{l : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\}\};\texttt{end}}}{\mathit{\Gamma};\emptyset \vdash \texttt{rec}\ Y_1.(l).\texttt{if}\ \mathit{test}\ \texttt{then}\ \langle l_1\rangle.Y_1\ \texttt{else}\ \langle l_3\rangle.Y_1 : \mathsf{unfold}(T_\alpha);\texttt{end}}}{\mathit{\Gamma};\emptyset \vdash a.\texttt{rec}\ Y_1.(l).\texttt{if}\ \mathit{test}\ \texttt{then}\ \langle l_1\rangle.Y_1\ \texttt{else}\ \langle l_3\rangle.Y_1 : \texttt{end};\texttt{end}}
$$

Fig. 6. Example of typing

the linearity check on r^p w.r.t. the set L. Rule (Tin), (Tout) and (Tret) insert either the input or the output in the correct place. Rule (Tbranch) allows to insert in the result type only a subset J of the total choices, while rule (Tchoice) add arbitrarily branches in the type together with the branch labeled with l. Both this rules are not syntax directed and together with (Trec) allow flexibility in the protocol specification of each side, for example we can correctly type $\mu\alpha.\&\{l : \texttt{end}, l_1 : \alpha\}$ against both $\oplus\{l_1 : \oplus\{l : \texttt{end}\}\}$ and $\mu\alpha.\oplus\{l_1 : \alpha\}$ (i.e. we can find a typing that makes these protocols dual).

Example 3. In Fig. 6 we report the proof tree of the two definitions of service a where the set L is omitted for conciseness (it is equal to $\emptyset$) and $T_\alpha = \mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha\}\}$, $\mathit{\Gamma} = a : [T_\alpha]$, $\mathit{\Theta} = Y_2 : T_\alpha;\texttt{end}$ and $\mathit{\Theta}_1 = Y_1 : T_\alpha;\texttt{end}$.

Subject reduction is proved with the help of standard Weakening, Strengthening and Substitution lemmas together with subject congruence (cf. Appendix for their statements). In addition, we need auxiliary lemmas to deal with the presence of contexts.

Lemma 1 (Replacement $\mathbb{C}$). *If $\mathit{\Gamma};\emptyset \vdash \mathbb{C}[\![P]\!] : T;U;L$ then there exist some T',U',L' and $\mathit{\Gamma}'$ s.t. $\mathit{\Gamma}';\emptyset \vdash P : T';U';L'$ with $L' \subseteq L$. Moreover for any Q, s.t. $\mathit{\Gamma}';\emptyset \vdash Q : T';U';L''$ with $L'' \cap L = L'$ then $\mathit{\Gamma};\emptyset \vdash \mathbb{C}[\![Q]\!] : T;U;L \cup L''$.*

Session linearity asserts that in a well-typed process no duplicate session sides are present.

Lemma 2 (Session linearity). *If $\mathit{\Gamma},\mathit{\Theta} \vdash P : T;U;L$ then for any $\mathbb{C}, P', r^p$, s.t. $P \equiv (\nu\tilde{m})\mathbb{C}[\![r^p \rhd P']\!]$ we have $r^p \notin \mathsf{fn}(\mathbb{C}[\![P']\!])$*

Next lemma is necessary to prove the replacement lemma for contexts $\mathbb{D}_r$. Let $\subseteq$ denote the smallest relation on $TYPE \times TYPE$ such that: $T \subseteq ?(\tilde{S}).T$,$T \subseteq !(\tilde{S}).T$, $T_i \subseteq \&\{l_1 : T_1, \ldots, l_n : T_n\}$ and $T_i \subseteq \oplus\{l_1 : T_1, \ldots, l_n : T_n\}$ we have:

Lemma 3 (Replacement $\mathbb{D}_r$). *If $\Gamma, r^p : [T_r], r^{\overline{p}} : [\overline{T_r}]; \emptyset \vdash \mathbb{D}_r[P,Q] : T;U;L$ then there exist some Γ_1, U_1, $L_1 \subseteq L$ and Γ_2, U_2, $L_2 \subseteq L$ s.t. $\Gamma_1;\emptyset \vdash P : T_r;U_1;L_1$ and $\Gamma_2;\emptyset \vdash Q : \overline{T_r};U_2;L_2$. Moreover for any P' and Q' s.t. $\Gamma_1;\emptyset \vdash P' : T_r^{\subseteq};U_1;L_1$ and $\Gamma_2;\emptyset \vdash Q' : \overline{T_r^{\subseteq}};U_2;L_2$ then $\Gamma, r^p : [T_r^{\subseteq}], r^{\overline{p}} : [\overline{T_r^{\subseteq}}]; \emptyset \vdash \mathbb{D}_r[P',Q'] : T;U;L$ for some $T_r^{\subseteq} \subseteq T_r$.*

Theorem 1 (Subject reduction). *If $\Gamma;\emptyset \vdash P : T;U;L$ and $P \to P'$ then there exist Γ', L' s.t. $\Gamma';\emptyset \vdash P' : T;U;L'$*

As it is common for session type systems, our type system enjoys sessions safety property. An error is a process which has incompatible kinds of communications in a dual sessions e.g. $\mathbb{D}[\![\mathbb{C}_{r^p}[\![(\tilde{x}).P']\!], \mathbb{C}_{r^{\overline{p}}}[\![(\tilde{x}).P'']\!]]\!]$, $\mathbb{D}[\![\mathbb{C}_{r^p}[\![(\tilde{x}).P']\!], \mathbb{C}_{r^{\overline{p}}}[\![\Sigma_{i=1}^n(l_i).P_i]\!]]\!]$, $\mathbb{D}[\![\mathbb{C}_{r^p}[\![(\tilde{x}).P']\!], \mathbb{C}_{r^{\overline{p}}}[\![\langle l\rangle.P'']\!]]\!]$ and so on for the other mismatching prefixes. A process P has an error if $P \equiv (\nu\tilde{n})\mathbb{D}[\![\mathbb{C}_{r^p}[\![P']\!], \mathbb{C}_{r^{\overline{p}}}[\![P'']\!]]\!]$ and $\mathbb{D}[\![\mathbb{C}_{r^p}[\![P']\!], \mathbb{C}_{r^{\overline{p}}}[\![P'']\!]]\!]$ is an error.

Theorem 2 (Safety). *Let $\Gamma, \emptyset \vdash P : T;U;L$ then P never reduces to process with an error.*

Next proposition allows to relax the type of the current session.

Proposition 3 (Type weakening). *If $\Gamma;\Theta \vdash P : T;U;L$ and $T' \leq T$ then $\Gamma;\Theta \vdash P : T';U;L$*

Corollary 1. *Let $T \lesseqgtr T'$. $\Gamma;\Theta \vdash P : T';U;L$ iff $\Gamma;\Theta \vdash P : T;U;L$*

and finally a technical proposition which will be useful later in the next section since it shows all the possible types of a well typed recursive process.

Proposition 4. *If $\Gamma;\Theta \vdash \texttt{rec}\ X.Q : T;U;\emptyset$ and $X \in \mathsf{fpv}(Q)$ then either $T = \texttt{end}$ and $U \lesseqgtr \mu\alpha.U'$ or $T \lesseqgtr \mu\alpha.T'$ and $U \lesseqgtr \texttt{end}$ or $T \lesseqgtr \mu\alpha.T'$ and $U \lesseqgtr \mu\alpha.U'$ or $T \lesseqgtr \texttt{end}$ and $U \lesseqgtr \texttt{end}$.*

Proof. The proof is given by induction on the judgment of $\texttt{rec}X.Q$. We sketch some cases: **Base Case:** The only possible case is when the last applied rule is (Tpvar). Then we have: $\dfrac{\Gamma;\Theta, X : \texttt{end};\texttt{end} \vdash X : \texttt{end};\texttt{end};\emptyset}{\Gamma;\Theta \vdash \texttt{rec}\ X.X : \texttt{end};\texttt{end};\emptyset}\ (\textsc{Trec})$. **Inductive Cases:** The last applied rule is (Tin). By induction hypothesis we have (given that $X \in \mathsf{fpv}(P)$) $\Gamma;\Theta \vdash \texttt{rec}X.P : T_1;U_1;L$ and either $T_1 \lesseqgtr \texttt{end}$ then $T \lesseqgtr \mu\alpha.?(\tilde{S}).\alpha$ or $T_1 \lesseqgtr \mu\alpha.T_1'$ then $T \lesseqgtr \mu\alpha.?(\tilde{S}).T_1'$. In particular for the latter case we have $\dfrac{\Gamma, \tilde{x} : \tilde{S};\Theta, X : \mu\alpha.T_1';U \vdash P : \mu\alpha.T_1';U;\emptyset}{\Gamma \vdash (\tilde{x}).P : ?(\tilde{S}).\mu\alpha.T_1';U;\emptyset}\ (\textsc{Tin})$ then in order to have $\mu\alpha.T_1' \lesseqgtr ?(\tilde{S}).\mu\alpha.T_1'$ it must be $\mu\alpha.T_1' \lesseqgtr \mu\alpha.?(\tilde{S}).T_1'$. The other cases are similar but the case for (Tbranch) requires an inner induction on the number of branches. $\square$

(TnilSD)
$$\Gamma;\Theta \vdash_{\mathsf{SD}} \mathbf{0} : \{\mathtt{end}\}; \mathtt{end}; \emptyset$$

(TnewSD)
$$\frac{\Gamma, s : S;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L}{\Gamma;\Theta \vdash_{\mathsf{SD}} (\nu s)P : \mathcal{T}; U; L}$$

(TnewRSD)
$$\frac{\Gamma, r^+ : [T], r^- : [\overline{T}];\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L}{\Gamma;\Theta \vdash_{\mathsf{SD}} (\nu r)P : \mathcal{T}; U; L \setminus \{r^+, r^-\}}$$

(TifSD)
$$\frac{\Gamma \vdash_{\mathsf{SD}} v_i : S \quad i = 1,2 \quad \Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}_1; U; L_1 \quad \Gamma;\Theta \vdash_{\mathsf{SD}} Q : \mathcal{T}_2; U; L_2}{\Gamma \vdash_{\mathsf{SD}} \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : \mathcal{T}_1 \cup \mathcal{T}_2; U; L_1 \cup L_2}$$

(TrecSD)
$$\frac{\Gamma;\Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{SD}} P : \mathcal{T}; U; \emptyset}{\Gamma;\Theta \vdash_{\mathsf{SD}} \mathtt{rec}\ X.P : \mu\alpha_X(\mathcal{T}); \mu\alpha_X(U); \emptyset}$$

(TpvarSD)
$$\Gamma;\Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{SD}} X : \alpha_X; \alpha_X; \emptyset$$

(TsesSD)
$$\frac{\Gamma;\emptyset \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L \quad T \leq \mathcal{T}}{\Gamma, r^p : [T];\Theta \vdash_{\mathsf{SD}} r^p \triangleright P : U; \{\mathtt{end}\}; L \uplus \{r^p\}}$$

(TdefSD)
$$\frac{\Gamma;\emptyset \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L \quad \Gamma \vdash s : [T] \quad T \leq \mathcal{T}}{\Gamma;\Theta \vdash_{\mathsf{SD}} s.P : \{U\}; \mathtt{end}; L}$$

(TinvSD)
$$\frac{\Gamma;\emptyset \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L \quad \Gamma \vdash v : [T] \quad \overline{\mathcal{T}} \leq T}{\Gamma;\Theta \vdash_{\mathsf{SD}} \overline{v}.P : \{U\}; \mathtt{end}; L}$$

(TinSD)
$$\frac{\Gamma, \tilde{x} : \tilde{S};\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L}{\Gamma \vdash_{\mathsf{SD}} (\tilde{x}).P : ?(\tilde{S}).\mathcal{T}; U; L}$$

(ToutSD)
$$\frac{\Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L \quad \Gamma \vdash_{\mathsf{SD}} \tilde{v} : \tilde{S}}{\Gamma;\Theta \vdash_{\mathsf{SD}} \langle\tilde{v}\rangle.P : !(\tilde{S}).\mathcal{T}; U; L}$$

(TbranchSD)
$$\frac{\emptyset \subset J \subseteq I = \{1,\ldots,n\} \quad \forall i \in I,\ \Gamma;\Theta \vdash_{\mathsf{SD}} P_i : \mathcal{T}_i; U; L_i}{\Gamma;\Theta \vdash_{\mathsf{SD}} \Sigma_{i=1}^{n}(l_i).P_i : \mathsf{split}(\&\{l_j : \mathcal{T}_j\}_{j\in I}); U; \bigcup_{j\in J} L_i}$$

(TchoiceSD)
$$\frac{\Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L}{\Gamma;\Theta \vdash_{\mathsf{SD}} \langle l\rangle.P : \oplus\{l : \mathcal{T}\}; U; L}$$

(TparLSD)
$$\frac{\Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L_1 \quad \Gamma;\Theta \vdash_{\mathsf{SD}} Q : \{\mathtt{end}\}; \mathtt{end}; L_2}{\Gamma;\Theta \vdash_{\mathsf{SD}} P|Q : \mathcal{T}; U; L_1 \uplus L_2}$$

(TparRSD)
$$\frac{\Gamma;\Theta \vdash_{\mathsf{SD}} P : \{\mathtt{end}\}; \mathtt{end}; L_1 \quad \Gamma;\Theta \vdash_{\mathsf{SD}} Q : \mathcal{T}; U; L_2}{\Gamma;\Theta \vdash_{\mathsf{SD}} P|Q : \mathcal{T}; U; L_1 \uplus L_2}$$

Fig. 7. Syntax directed typing rules

4 Type inference

The types system in Section 3 cannot be exploited directly for type inference, because three main sources of non-determinism that would require guesses and backtracking: the rule for choices (Tbranch) and (Tchoice) and the rule for recursion (Trec). Moreover, the type equality for the two branches in rule (Tif) is too demanding, and difficult to infer by considering the two branches separately.

4.1 Syntax directed rules

In this section we introduce a syntax-directed type system to overcome the above mentioned sources of non determinism. The typing judgements in Fig. 7 have the form $\Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L$ where $\mathcal{T}$ is a set of (possibly not closed) session types.

The basic intuitions behind this type system are two. First, the set $\mathcal{T}$ is obtained multiplexing each branch of the if-then-else (see rule (TifSD)), and second recursion is typed using a type variable with the name dependent of the current process variable, to mean that its type is unknown (see rules (TrecSD)). Rules (TBranchSD) and (TChoiceSD) are now fully syntax directed but rules (TdefSD) and (TinvSD) require a little care. In particular, we introduce the subtyping relation to relax the service protocol w.r.t. the actual protocol specified in the process.

$$\diamondsuit\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\} = \{\diamondsuit\{l_1 : T_1, \ldots l_n : T_n\} \mid i \in \{1, \ldots, n\} \text{ and } T_i \in \mathcal{T}_i\}$$
$$!(\tilde{S}).\mathcal{T} = \{!(\tilde{S}).T \mid T \in \mathcal{T}\} \qquad ?(\tilde{S}).\mathcal{T} = \{?(\tilde{S}).T \mid T \in \mathcal{T}\} \qquad \mu\alpha(\mathcal{T}) = \{\mu\alpha(T) \mid T \in \mathcal{T}\}$$
$$\mathcal{T} \leq T \Leftrightarrow \forall T' \in \mathcal{T}.T' \leq T \qquad T \leq \overline{\mathcal{T}} \Leftrightarrow \forall T' \in \mathcal{T}.T \leq \overline{T'}$$
$$\mu\alpha(T) = \begin{cases} \mu\alpha.T & \text{if } \alpha \in \mathsf{fv}(T) \wedge \mu\alpha.T \text{ is contractive} \\ \mathtt{end} & \text{if } \alpha \in \mathsf{fv}(T) \wedge \mu\alpha.T \text{ is not contractive} \\ T & \text{otherwise} \end{cases}$$
$$\&\{l : T\} \,;;\, \&\{l_1 : T_1, \ldots, l_n : T_n\} = \begin{cases} \&\{l_1 : T_1, \ldots, l : T, \ldots, l_n : T_n\} & \text{if } l \notin \{l_1, \ldots, l_n\} \\ \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{if } l = l_i \wedge T = T_i \text{ for some } i \\ \textit{undefined} \text{ otherwise} \end{cases}$$
$$\mathsf{split}(\&\{l_i : \mathcal{T}_i\}_{i \in I}) = \bigcup_{U \in \mathcal{T}_i} (\&\{l_i : U\} \,;;\, \mathsf{split}(\&\{l_j : \mathcal{T}_j\}_{j \in J})) \text{ where } J = I \setminus \{i\}$$
$$\mathsf{fv}(\mathcal{T}) = \bigcup_{T' \in \mathcal{T}} \mathsf{fv}(T')$$
$$\mathsf{fv}(\emptyset) = \emptyset \qquad \mathsf{fv}(\Theta, X : \alpha_X ; \alpha_X) = \{\alpha_X\} \cup \mathsf{fv}(\Theta) \qquad \mathsf{fv}(\Theta, X : T; U) = \mathsf{fv}(\Theta) \text{ if } T;U \neq \alpha_X ; \alpha_X$$

Fig. 8. Auxiliary functions for sets of types (where $\diamondsuit \in \{\&, \oplus\}$)

This check, e.g., allows to offer more options for internal choice than the choices actually needed. The subtyping relation between sets and type is defined as expected in Fig. 8. Note that the subtyping check is inverted in rule (TinvSD) due to the property in Proposition 1, but it is not related to the contravariance [17] of output (not present here because we require the syntactic equality of i/o value types). Let us focus on the rule (TrecSD) for recursive processes. The type of the process P is computed by means of a type variable inserted in Θ. The resulting type is computed with the help of the function $\mu\alpha(T)$ (see Fig. 8) which checks the possibility to close the variable α in the type T. We have three possibilities depending on whether α is free in T and the resulting type $\mu\alpha.T$ is contractive. Note that this rule captures all the possibilities of Proposition 4 (the type systems are similar when $\mathcal{T}$ is a singleton).

Some auxiliary functions used in the type system are also in Fig. 8. They are all simple (homomorphic extension of the operators in the type signature defined by pointwise application) but split, which deserves some explanations. Intuitively split computes the products of each branch in a choice and returns its split accordingly to all possible different combination of choices. For example $\mathsf{split}(\&\{l : \{T_1, T_2\}, l : T_3, l_1 : U\})$ returns $\{\&\{l : T_i, l_1 : U\} \mid i \in \{1, 2, 3\}\}$. Next proposition relates the free variables of the set $\mathcal{T}$ w.r.t the free variables in Θ.

Proposition 5. *If* $\Gamma; \Theta \vdash_{\mathsf{SD}} P : \mathcal{T}; U; L$ *then* $\mathsf{fv}(\mathcal{T}) \subseteq \mathsf{fv}(\Theta)$

Soundness and correctness are formulated following one basic intuition: each type contained in $\mathcal{T}$ obtained with $\vdash_{\mathsf{SD}}$ can simulate the corresponding T obtained with $\vdash$. This fact is evident, e.g. if we think of the if-then-else as a construct able to make the type of the current session stricter; that is the two types in each branches when considered alone are generally weaker then the total type of the if-then-else (if the process is typable). The function $\mathsf{forget}(\Theta)$ replaces the recursive type annotations with standard type variables s.t. $\mathsf{forget}(\Theta', X :$

$T;U) = \mathsf{forget}(\Theta'), X : \alpha_X;\alpha_X$ if either $T \lesseqgtr \mu\alpha.T'$ or $U \lesseqgtr \mu\alpha.U'$ and it is inductively defined as expected in the other cases.

To account for the free variables present in a type during the typing derivation we introduce some auxiliary syntax used in the proof of the theorem, called the $\bullet$ machinery. The idea is that the special symbol $\bullet$ replaces a free variable in the type. First of all we add a production for $\bullet$ both as a process and as a session type with the two obvious axioms $\Gamma;\Theta \vdash \bullet : \bullet;\bullet;\emptyset$ and $\Gamma;\Theta \vdash_{\mathsf{SD}} \bullet : \{\bullet\};\bullet;\emptyset$. Moreover we add a rule in the subtyping relation which asserts that "if $\mathsf{unfold}(T) = \bullet$ then $\mathsf{unfold}(V) = \bullet$", e.g. $\bullet \leq \bullet$. An useful properties of the $\bullet$ machinery is stated in the following proposition:

Proposition 6. *Let T_1 and T_2 two types s.t. $\mathsf{fv}(T_1) = \{\alpha_1\}$ and $\mathsf{fv}(T_2) = \{\alpha_2\}$. If $T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$ then $\mu\alpha_1.T_1 \leq \mu\alpha_2.T_2$.*

Proof. It suffices to prove that $R = \{(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}], T_2[^{\mu\alpha_2.V_2}/_{\alpha_2}])|T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$ and $V_1[^{\bullet}/_{\alpha_1}] \leq V_2[^{\bullet}/_{\alpha_2}]\}$ is a type simulation relation, which contains both $T_1[^{\mu\alpha_1.T_1}/_{\alpha_1}]$ and $T_2[^{\mu\alpha_2.T_2}/_{\alpha_2}]$. ☐

Next, we formulate the theorems of soundness and completeness assuming the $\bullet$ machinery is present in the framework.

Theorem 3 (Completeness of the syntax directed rules). *If $\Gamma;\Theta \vdash P : T;U;L$ then $\exists\mathcal{T}, U'$ s.t $\Gamma;\Theta' \vdash_{\mathsf{SD}} P : \mathcal{T};U';L$ and $\mathsf{forget}(\Theta) = \Theta'$ and for all $T' \in \mathcal{T}$ if $\mathsf{fv}(T') = \emptyset$ then $T \leq T'$ and if $\mathsf{fv}(U') = \emptyset$ then $U = U'$*

Proof. The proof is by induction on the $\vdash$-typing judgments. About rules (Trec), we use either Strengthening Lemma or we need case analysis, one case for each of the conclusions of the Proposition 4. For example if only T is recursive we have: $P = \mathtt{rec}\, X.P'$ and $\Gamma;\Theta, X : \mu\alpha.T';\mathtt{end} \vdash P' : T'[^{\mu\alpha.T'}/_{\alpha}];\mathtt{end};\emptyset$. From the $\vdash_{\mathsf{SD}}$ side we are interested to all $T'' \in \mathcal{T}'$ s.t. $\mathsf{fv}(T'') = \{\alpha_X\}$ and $\Gamma;\Theta', X : \alpha_X;\alpha_X \vdash_{\mathsf{SD}} P' : \mathcal{T}';\mathtt{end};\emptyset$. Now, consider the process $\mathtt{rec}\, X.(P'[^{\bullet}/_X])$. If $P'[^{\bullet}/_X]$ can be typed we would obtain $T'[^{\bullet}/\alpha]$ and by induction hypothesis take $\mathcal{T}'[^{\bullet}/_{\alpha_X}]$ (where the substitution is applied pointwise) in $\vdash_{\mathsf{SD}}$. Then for all $T'' \in \mathcal{T}'$, $T'[^{\bullet}/_{\alpha}] \leq T''[^{\bullet}/_{\alpha_X}]$ and the result follows applying Proposition 6. ☐

Theorem 4 (Soundness of the syntax directed rules). *Let $\Gamma;\Theta' \vdash_{\mathsf{SD}} P : \mathcal{T};U;L$. For all T s.t. for all $T' \in \mathcal{T}$ we have $\mathsf{fv}(T') = \emptyset$ and $T \leq T'$ then $\Gamma;\Theta \vdash P : T;U';L$ for some U' and $\mathsf{forget}(\Theta) = \Theta'$, and if $\mathsf{fv}(U') = \emptyset$ then $U = U'$*

Proof. The proof is by induction on the $\vdash_{\mathsf{SD}}$-typing judgments. We sketch the case where the last applied rule is (TdefSD). We have $P = s.P'$ and $\Gamma;\emptyset \vdash_{\mathsf{SD}} P' : \mathcal{T}';U'';L$ with $\Gamma \vdash s : [T_1]$ and $T_1 \leq T_1'$ for all $T_1' \in \mathcal{T}'$ and $\mathsf{fv}(T_1') = \emptyset$ by Proposition 5. By inductive hypothesis $\Gamma;\emptyset \vdash P' : T'';U'';L$ if $T'' \leq T_1'$ for all $T_1' \in \mathcal{T}'$. Since for all T_1' also holds $T_1 \leq T_1'$, we can conclude $\Gamma;\emptyset \vdash P' : T_1;U;L$ satisfying the premises of (Tdef).About rule (TrecSD), we use either the Weakening Lemma, or Proposition 6. ☐

Example 4. The first instance of service a in Example 1 is typed as below, where $\mathcal{T} = \mathsf{split}(\&\{l : \{\oplus\{l_1 : \alpha_{Y_2}\}, \oplus\{l_2 : \alpha_{Y_2}\}\}\}) = \{\&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\}, \&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\}\}$ and $\Theta = \alpha_{Y_2}; \alpha_{Y_2}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Theta \vdash Y_2 : \alpha_{Y_2}; \mathtt{end}}{\Gamma; \Theta \vdash \langle l_1 \rangle.Y_2 : \oplus\{l_1 : \alpha_{Y_2}\}; \mathtt{end}} \qquad \dfrac{\Gamma; \Theta \vdash Y_2 : \alpha_{Y_2}; \mathtt{end}}{\Gamma; \Theta \vdash \langle l_2 \rangle.Y_2 : \oplus\{l_2 : \alpha_{Y_2}\}; \mathtt{end}}}{\Gamma; \Theta \vdash \Sigma_{i=1}^{2}(l).\langle l_i \rangle.Y_2 : \mathcal{T}; \mathtt{end}}}{}}{\Gamma; \emptyset \vdash \mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i \rangle.Y_2 : \mu\alpha_{Y_2}.\mathcal{T}; \mathtt{end}}$$

4.2 Extract and solve constraints

From the syntax directed rules we extract the algorithm `INF` reported in Fig. 9 (resp. `VALUEINF` for transmittable values). (In the algorithm $\{\mathcal{T} \leq T\}$ is a shorthand for $\{T' \leq T \mid T' \in \mathcal{T}\}$.) The algorithm takes a process P without free process variables (i.e. Θ is empty) and with every bound and free names different. Together with P the algorithm expects also a typing environment Γ containing type assumptions on free names (each assumption can be either a type variable or a type inserted for type checking purposes) and returns a set $\mathcal{C}$ of constraints over type variables, the set of types $\mathcal{T}$ of the current session and the type U of the parent session. In the following we use: σ, ρ for substitutions, the juxtaposition $\sigma\rho$ for substitution composition, σT for the result of the simultaneous substitution σ applied to the free variables of T and $\sigma\mathcal{T}$,$\sigma\mathcal{C}$ simultaneous substitution applied to each element in the set. We write $\sigma \vDash \mathcal{C}$ if σ is a substitution s.t. the entire set of constraints $\sigma\mathcal{C}$ holds.

Theorem 5 (Soundness and Correctness). *Let* $\mathtt{VALUEINF}(v, \Gamma) = (\mathcal{C}, S)$ *then* $\sigma \vDash \mathcal{C}$ *iff* $(\sigma\Gamma) \vdash v : \sigma S$. *Let* $\mathtt{INF}(P, \Gamma, \Theta) = (\mathcal{C}, \mathcal{T}, U, L)$ *then* $\sigma \vDash \mathcal{C}$ *iff* $(\sigma\Gamma); \Theta \vdash_{\mathsf{SD}} P : \sigma(\mathcal{T}); \sigma U; L$.

Next we show how to check the existence of a solution to the set of constraints $\mathcal{C}$ returned by the `INF` algorithm. Given two types T and U we let $\sigma_{T \leq U}$ denote the most general unifier, when it exists, which satisfies $\sigma_{T \leq U} T \leq \sigma_{T \leq U} U$ and for any σ' such that $\sigma' T \leq \sigma' U$ there exists ρ with $\sigma' = \rho\sigma_{T \leq U}$ (the subtyping algorithm is given in [11]). In the same way we let $\sigma_{T=U}$ denote the most general unifier, when it exists, of T and U and we let $\sigma_{T \wedge U}$ denote the most general unifier, when it exists, such that the meet operation $\sigma_{T \wedge U} T \wedge \sigma_{T \wedge U} U$ is defined (the meet algorithm is discussed in [4]). Let $(\mathcal{C}, \mathcal{T}, \_, \_) = \mathtt{INF}(P, \Gamma, \emptyset)$, we apply the following `solve` algorithm on $\mathcal{C} \cup \{(\alpha_t \leq T'') | T'' \in \mathcal{T}\}$ where α_t is fresh. The semantic of the `match` is Ocaml-like but with a slight abuse of notation we assume pattern-matching over sets is possible (while we had to implement it in the running version of the algorithm, see `http://www.di.unipi.it/~mezzina`).

$$\begin{array}{l}
\mathtt{let\ rec\ solve}\ \mathcal{C}\ \mathtt{=\ match}\ \mathcal{C}\ \mathtt{with} \\
\quad | T_1 = T_2, \mathcal{C}'\ \mathtt{->\ solve}\ \sigma_{T_1 = T_2}\mathcal{C}' \\
\quad | \alpha \leq T_1, \alpha \leq T_2, \mathcal{C}'\ \mathtt{->\ solve}\ \sigma_{T_1 \wedge T_2}\mathcal{C}' \cup \{\alpha \leq T\} \quad \mathtt{where}\ T = \sigma_{T_1 \wedge T_2} T_1 \wedge \sigma_{T_1 \wedge T_2} T_2 \\
\quad | T_1 \leq \alpha, T_2 \leq \alpha, \mathcal{C}'\ \mathtt{->\ solve}\ \sigma_{\overline{T_1} \wedge \overline{T_2}}\mathcal{C}' \cup \{T \leq \alpha\} \quad \mathtt{where}\ \overline{T} = \sigma_{\overline{T_1} \wedge \overline{T_2}} \overline{T_1} \wedge \sigma_{\overline{T_1} \wedge \overline{T_2}} \overline{T_2} \\
\quad | \alpha \leq T_1, T_2 \leq \alpha, \mathcal{C}'\ \mathtt{->\ solve}\ \mathcal{C}' \cup \{T_2 \leq T_1\} \cup \{\alpha \leq T_1\} \\
\quad | \alpha_1 = \overline{\alpha_1}, \alpha_1 \leq T_1, \alpha_2 \leq T_2, \mathcal{C}'\ \mathtt{->}\ \ \mathtt{solve}\ \mathcal{C}' \cup \{T_1 \leq \overline{T_2}\} \\
\quad | T \leq T_1, \mathcal{C}'\mathtt{->\ solve}\ \sigma_{T \leq T_1}\mathcal{C}' \\
\quad | _\ \mathtt{->}\ \mathit{success}
\end{array}$$

$\texttt{VALUEINF}(x, \Gamma)\texttt{=}\ (\emptyset, \Gamma(x)) \qquad \texttt{VALUEINF}(s, \Gamma)\texttt{=}\ (\emptyset, \Gamma(s)) \qquad \texttt{VALUEINF}(n, \Gamma)\texttt{=}\ (\emptyset, int)$

$\texttt{INF}(\mathbf{0}, \Gamma, \Theta)\texttt{=}\ (\emptyset, \{\mathsf{end}\}, \mathsf{end}, \emptyset)$

$\texttt{INF}(s.P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \emptyset)$
$\quad \texttt{in}\ (\mathcal{C} \cup \{\alpha \leq \mathcal{T}\} \cup \{\Gamma(s) = [\alpha]\}, \{U\}, \mathsf{end}, L) \quad \texttt{(where}\ \alpha\ \texttt{fresh)}$

$\texttt{INF}(\overline{v}.P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \emptyset)$
$\quad \texttt{in}\ (\mathcal{C} \cup \{\overline{\mathcal{T}} \leq \alpha\} \cup \{\Gamma(v) = [\alpha]\}, \{U\}, \mathsf{end}, L) \quad \texttt{(where}\ \alpha\ \texttt{fresh)}$

$\texttt{INF}((x_1, \ldots, x_n).P, \Gamma, \Theta)\texttt{=}$
$\quad \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma \cup \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}, \Theta) \quad \alpha_1, ., \alpha_n\ \texttt{fresh}$
$\quad \texttt{in}\ (\mathcal{C}, ?(\Gamma(x_1), \ldots, \Gamma(x_n)).\mathcal{T}, U, L)$

$\texttt{INF}(\langle v_1, \ldots, v_n\rangle.P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \Theta)\ \texttt{in}$
$\quad \texttt{let}\ (\mathcal{C}_1, S_1)\texttt{=VALUEINF}(v_1, \Gamma) \ldots\ldots (\mathcal{C}_n, S_n)\texttt{=VALUEINF}(v_n, \Gamma)$
$\quad \texttt{in}\ (\mathcal{C} \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n, !(S_1, \ldots, S_n).\mathcal{T}, U, L)$

$\texttt{INF(return}\ v_1, \ldots, v_n.P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \Theta)\ \texttt{in}$
$\quad \texttt{let}\ (\mathcal{C}_1, S_1)\texttt{=VALUEINF}(v_1, \Gamma) \ldots\ldots (\mathcal{C}_n, S_n)\texttt{=VALUEINF}(v_n, \Gamma)$
$\quad \texttt{in}\ (\mathcal{C} \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n, \mathcal{T}, !(S_1, \ldots, S_n).U, L)$

$\texttt{INF(if}\ v_1 = v_2\ \texttt{then}\ P\ \texttt{else}\ Q, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L_1)\texttt{=INF}(P, \Gamma, \Theta)\ \texttt{in}$
$\quad \texttt{let}\ (\mathcal{C}_1, \mathcal{T}_1, U_1, L_2)\texttt{=INF}(Q, \Gamma, \Theta)\ \texttt{in}$
$\quad \texttt{let}\ (\mathcal{C}_1, S_1)\texttt{=VALUEINF}(v_1, \Gamma) \ldots (\mathcal{C}_2, S_2)\texttt{=VALUEINF}(v_2, \Gamma)$
$\quad \texttt{in}\ (\mathcal{C} \cup \mathcal{C}_1 \cup \{U = U_1\} \cup \{S_1 = S_2\}, \mathcal{T} \cup \mathcal{T}_1, U, L_1 \uplus L_2)$

$\texttt{INF}((\nu s)P, \Gamma, \Theta)\texttt{=}$
$\quad \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma \cup \{s : \alpha\}, \Theta)\ \texttt{(where}\ \alpha\ \texttt{fresh)}$
$\quad \texttt{in}\ (\mathcal{C}, \mathcal{T}, U, L)$

$\texttt{INF}((\nu r)P, \Gamma, \Theta)\texttt{=}$
$\quad \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma \cup \{r^+ : \alpha_1, r^- : \alpha_2\}, \Theta)\ \texttt{(where}\ \alpha_1, \alpha_2\ \texttt{fresh)}$
$\quad \texttt{in}\ (\mathcal{C} \cup \{\alpha_1 = \overline{\alpha_2}\}, \mathcal{T}, U, L \setminus \{r^+, r^-\})$

$\texttt{INF}(P|Q, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L_1)\texttt{=}\ \texttt{INF}(P, \Gamma, \Theta)\ \texttt{in}$
$\quad \texttt{let}\ (\mathcal{C}_1, \mathcal{T}_1, U_1, L_2)\texttt{=INF}(Q, \Gamma, \Theta)\ \texttt{in}$
$\quad ((\mathcal{T} = \{\mathsf{end}\} \wedge U = \mathsf{end}) \vee (\mathcal{T}_1 = \{\mathsf{end}\} \wedge U_1 = \mathsf{end}))$
$\quad \texttt{if}\ \mathcal{T}\texttt{==}\{\mathsf{end}\}\ \texttt{then}\ (\mathcal{T}_2, U_2)\texttt{=}(\mathcal{T}_1, U_1)$
$\quad \texttt{else if}\ \mathcal{T}_1\texttt{==}\{\mathsf{end}\}\ \texttt{then}\ (\mathcal{T}_2, U_2)\texttt{=}(\mathcal{T}, U)$
$\quad \texttt{in}\ (\mathcal{C} \cup \mathcal{C}_1, \mathcal{T}_2, U_2, L_1 \uplus L_2)$

$\texttt{INF}(\Sigma_{i=1}^{n}(l_i).P_i, \Gamma, \Theta)\texttt{=}$
$\quad \texttt{let}\ (\mathcal{C}_1, \mathcal{T}_1, U_1, L_1)\texttt{=INF}(P_i, \Gamma, \Theta) \ldots\ldots (\mathcal{C}_n, \mathcal{T}_n, U_n, L_n)\texttt{=INF}(P_n, \Gamma, \Theta)$
$\quad \mathcal{C}'\texttt{=}\bigcup_i \{U_i = U_{i+1}\} \quad \forall\ i \in\ 1 \ldots n-1$
$\quad \texttt{in}\ (\mathcal{C}' \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n, \mathsf{split}(\&\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\}), U_1, L_1 \uplus \ldots \uplus L_n)$

$\texttt{INF}(\langle l\rangle.P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \Theta)$
$\quad \texttt{in}\ (\mathcal{C}, \oplus\{l : \mathcal{T}\}, U, L)$

$\texttt{INF}(r^p \triangleright P, \Gamma, \Theta)\texttt{=}\ \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, L)\texttt{=INF}(P, \Gamma, \emptyset)$
$\quad \texttt{in}\ (\mathcal{C} \cup \{T \leq \mathcal{T}\} \cup \{\Gamma(r^p) = [T]\}, \{U\}, \mathsf{end}, (L \uplus \{r^p\}))$

$\texttt{INF(rec}\ X.P, \Gamma, \Theta)\texttt{=}\ \texttt{if}\ X \in \mathsf{fpv}(P)\ \texttt{then}$
$\quad \texttt{let}\ (\mathcal{C}, \mathcal{T}, U, \emptyset)\texttt{=INF}(P, \Gamma, \Theta \cup \{X : \alpha_X; \alpha_X\})$
$\quad \texttt{in}\ (\mathcal{C}, \mu\alpha(\mathcal{T}), \mu\alpha(U), \emptyset)$

$\texttt{INF}(X, \Gamma, \Theta)\texttt{=}\ (\emptyset, \{(\texttt{fst}\ \Theta(X))\}, (\texttt{snd}\ \Theta(X)), \emptyset)$

Fig. 9. The algorithm to extract constraints in Ocaml-like syntax

It easy to see that each step preserves the solvability of the original constraints (remember that $\leq$ is a preorder). In particular, for the third one observe that $T \vee U$ is equal to $\overline{\overline{T} \wedge \overline{U}}$ by Proposition 1 and by the fact that syntactic duality is involutive i.e., $\overline{\overline{T}} = T$. Moreover, the fourth case keeps the constraint relative to a service definition (i.e. $\alpha \leq T_1$) since the name of a service can be communicated. Of course the algorithm will fail if the most general unifier does not exist.

Lemma 4. `solve` *terminates.*

Proof. Observe that, at each iteration, the algorithm decrements either the number of constraints or the number of constraints of the form $\alpha \leq T_1$ and $T_2 \leq \alpha$.

Next proposition states the soundness and the completeness of the `solve` w.r.t. the constraints generated by `INF`.

Proposition 7. *Let* $\mathcal{C}$ *s.t.* $(\mathcal{C}, \_, \_, \_) = \mathtt{INF}(P, \Gamma, \emptyset)$ *then* `solve` $\mathcal{C} =$ *success if and only is there ecists* σ *such that* $\sigma \vDash \mathcal{C}$.

Proof. The proof is by induction on P with case analysis on the last case applied by `INF` for the only-if part of the proof. The proof exploits the fact that pattern matching matches patterns in the given order and with a sets semantics. For the if part instead we proceed by induction on `solve` with case analysis on the last case applied in the pattern matching. The proof exploits session linearity in the fifth case to remove the three equations relative to a session or the fact that a service cannot be redefined (i.e. $x.P$ it is not allowed by the syntax) to remove the equation relative to the client invocation.

Example 5. The constraints generated for our running example are:

$$\alpha_a \leq \mu\alpha_{Y_2}.\&\{l : \oplus\{l_1 : \alpha_{Y_2}, l_2 : \alpha_{Y_2}\}\} \quad \alpha_a \leq \mu\alpha_{Y_1}.\&\{l : \oplus\{l_1 : \alpha_{Y_1}, l_3 : \alpha_{Y_1}\}\}$$

$$\overline{\mu\alpha_X. \oplus \{l : \&\{l_1 : \alpha_X, l_2 : \alpha_X, l_3 : \alpha_X, l_4 : \alpha_X\}\}} \leq \alpha_a$$

which reduce, after computing the meet, to verify: $\mu\alpha_X.\&\{l : \oplus\{l_1 : \alpha_X, l_2 : \alpha_X, l_3 : \alpha_X, l_4 : \alpha_X\}\} \leq \mu\alpha_{Y_1Y_2}.\&\{l : \oplus\{l_1 : \alpha_{Y_1Y_2}, l_2 : \alpha_{Y_1Y_2}, l_3 : \alpha_{Y_1Y_2}\}\}$.

5 Conclusion

As witnessed by the profusion of works about session types [18, 7, 6, 9, 10] (just to cite few), they provide us with an important paradigm to type communication protocols. In this paper we have proposed a solution to the problem of session type inference which has not been addressed explicitly in the previous literature. Our target calculus is a service-oriented calculus, with basic notion of dyadic sessions and recursive protocols, but no delegation.

First of all we introduced a typical non-syntax-directed type system for session types. Nondeterminism in the rule for typing choices, if-then-else and recursion is solved by means of another syntax directed type system. Two are the basic ideas behind this new syntax directed type system: using sets for multiplexing the if-then-else and using type variables to type process variables. The result is that the type associated to a session is a set of possibly not closed types. From this type system we extract the algorithm `INF` which returns a set of constraints. If all constraints in the set are satisfiable then a process is typable in the original non-syntax-directed type system. In the end, we propose the algorithm `SOLVE` which uses the set-operations [4] and subtyping relation [11] (on session types) to check the existence of a solution.

Due to space limitation, we leave a full discussion of related work on behavioural type systems for other service-oriented calculi (like [1, 14, 5]) to the full version of this paper (because they are not strictly related to the problem of session type inference). Here, it is just worth mentioning the interesting connection between contracts and session types spelled out in [15].

As a future research topic we want to apply similar results to solve type inference for the session calculus in [12]. Session delegation can already be handled just by modifying the `INF` algorithm so to add constraints of the form $\alpha \leq T.\alpha_1$, where α_1 is the delegated part of the session. On the other hand the `solve` algorithm is more subtle. We already developed a prototypical tool but the full discussion on the algorithm with proofs will appear in the author's forthcoming PhD thesis. The tool has been an important support for the development of the type system, allowing the early discovery of misconceptual errors.

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.*, pages 642–658. Springer Verlag, 2008.
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 *Lect. Notes in Comput. Sci.*, pages 38–57. Springer Verlag, 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 *Lect. Notes in Comput. Sci.*, pages 19–38. Springer Verlag, 2008.
4. R. Bruni and L. G. Mezzina. Meet recursive session types, 2008. Submitted manuscript, available at `http://www.di.unipi.it/~bruni/publications`.
5. R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST'08*, volume 5140 of *Lect. Notes in Comput. Sci.*, pages 100–115. Springer Verlag, 2008.
6. 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.
7. 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-Verlag, 2006.
8. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed: (functional pearl). In *ICFP'00*, pages 221–231. ACM, 2000.
9. P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. Bass: boxed ambients with safe sessions. In *PPDP'06*, pages 61–72. ACM, 2006.

10. S. Gay. Bounded polymorphism in session types. *Math. Struct. in Comput. Sci.*, 18(5):895–930, 2008.
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 *Lect. Notes in Comput. Sci.*, pages 122–138. Springer Verlag, 1998.
13. N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, 2005.
14. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
15. C. Laneve and L. Padovani. The pairing of contracts and session types. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.*, pages 681–700. Springer Verlag, 2008.
16. L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 216–231. Springer Verlag, 2008.
17. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385. IEEE Computer Society, 1993.
18. V. Vasconcelos, S. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

Appendix

Lemma 5 (Weakening). *If $\Gamma;\Theta \vdash P : T;U;L$ and $m \notin \mathsf{fn}(P)$ then $\Gamma, m : S;\Theta \vdash P : T;U;L$ for any S. If $\Gamma;\Theta \vdash P : T;U;L$ and $X \notin \mathsf{fpv}(P)$ then $\Gamma;\Theta, X : T';U' \vdash P : T;U;L$ for any T', U'.*

Lemma 6 (Strengthening). *If $\Gamma, m : S;\Theta \vdash P : T;U;L$ and $m \notin \mathsf{fn}(P)$ then $\Gamma;\Theta \vdash P : T;U;L$. If $\Gamma;\Theta, X : T';U' \vdash P : T;U;L$ and $X \notin \mathsf{fpv}(P)$ then $\Gamma;\Theta \vdash P : T;U;L$.*

Lemma 7 (Substitution lemma). *If $\Gamma, x : S;\Theta \vdash P : T;U;L$ and $\Gamma \vdash v : S$ then $\Gamma;\Theta \vdash P[^v/_x] : T;U;L$. If $\Gamma;\Theta, X : T';U' \vdash Q : T;U;L$ and $\Gamma;\Theta \vdash P : T';U';\emptyset$ then $\Gamma;\Theta \vdash Q[^P/_X] : T;U;L$.*

Proposition 8 (Subject Congruence). *If $\Gamma;\emptyset \vdash P : T;U;L$ and $P \equiv Q$ then $\Gamma;\emptyset \vdash Q : T;U;L$*

Theorem 6 (Subject reduction). *If $\Gamma;\emptyset \vdash P : T;U;L$ and $P \rightarrow P'$ then there exist Γ', L' s.t. $\Gamma';\emptyset \vdash P' : T;U;L'$*

Proof. The proof is by induction on the derivation with case analysis on the last typing rule used. We sketch the cases where the last applied $\rightarrow$-rules are (Inv) and (Lcom). For (Inv), we have that $\mathbb{D}[\![\overline{s}.P, s.Q]\!]$ is well-typed by hypothesis. Applying Lemma 1 twice on the context $\mathbb{D}$ with say $\Gamma \vdash s : [T'']$, it must be the case that the following instances of rules appeared in the type derivation:

$$\frac{\Gamma_1';\emptyset \vdash P : \overline{T''};U_1';L_1}{\Gamma_1';\emptyset \vdash \overline{s}.P : U_1';\mathtt{end};L_1} \qquad \frac{\Gamma_2';\emptyset \vdash Q : T'';\mathtt{end};\emptyset}{\Gamma_2';\emptyset \vdash s.Q : \mathtt{end};\mathtt{end};\emptyset}$$

To type the result obtained after the reduction (Inv), we can apply the Weakening Lemma (thanks to the side condition on r in the rule (Inv)) and Proposition 1.

$$\frac{\Gamma_1';\emptyset \vdash P : T'';U_1';L_1}{\Gamma_1', r^- : [\overline{T''}];\emptyset \vdash r^- \triangleright P : U_1';\mathtt{end};L_1 \uplus \{r^-\}}$$

$$\frac{\dfrac{\Gamma_2';\emptyset \vdash Q : T'';\mathtt{end};\emptyset}{\Gamma_2', r^+ : [T''];\emptyset \vdash r^+ \triangleright Q : \mathtt{end};\mathtt{end};\{r^+\}} \qquad \Gamma_2', r^+ : [T''];\emptyset \vdash s.Q : \mathtt{end};\mathtt{end};\emptyset}{\Gamma_2', r^+ : [T''];\emptyset \vdash s.Q | r^+ \triangleright Q : \mathtt{end};\mathtt{end};\{r^+\}}$$

The result follows re-applying Lemma 1 twice, in particular this is possible since the set of labels after the invocation still remains disjoint. $\square$

Finally we show that using the tighter notion of subtyping which requires syntactic equality of the sent/received values is the same as to use the full subtyping relation, if we consider closed systems. Let $\leq'$ the full subtyping relation introduced by Gay and Hole and let $\vdash^{GH}_{\mathsf{SD}}$ the type system reported in Figure 7 with rules (TdefSD) and (TinvSD) replaced with a couple of similar rules but for the use of $\leq'$ instead of the use of $\leq$. Additionally $\Gamma' \leq' \Gamma$ is defined if $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ and $\forall x \in \Gamma'(x) \leq' \Gamma(x)$ and $\forall m \in \Gamma'(m) \leq' \Gamma(m)$.

Theorem 7. *$\Gamma;\Theta \vdash_{\mathsf{SD}} P : \mathcal{T};U$ iff $\Gamma';\Theta \vdash^{GH}_{\mathsf{SD}} P : \mathcal{T};U$ with $\Gamma' \leq' \Gamma$.*