How to Infer Finite Session Types in a Calculus of Services and Sessions

Leonardo Gaetano Mezzina

IMT Lucca, Institute for Advanced Studies, Italy
leonardo.mezzina@imtlucca.it

Abstract. The notion of session is fundamental in service oriented applications, as it separates interactions between different instances of the same service, and it groups together basic units of work. Together with sessions, session types were introduced to track the type of the values exchanged in each session. In this paper we propose an algorithm to infer a restricted form of session types and we show that the problem is not directly related to the unification since we are in a context with duality in interactions. The discussion is based on a SCC-like [3] calculus adapted to fit session types. The calculus simplifies the discussion imposing strong syntactic constraints, but the ideas and the proposed algorithm can be adopted to study the type inference for other session oriented calculi. Also an OCaml prototype of the algorithm has been developed to show its feasibility.

1 Introduction

Sessions are used to structure interactions among parties resulting in a clearer and bug free way to write communicating programs. Session oriented calculi [13,14,20,12] were proposed to reason formally about communication patterns that encompass the simple *one-way* remote procedure call [7,8,9,2] but also allow for more sophisticated message exchanges.

Since the π-calculus is the *lingua franca* for expressing concurrent processes, we can translate sessions in π-calculus, representing them like a freshly created channel (a session channel) used by both the client and the particular service instance (created to serve the client) as an exchanging context.

However, from the type system point of view no (interesting) session channel is well typed under the simply typed π-calculus [17] which allows to transmit only a single type of message over each channel. Thus, session types were introduced to type session channels so as to describe both sequences (of input/output) and choices (internal/external) taking place on a session side.

The duality of session types also changes the way to consider the type inference problem which is no longer directly related to the unification as for the simply typed π-calculus. In fact, in the simply typed π-calculus we consider both input and output actions (which are dual) to reconstruct the channel sort, that is, sorting says what kind of values each channel can input and output. For example, the process $xc \mid \overline{x}5$ uses the channel x to input values of an unknown type (the

© Springer-Verlag Berlin Heidelberg 2008

same as c), say α_1, and to output an integer value. Here we can safely substitute α_1 with *int* and judge this channel of type $\mathtt{chan}(int)$; i.e. a channel used to exchange integer values. However, in a dual interaction we independently need the type of each side of the communication and, for example, the type of x become $?(int)$ (input of an integer) for the first side of the parallel and $!(int)$ (output of an integer) for the second side. This separation easily allows to judge that the interaction is safe, since each side performs the dual action with respect to the other side. It is worth noticing that the substitution $\{\alpha_1 \mapsto int\}$ still holds. The reasoning can be iterated if we want to capture types expressing sequence of inputs and outputs; e.g., $xc.\overline{x}c \mid \overline{x}5.xy$ then we have both types $?(int).!(int)$ and $!(int).?(int)$ in which we unified the type of the first value exchanged in the sequence and the type of the second value exchanged in the sequence. Similar considerations are made for the type inference algorithm described in [10].

Furthermore, session types extend basic sequences of actions adding both external and internal choices which can be considered as a set of offered options exposed by means of labels and as a selection among a set of options respectively. Unfortunately, the expressivity introduced by choices makes the type inference problem not directly related to the unification. First of all, the labels of each choice are unordered and also we would accept the comparison between an internal choice that offers more options and its external choice counterpart; that is, the "unification"could be possible if a part chooses only some of the options offered by the dual part. Given this, one may argue that the problem is similar to what is described in [19,18] for an object calculus and successfully solved by means of kinds. However, we think that the use of kinds for session types is not trivial since each session may offer multiple choices at different levels of nesting.

Instead, we tackle the problem by introducing another kind of constraints, indicated by $\bowtie$, between two dual session types. Moreover, an algorithm is proposed to solve this kind of constraints together with the simple equality (unifiable) equations.

To have a practical result, we apply the algorithm in the service oriented architecture scenario for reconstructing the type of each service. Thus, we model a system in which each service invocation creates a new session permitting both the exchanging of correlated messages and the isolation from different instances of the same service. As the possibility of different clients for a service, we assume persistent services always available for client requests.

The algorithm is built on top of a language with SCC-like [3] syntax since its syntactic constraints permits to focus our attention on at most two sessions usages each time (the current session and the parent session) whilst it maintains the expressivity to write interesting programs (such as factorial service) to test our results. Notwithstanding, the results can be adapted to any language, for example, our language is a particular instance of the system studied in [20], constraining the typing Δ to contain at most two session channels at the same time.

Outline of the paper. Section 2 fixes the syntax and the operational semantics of our session calculus. Section 3 shows the classic nondeterministic typing rules. Section 4 presents the type inference algorithm subdivided in two parts:

$$
\begin{array}{lcll}
P, Q & ::= & \mathbf{0} & \text{(nil)} \\
& \mid & s.P & \text{(service definition)} \\
& \mid & \overline{v}.Q & \text{(invocation)} \\
& \mid & \texttt{if } v = v_1 \texttt{ then } P \texttt{ else } Q & \text{(if-then-else)} \\
& \mid & (\tilde{x}).P & \text{(tuple input)} \\
& \mid & \langle \tilde{v} \rangle.P & \text{(value output)} \\
& \mid & \Sigma_{i=1}^{n}(l_i).P_i & \text{(label guarded sum)} \\
& \mid & \langle l \rangle.P & \text{(label choice)} \\
& \mid & \texttt{return } \tilde{v}.P & \text{(value return)} \\
& \mid & P|Q & \text{(parallel)} \\
& \mid & (\nu s)P & \text{(service restriction)} \\
\\
v & ::= & \mathsf{f}(\tilde{v}) & \text{(external function call)} \\
& \mid & x & \text{(variable)} \\
& \mid & s & \text{(service)} \\
& \mid & \ldots, -1, 0, 1, \ldots & \text{(integer)}
\end{array}
$$

Fig. 1. Syntax of our service calculus

a constraints extractor and a solving algorithm. We have also implemented all the algorithms described [15] and Section 5 shows some examples of usage of our tool.

2 A Session Oriented Calculus

Our processes are generated by the abstract syntax in Figure 1, where the meta-variable x ranges over variables, s over service names and l over labels. Values can be either a variable, a service, an integer or the result of an *external function* call f.

As usual $\mathbf{0}$ identifies the inaction process (omitted in tail position), $|$ is the parallel composition of two processes and (νs) is the restriction of s. Service definition $s.P$ and service invocation $\overline{v}.Q$ are used to instantiate a new session, i.e., a way to put in direct connection a service instance P with the body Q of the invocation client. For each service invocation a fresh instance of the body is generated to serve the client, in this manner the service is ready for another client invocation. Once the service side and the client side are connected by means of a session, both parties can communicate via dual operators. This means that if one side performs an input $(\tilde{x}).P$, the other side can send a value tuple with $\langle \tilde{v} \rangle.P$ and if one side offers a choice $\Sigma_{i=1}^{n}(l_i).P_i$, the other side can select a label with $\langle l \rangle.P$.

To logically connect client and service instance, we use a special session construct $r \triangleright P$ and $r \triangleright Q$ which says that both, the service instance P and the client invocation body Q agree on the private name r and they will use it as communication context. Sessions can be arbitrarily nested and the operator $\texttt{return}\ \tilde{v}.P$ is used to output a value upward the parent session.

Binders are $(\nu s)P$ for s in P and $(\tilde{x}).P$ for $\tilde{x}$ in P; the former is the binder for service names and the latter is the binder for variables. As usual processes are considered up to alpha equivalence and the set of free names is defined in the standard way. Moreover, the operation of substitution $P[{}^{\tilde{v}}/_{\tilde{x}}]$ is the standard capture avoiding substitution of variables with values.

Differently from [4], we formalize the operational semantics of the calculus by a one-step reduction relation $\rightarrow$ up to the standard structural congruence $\equiv$ plus the rule $\mathbf{r} \triangleright (\nu \mathbf{m})\mathbf{P} \equiv (\nu \mathbf{m})(\mathbf{r} \triangleright \mathbf{P})$ if $\mathbf{r} \neq \mathbf{m}$, for sessions handling, where m range over both session and service names.

$$\begin{array}{lll}
(\mathit{Inv}) & \mathbb{D}[\![\mathbb{C}[\![\overline{s}.P]\!] \mid \mathbb{C}_1[\![s.Q]\!]]\!] \;\rightarrow\; \mathbb{D}[\![(\nu r)\mathbb{C}[\![r \triangleright P | r \triangleright Q]\!] \mid \mathbb{C}_1[\![s.Q]\!]]\!] & r \notin \mathsf{fn}(\mathbb{C}[\![\overline{s}.P]\!]|Q) \\
(\mathit{Com}) & \mathbb{C}[\![r \triangleright (\tilde{x}).P | r \triangleright \langle \tilde{v} \rangle.Q]\!] \;\rightarrow\; \mathbb{C}[\![r \triangleright P[{}^{\tilde{v}}/_{\tilde{x}}] | r \triangleright Q]\!] & \\
(\mathit{Lcom}) & \mathbb{C}[\![r \triangleright \Sigma_{i=1}^{n}(l_i).P_i | r \triangleright \langle l_k \rangle.Q]\!] \;\rightarrow\; \mathbb{C}[\![r \triangleright P_k | r \triangleright Q]\!] & (1 \leq k \leq n) \\
(\mathit{Ret}) & \mathbb{C}[\![r \triangleright (\tilde{x}).P | r \triangleright (r_1 \triangleright \mathtt{return}\ \tilde{v}.Q|Q')]\!] \;\rightarrow\; \mathbb{C}[\![r \triangleright P[{}^{\tilde{v}}/_{\tilde{x}}] \mid r \triangleright (r_1 \triangleright Q|Q')]\!] & \\
(\mathit{IfT}) & \mathbb{C}[\![\mathtt{if}\ v = v_1\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \rightarrow \mathbb{C}[\![P]\!] & (v = v_1) \downarrow \mathtt{true} \\
(\mathit{IfF}) & \mathbb{C}[\![\mathtt{if}\ v = v_1\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \rightarrow \mathbb{C}[\![Q]\!] & (v = v_1) \downarrow \mathtt{false} \\
(\mathit{Scop}) & P \;\rightarrow\; P' \;\Rightarrow\; (\nu m)P \rightarrow (\nu m)P' & \\
(\mathit{Str}) & P \equiv P' \;\; P' \rightarrow Q' \;\; Q' \equiv Q \Rightarrow \; P \rightarrow Q & \\
 & \text{where } \mathbb{C}, \mathbb{D} \;::=\; [\![\cdot]\!] \mid \mathbb{C}|P \mid r \triangleright \mathbb{C} &
\end{array}$$

Priority of the operators in order of increasing relevance is: $|$, $\triangleright$ and ν so, for example $r \triangleright P|Q$ means $(r \triangleright P)|Q$ and $(\nu r)P|Q$ means $((\nu r)P)|Q$.

Rule (Inv) shows how the invocation of a service creates a new session that puts in direct communication an instance of the service with the client body; now the two processes are able to communicate.

The rules (Com), (Lcom) show respectively how a tuple is transmitted between the two sides of a session and how the process Q can choose one of the options offered by P. Rule (Ret) illustrates how a nested session r_1 can output a value, upward the parent session, which is read by P in the dual side of r. Both the rules (Com) and (Ret) manage similar communication patterns to what is defined in [5] which describes a variant of *Mobile Ambients* calculus [6].

As an example consider the following `calc` service

$$\mathtt{calc}.(\mathbf{sum}).(\mathsf{x},\mathsf{y}).\langle \mathsf{add}(\mathsf{x},\mathsf{y}) \rangle \;+\; (\mathbf{inc}).(\mathsf{x}).\langle \mathsf{add}(\mathsf{x},1) \rangle$$

which offers two options. Option **sum** reads (x, y) from the client and replies with the result of the external function call add. The add function is only available on one session side, directly implemented in some programming language (i.e., $\mathsf{add} = \lambda(x, y).x{+}y$). Option **inc** only inputs a value and then emits the result. One client that successful interacts with the service is:

$$\overline{\mathtt{calc}}.\langle \mathbf{sum} \rangle.(1,1).(\mathit{res}).\mathtt{return}\ \mathit{res}$$

After rules (Inv) and (Lcom) are applied, the parallel of the two processes above become:

$$r \triangleright (\mathsf{x},\mathsf{y}).\langle \mathsf{add}(\mathsf{x},\mathsf{y}) \rangle \mid r \triangleright (1,1).(\mathit{res}).\mathtt{return}\ \mathit{res} \rightarrow$$
$$r \triangleright \langle \mathsf{add}(1,1) \rangle \mid r \triangleright (\mathit{res}).\mathtt{return}\ \mathit{res} \rightarrow r \triangleright \mathbf{0} \mid r \triangleright \mathtt{return}\ 2$$

At the end of interaction, the client has the result in the *res* variable which is returned to the parent session.

$$\begin{array}{ll}
\mathsf{wf}(\mathbf{0}, X) & = X \\
\mathsf{wf}(s.P, X) & = \mathsf{wf}(P, X \setminus s) \\
\mathsf{wf}(\overline{v}.P, X) & = \mathsf{wf}(P, X) \\
\mathsf{wf}(\texttt{if } v = v_1 \texttt{ then } P \texttt{ else } Q, X) & = \mathsf{wf}(Q, \mathsf{wf}(P, X)) \\
\mathsf{wf}((\tilde{x}).P, X) \wedge \mathsf{wf}(\langle\tilde{v}\rangle.P, X) & = \mathsf{wf}(P, X) \\
\mathsf{wf}(\Sigma_{i=1}^{n}(l_i).P_i, X) & = \mathsf{wf}(P_n, \mathsf{wf}(P_{n-1}, \ldots \mathsf{wf}(P_1, X)) \ \forall i, j.l_i \neq l_j \text{ if } i \neq j \\
\mathsf{wf}(\langle l\rangle.P, X) \wedge \mathsf{wf}(\texttt{return } \tilde{v}.P, X) & = \mathsf{wf}(P, X) \\
\mathsf{wf}(P|Q, X) & = \mathsf{wf}(Q, \mathsf{wf}(P, X)) \\
\mathsf{wf}((\nu s)P, X) & = \mathsf{wf}(P, X \cup \{s\})
\end{array}$$

Fig. 2. Definition of wf

3 Typing

3.1 Well Formedness

In this sub-section we discuss the notion of well-formed process. Since we are in a context with duality each service restriction (νs) authorizes to use in its scope both s as service declaration and $\overline{s}$ as service invocation. However, syntax does not constrain programmers to insert a service declaration in the scope of a restriction, and it can happen that a process has a service invocation without the corresponding declaration. Thus, we require our processes to have at least the service declaration for each service restriction. This requirement, besides to be reasonable, is also crux to successfully solve the constraints generated with the type inference algorithm (see Proposition 1).

The formal definition of well formedness is built from the function wf (Figure 2) which takes a process with all bound and free names different, the set of service names that should be declared and returns the set of names not still declared.

Definition 1 (Well formedness). *A process P is well formed if* $\mathsf{wf}(P, \emptyset) = \emptyset$

The definition ensures the process P declares every service annunciated by means of restrictions (no matters where!). Moreover, all the labels of a choice must be different.

From now on, all the processes we are going to handle are implicitly assumed to satisfy Definition 1.

3.2 Typing Rules

The set of types is defined by the abstract syntax in Figure 3. Session types (ranged over by T,U) express sequences of typed tuples of input and output. Intuitively, types capture the actions performed in a side of a session; $?(S_1, \ldots, S_n).T$ expresses the fact that a process performs an input within a session and then behaves like T. Similar holds for $!(S_1, \ldots, S_n).T$ in which an output action is

$$
\begin{array}{rcll}
T, U & ::= & \texttt{end} & \text{(no action)} \\
 & \mid & ?(S_1, \ldots, S_n).T & \text{(input of a tuple)} \\
 & \mid & !(S_1, \ldots, S_n).T & \text{(output of a tuple)} \\
 & \mid & \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(external choice)} \\
 & \mid & \oplus\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(internal choice)} \\
S & ::= & \mathit{int} & \text{(basic integers type)} \\
 & \mid & [T] & \text{(session type)}
\end{array}
$$

Fig. 3. Syntax of types

performed, instead. The type of an external choice is a list of (offered) labels with the corresponding subprocess usage. Also, the type of an internal choice contains a list, because multiple choices may be performed at the same time.

Sorts S can be either the type of a service $[T]$ or an integer.

Our set of type judgments is in Figure 4. Type judgments for values take the form $\Gamma \vdash v : S$ where the *type environment* Γ is a finite partial mapping from variables, services and external function names to sorts and function types. When $x \notin \mathit{dom}(\Gamma)$ (same holds for $s \notin \mathit{dom}(\Gamma)$) we write $\Gamma, x : S$ for the type environment obtained by extending Γ with the binding of x to S. First four rules for values are standard and the signature of each used external function must be inserted in the environment as functional type (rule (FuncV)) because they are not bound by the process.

Type judgments for processes take the form $\Gamma \vdash P : T; U$ where T is the type of the current session, while the type U represents outputs of P towards the parent session. The type of $\mathbf{0}$ in (Tzero) is $\texttt{end}; \texttt{end}$ since no action is 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 environment. Rule (Tdef) constraints the protocol of the service to be the same as the body type of the process P and no return is allowed toward the parent session. This condition is necessary, because we want that the service body does not interfere within the client's context. (Tinv) checks the service behaves in the dual manner with respect to the current client. Here, the dual of T, written $\overline{T}$ is inductively defined as:

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

Rules (Tin), (Tout) and (Tret) insert the usage type in the correct place. Rule (Tbranch) considers any subset of the branches while rule (Tchoice) can arbitrarily add some branches. The shape of the rule (Tchoice) is necessary since, the if-then-else construct allows choosing between many branches at the same time and also different clients can invoke the same service making their own choices. When we have multiple paths, returns to the parent session must have the same type U. The nondeterminism in the rules (Tbranch) and (Tchoice) is typical for

$$\overset{(\textsc{Ser})}{\Gamma, s : S \vdash s : S} \qquad \overset{(\textsc{Var})}{\Gamma, x : S \vdash x : S} \qquad \overset{(\textsc{IntV})}{\Gamma \vdash n : int} \qquad \overset{(\textsc{FuncV})}{\frac{\Gamma \vdash v_1 : S_1 \ldots \Gamma \vdash v_n : S_n}{\Gamma, \mathsf{f} : S_1 \times \ldots \times S_n \to S' \vdash \mathsf{f}(v_1, \ldots, v_n) : S'}}$$

$$\overset{(\textsc{Tzero})}{\Gamma \vdash \mathbf{0} : \mathtt{end}; \mathtt{end}} \qquad \overset{(\textsc{Tnew})}{\frac{\Gamma, s : S \vdash P : T; U}{\Gamma \vdash (\nu s)P : T; U}} \qquad \overset{(\textsc{Tif})}{\frac{\Gamma \vdash P : T; U \quad \Gamma \vdash Q : T; U}{\Gamma \vdash \mathtt{if}\ v = v_1\ \mathtt{then}\ P\ \mathtt{else}\ Q : T; U}}$$

$$\overset{(\textsc{Tdef})}{\frac{\Gamma \vdash P : T; \mathtt{end} \quad \Gamma \vdash s : [T]}{\Gamma \vdash s.P : \mathtt{end}; \mathtt{end}}} \qquad \overset{(\textsc{Tinv})}{\frac{\Gamma \vdash P : T; U \quad \Gamma \vdash v : [T'] \quad \overline{T} = T'}{\Gamma \vdash \overline{v}.P : U; \mathtt{end}}}$$

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

$$\overset{(\textsc{Tbranch})}{\frac{I \subseteq \{1, \ldots, n\} \quad \forall i \in \{1, \ldots, n\} \quad \Gamma \vdash P_i : T_i; U}{\Gamma \vdash \Sigma_{i=0}^{n} (l_i).P_i : \&\{l_j : T_j\}_{j \in I}; U}}$$

$$\overset{(\textsc{Tchoice})}{\frac{l = l_i \in \{l_1, \ldots, l_n\} \quad \Gamma \vdash P : T_i; U}{\Gamma \vdash \langle l \rangle.P : \oplus\{l_1 : T_1, \ldots, l_n : T_n\}; U}}$$

$$\overset{(\textsc{TparL})}{\frac{\Gamma \vdash P : T; \mathtt{end} \quad \Gamma \vdash Q : \mathtt{end}; \mathtt{end}}{\Gamma \vdash P | Q : T; \mathtt{end}}} \qquad \overset{(\textsc{TparR})}{\frac{\Gamma \vdash P : \mathtt{end}; \mathtt{end} \quad \Gamma \vdash Q : T; \mathtt{end}}{\Gamma \vdash P | Q : T; \mathtt{end}}}$$

Fig. 4. Typing rules

$$\overset{(\textsc{TbranchSD})}{\frac{\forall i \in \{1, \ldots, n\} \quad \Gamma \vdash_{\mathsf{SD}} P_i : T_i; U}{\Gamma \vdash_{\mathsf{SD}} \Sigma_{i=0}^{n} (l_i).P_i : \&\{l_i : T_i\}_{i \in \{1, \ldots, n\}}; U}}$$

$$\overset{(\textsc{TchoiceSD})}{\frac{\Gamma \vdash_{\mathsf{SD}} P : T; U}{\Gamma \vdash_{\mathsf{SD}} \langle l \rangle.P : \oplus\{l : T\}; U}} \qquad \overset{(\textsc{TinvSD})}{\frac{\Gamma \vdash_{\mathsf{SD}} P : T; U \quad \Gamma \vdash_{\mathsf{SD}} v : [T'] \quad T \bowtie T'}{\Gamma \vdash_{\mathsf{SD}} \overline{v}.P : U; \mathtt{end}}}$$

$$\overset{(\textsc{TifSD})}{\frac{\Gamma \vdash_{\mathsf{SD}} P : T; U \quad \Gamma \vdash_{\mathsf{SD}} Q : T'; U \quad T'' = \mathsf{merge}(T, T')}{\Gamma \vdash_{\mathsf{SD}} \mathtt{if}\ v = v_1\ \mathtt{then}\ P\ \mathtt{else}\ Q : T''; U}}$$

Fig. 5. Syntax directed typing rules

session type systems, and it is actually useful in subject reduction proofs (see [4] for the subject reduction proof of the current framework).

The two rules for parallel composition (TparL) and (TparR) allow parallel composition of two processes only if at least one does not make any action in both the current session and the parent session, i.e., it has type $\mathtt{end}; \mathtt{end}$.

Now we show an example of typing,

Example 1. Take the calculator example. The type $!(\mathit{int},\mathit{int}).?(\mathit{int});!(\mathit{int})$ expresses the client usage after the sum choice: the output of two integers is followed by the reading of the result and an integer is returned outside the session (the type after semicolon always stands for a return action, that is, an output out of the current session). Previous in-session usage is compared with the dual session usage $?(\mathit{int},\mathit{int}).!(\mathit{int})$ to ensure the soundness of the invocation. Below we report the typing proof, where we let $\Gamma = \mathit{calc} : [\&\{\mathsf{sum} :?(\mathit{int},\mathit{int}).!(\mathit{int})\}]$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{return}\ x : \texttt{end};!(\mathit{int})}{\Gamma \vdash (x).\texttt{return}\ x : ?(\mathit{int});!(\mathit{int})}\,(\text{Tin})}{\Gamma \vdash \langle 1,1\rangle.(x).\texttt{return}\ x : !(\mathit{int},\mathit{int}).?(\mathit{int});!(\mathit{int})}\,(\text{Tout})}{\Gamma \vdash \langle \mathsf{sum}\rangle.\langle 1,1\rangle.(x).\texttt{return}\ x : \oplus\{\mathsf{sum} :!(\mathit{int},\mathit{int}).?(\mathit{int})\};!(\mathit{int})}\,(\text{Tchoice})}{\Gamma \vdash \overline{\texttt{calc}}.\langle \mathsf{sum}\rangle.\langle 1,1\rangle.(x).\texttt{return}\ x : !(\mathit{int});\texttt{end}}\,(\text{Tinv})$$

It is worth noticing that we are authorized to apply the rule (Tinv) because $\Gamma(\texttt{calc}) = [\overline{\oplus : \{\mathsf{sum} :!(\mathit{int},\mathit{int}).?(\mathit{int})}]$. Thus, the assumption about $\texttt{calc}$ ignores the option labeled with inc since it is useless for this particular client.

The main problem we are going to face in the algorithmic type inference is due to the nondeterministic nature of the typing rules for choices. Relatively to the previous example, it is not strictly necessary to discard the inc branch when inserting the type of the $\texttt{calc}$ service in the environment (rule (Tbranch)). In fact, rule (Tchoice) would allow to correctly typecheck the client even if the inc branch were not specified. Consequently, a client can arbitrarily discard unused branches allowing to correctly typecheck other clients with different choices.

4 Type Inference

4.1 Syntax Directed Rules

Before introducing an algorithm for the type inference we need to solve the nondeterminism of the type system (due to both rules (Tchoice) and (Tbranch)) replacing it with another set of syntax directed rules, shown in Figure 5 (only different rules are reported). Next, we are able to show that the two set of rules coincide so that we can use the syntax directed rules to formulate our algorithm.

The previous type system permits to arbitrarily add or remove the branches of a choice until the rule (Tinv) holds. We factorize out the nondeterminism building the type with all the currently available information, which is equivalent to take all the branches in rule (TbranchSD) and only one branch in rule (TchoiceSD). Also, the new rule (TifSD) needs a way to deterministically get the correct type, and it uses the support function merge defined in Figure 6. In other words, merge works as follow: if both P and Q are internal choices we create a new type with those branches that are not within the intersection of the two sets of labels plus the merge of those branches that are within the intersection. In fact, a compliant external choice should account for all the possible

$$
\begin{array}{ll}
\mathsf{merge}(\mathsf{end},\mathsf{end}) & = \mathsf{end} \\
\mathsf{merge}(!(\tilde{S}).T, !(\tilde{S}).T') & = !(\tilde{S}).\mathsf{merge}(T,T') \\
\mathsf{merge}(?(\tilde{S}).T, ?(\tilde{S}).T') & = ?(\tilde{S}).\mathsf{merge}(T,T') \\
\mathsf{merge}(\oplus\{l_1 : T_1, \ldots, l_n : T_n\}, \oplus\{l'_1 : T'_1, \ldots, l'_m : T'_m\}) & = \oplus\{_{\forall i\, \exists j\, l_i = l'_j} l_i : \mathsf{merge}(T_i, T'_j)\ , \\
 & \quad {}_{\forall i,j\, l_i \neq l'_j} l_i : T_i\ ,\ {}_{\forall j,i\, l'_j \neq l_i} l'_j : T'_j\} \\
\mathsf{merge}(\&\{l_1 : T_1, \ldots, l_n : T_n\}, \&\{l'_1 : T'_1, \ldots, l'_m : T'_m\}) & = \&\{_{\forall i\, \exists j\, l_i = l'_j} l_i : \mathsf{merge}(T_i, T'_j)\}
\end{array}
$$

Fig. 6. Merge for the if branches

$$
\begin{array}{ll}
\mathsf{end} \bowtie \mathsf{end} & \\
\oplus\{l_1 : T_1, \ldots, l_n : T_n\} \bowtie \&\{l'_1 : T'_1, \ldots, l'_m : T'_m\} & = \forall i,j\ l_i = l'_j\ \rightarrow T_i \bowtie T'_j \quad \wedge \\
 & \quad \{l_1, \ldots, l_n\} \subseteq \{l'_1, \ldots, l'_m\} \\
?(\tilde{S}).T \bowtie !(\tilde{S}).T' & = T \bowtie T'
\end{array}
$$

Fig. 7. Services Join

options the process could select during its evaluation. Instead, if we are merging two branches of an external choice we are able only to guarantee options that are within the intersection of the set of labels and additionally these branches must be mergeable.

The problem is that, at this point, the standard syntactic equivalence is not useful for the comparison of two types since the branches in internal choices are a subset of the corresponding branches in external choices. The $\bowtie$ relation reported in Figure 7, combined with the symmetric cases, solves the above problem and is used by (InvSD) to validate the client protocol (it is just a restricted form of subtyping, written as a symmetric operator).

The next lemma shows that a typable process in $\vdash$ is also typable in $\vdash_{\mathsf{SD}}$ and vice versa. In this manner, we can build our type inference algorithm on top of the syntax directed rules throwing out nondeterminism.

Lemma 1. *If $\Gamma \vdash P : T; U$ then there exist Γ' and T' s.t. $\Gamma' \vdash_{\mathsf{SD}} P : T'; U$. Conversely, if $\Gamma \vdash_{\mathsf{SD}} P : T; U$ is derivable, so is, $\Gamma \vdash P : T; U$.*

Proof. Straightforward induction on derivations of $\Gamma \vdash P : T; U$ and $\Gamma \vdash_{\mathsf{SD}} P : T; U$ □

4.2 Tree Unification

The inference algorithm relies on a unification algorithm unify among trees as the one described in [16]. Nevertheless, in order to use this algorithm we need to clarify how to build trees starting from our types. We first introduce the standard set of type variables V, and a set of constants $K = \{\mathsf{end}, int\}$. The meta variable α ranges over the elements of V. A production for type variables is also added to the syntax of sorts in Figure 3. A tree type is a partial function $\mathcal{T}$ from the set of finite strings over the alphabet of positive integers (describing paths in the tree), to a ranked alphabet $L = \{[\quad], \,.\, \} \cup V \cup K \cup \{?^i, !^i, \&^i, \oplus^i | i > 0\}$ where the rank of $V \cup K$ is 0, the rank of $[\quad]$ is 1 and the rank of $\{?^i, !^i, \&^i, \oplus^i | i > 0\}$ is i. For example, if $\mathcal{T}(\pi) = [\quad]$ then $\mathcal{T}(\pi \cdot 1)$ is defined, which means, if in the tree following the path as specified by the string π we find a $[\quad]$ then we can use the string $\pi \cdot 1$ to retrieve the service type.

Definition 2 (treeof)**.** *The function* treeof *translates types into trees and is inductively defined as:*

$$\begin{array}{ll}
\mathsf{treeof}(?(S_1,\ldots,S_n).T) & = ?^n(\mathsf{treeof}(S_1),\ldots,\mathsf{treeof}(S_n)).\mathsf{treeof}(T)\\
\mathsf{treeof}(!(S_1,\ldots,S_n).T) & = !^n(\mathsf{treeof}(S_1),\ldots,\mathsf{treeof}(S_n)).\mathsf{treeof}(T)\\
\mathsf{treeof}(\&\{l_1 : T_1,\ldots,l_n : T_n\}) & = \&^n\{l'_1 : \mathsf{treeof}(T_1),\ldots,l'_n : \mathsf{treeof}(T_n)\}\\
\mathsf{treeof}(\oplus\{l_1 : T_1,\ldots,l_n : T_n\}) & = \oplus^n\{l'_1 : \mathsf{treeof}(T_1),\ldots,l'_n : \mathsf{treeof}(T_n)\}\\
\mathsf{treeof}(K) & = K\\
\mathsf{treeof}(\alpha) & = \alpha\\
 & \qquad \textit{where } (l'_1,\ldots,l'_n) \textit{ is an ordering of } (l_1,\ldots,l_n)
\end{array}$$

Trees follow the same structure as types but we need arity annotations and a fixed ordering among the labels of each choice.

The substitution returned by unify is a mapping $p : V \to \mathcal{T}$. Given a substitution p and a tree $\mathcal{T}$, we obtain the tree $p\mathcal{T}$ as the result of the simultaneous substitution of the tree $p\alpha$ for each occurrence of variable α in $\mathcal{T}$. Standard substitution composition is written as $p \cdot p'$ if p and p' are two substitutions. Another subtle aspect is that valid substitutions returned by the unify must be acyclic (this can be verified e.g., by using the so-called *occur-check*), because (for simplicity) the current type system does not handle regular recursive types. Recursive types would permit to typecheck process like $(\nu a)(a.(x).\overline{x}.\langle x\rangle | \overline{a}.\langle a\rangle)$ and could be handled by allowing a solution for cyclic substitutions. Hereafter, we will use types and trees interchangeably, since they are isomorphic.

4.3 An Algorithm to Extract Constraints

The type inference is subdivided in two parts: the constraints extraction part and the solving part. For the first part, the algorithm `INF`, depicted in Figure 8, takes a process P and an environment Γ. Γ contains an entry for each service and variable in P corresponding to either a type variable (meaning that we rely on the algorithm to find out the type of a name) or a sort (if we simply want typecheck). Moreover, Γ must contain the functional type of each external function used as a value; environment Γ *restricted with the set of free names of* P is denoted $\Gamma_{\downarrow \mathsf{fn}(P)}$. The algorithm returns a triple: a set $\mathcal{C}$ of constraints, the type T of actions in the current session and the type U of outputs upwards the parent session. The set $\mathcal{C}$ of constraints contains equations of the form $T = T'$ and $T \bowtie T'$.

Basically, `INF` is extracted by reading the syntax directed rules (Figure 5) in a bottom-up manner and generating an equality constraint when the rule requires two types to be equal: e.g., in the `if` case of the algorithm we add an equation that requires equality for the returned type of both P and Q since the rule

(TifSD) requires two U's. It is worth noticing that, with an abuse of notation, we use merge for indicating a slightly different function from that defined in Figure 6, the new one behaves like the original function but also returns an equality constraint in each case the previous function requires syntactical equality. It is not expressively annotated when the algorithm fails but it should be clear that an error is generated each time the code does not match the algorithm expectation. E.g., a subtle case is in the invocation when we directly read from Γ the type of the value; we implicitly expect the value to be either a variable or a service (neither a function nor an integer) bound in the environment.

Next theorem is fundamental for the soundness of our results and it shows that if we have a substitution that solves the constraints set generated with $\mathtt{INF}(P, \Gamma)$ then such a substitution applied to Γ yields a correct typing for each service and variable in P.

Theorem 1. *Let* $\mathtt{INF}(Q, \Gamma) = (\mathcal{C}_1, T_1, U_1)$*,* $Q \equiv (\nu \tilde{s})P$ *and* p *a substitution for each type variable in* $\mathcal{C}_1$ *to a concrete type (a type without type variables). Then,* $\vDash p\mathcal{C}_1$ *holds if and only if* $(p\Gamma)_{\downarrow \mathsf{fn}(P)} \vdash_{\mathsf{SD}} P : pT_1; pU_1$*.*

Proof. The assumption on p is required for throwing out some valid solutions and recover the soundness with respect to the syntax directed typing rules; we reserve the problem of *principal typing* for further investigations. $\Rightarrow$ By induction on the first applied rule in the algorithm, we sketch some cases. If $\mathtt{INF}(s'.P', \Gamma)$ the respective case is applied. By inductive hypothesis $\vDash p\mathcal{C}$ holds and $(p\Gamma)_{\downarrow \mathsf{fn}(P')} \vdash_{\mathsf{SD}} P' \colon pT; \mathsf{end}$. Also if $\vDash p(\mathcal{C} \cup \{\Gamma(s) = [T]\})$ holds we can instantiate the premises of the rule (TdefSD) to obtain the typing for $(p\Gamma)_{\downarrow \mathsf{fn}(P')}, s \colon pT \vdash_{\mathsf{SD}} s.P' \colon \mathsf{end}; \mathsf{end}$. In the case of rules that introduce binders $p\Gamma$ is used to get the correct type. For example, if $\mathtt{INF}((\nu s)P', \Gamma)$ then $(p\Gamma)_{\downarrow \mathsf{fn}(P') \setminus s}, s \colon p\Gamma(s) \vdash_{\mathsf{SD}} P' \colon pT; pU$ and consequently $(p\Gamma)_{\downarrow \mathsf{fn}((\nu s)P')} \vdash_{\mathsf{SD}} (\nu s)P' \colon pT; pU$. $\Leftarrow$ By induction on the last applied rule in the type system. For example if the last applied rule was (TinvSD) we have that $p(\Gamma_{\downarrow \mathsf{fn}(P')})(v) = [T']$ and that $T' \bowtie pT''$ holds in the premises, where pT'' is the typing of P' in $\overline{v}.P'$. Since by inductive hypothesis $\vDash p\mathcal{C}$ holds then $\vDash p(\mathcal{C} \cup \{T' \bowtie T''\})$ holds too. $\square$

4.4 How to Solve the Constraints Set

At the end of the previous sub-section we show the fundamental role played by the substitution p, solution of the constraints set; next we show how to algorithmically get such a solution.

The algorithm in Figure 9, in OCaml like syntax, is used to find the solution of the constraints set $\mathcal{C}$ (which is treated like an ordered list of constraints). If the equation is a simple equality, it can be directly solved by unify, which returns a substitution applied to both the environment and the tail of the constraints list. If the constraint is a $\bowtie$ equation we are comparing two dual sides of a session and we cannot directly unify. In fact, as discussed in the introduction, the information which can be unified is only that information on the types of the trasmitted/received tuples since they should be the same for both sides.

```
VALUEINF($x, \Gamma$)= $(\emptyset, \Gamma(x))$
VALUEINF($s, \Gamma$)= $(\emptyset, \Gamma(s))$
VALUEINF($n, \Gamma$)= $(\emptyset, int)$
VALUEINF(f($v_1, \ldots, v_n$), $\Gamma$)=
          let ($\mathcal{C}_1, S_1$)=VALUEINF($v_1, \Gamma$)......($\mathcal{C}_n, S_n$)=VALUEINF($v_n, \Gamma$)
              $\Gamma(\mathsf{f}) = S_1 \times \ldots \times S_n \rightarrow S$
          in  ($\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n, S$)
INF($s.P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
              $U = \mathsf{end}$
              $\mathcal{C}_1$= $\mathcal{C} \cup \{\Gamma(s) = [T]\}$
          in ($\mathcal{C}_1$,end,end)
INF($\overline{v}.P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
              $\mathcal{C}_1$= $\mathcal{C} \cup \{\Gamma(v) \bowtie [T]\}$
          in ($\mathcal{C}_1, U$,end)
INF($(x_1, \ldots, x_n).P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
          in ($\mathcal{C}$,$?(\Gamma(x_1), \ldots, \Gamma(x_n)).T, U$)
INF($\langle v_1, \ldots, v_n \rangle.P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
          ($\mathcal{C}_1, S_1$)=VALUEINF($v_1, \Gamma$)......($\mathcal{C}_n, S_n$)=VALUEINF($v_n, \Gamma$)
          in ($\mathcal{C} \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$,$!(S_1, \ldots, S_n).T, U$)
INF(return $v_1, \ldots, v_n.P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
          ($\mathcal{C}_1, S_1$)=VALUEINF($v_1, \Gamma$)......($\mathcal{C}_n, S_n$)=VALUEINF($v_n, \Gamma$)
          in ($\mathcal{C} \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n, T$,$!(S_1, \ldots, S_n).U$)
INF(if $v = v_1$ then $P$ else $Q, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
              ($\mathcal{C}_1, T_1, U_1$)=INF($Q, \Gamma$)
              ($\mathcal{C}_2, T_2$)=merge($T, T_1$)
               $\mathcal{C}_2 = \mathcal{C}_2 \cup \{U = U_1\}$
          in ($\mathcal{C} \cup \mathcal{C}_1 \cup \mathcal{C}_2, T_2, U$)
INF($(\nu s)P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
          in ($\mathcal{C}, T, U$)
INF($P|Q, \Gamma$)=
          let ($\mathcal{C}, T$,end)=INF($P, \Gamma$)
              ($\mathcal{C}_1, T_1$,end)=INF($Q, \Gamma$)
              $T = \mathsf{end} \vee T_1 = \mathsf{end}$
              if $T$==end then $T_2$=$T_1$
              else if $T1$==end then $T_2$=$T$
          in ($\mathcal{C} \cup \mathcal{C}_1, T_2$,end)
INF($\Sigma_{i=1}^{n}(l_i).P_i, \Gamma$)=
          let ($\mathcal{C}_1, T_1, U_1$)=INF($P_i, \Gamma$)......($\mathcal{C}_n, T_n, U_n$)=INF($P_n, \Gamma$)
          $\mathcal{C}'$=$\bigcup_i \{U_i = U_{i+1}\}$    $\forall\ i \in 1 \ldots n-1$
          in ($\mathcal{C}' \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n$,$\&\{l_1 : T_1, \ldots, l_n : T_n\}$,$U_1$)
INF($\langle l \rangle.P, \Gamma$)=
          let ($\mathcal{C}, T, U$)=INF($P, \Gamma$)
          in($\mathcal{C}$,$\oplus\{l : T\}$,$U$)
```

Fig. 8. The algorithm to extract constraints

```
let solve $\mathcal{C}$ $\Gamma$=
match $\mathcal{C}$ with
  []->$\Gamma$
  |$T = T1$ ::$\mathcal{C}'$ -> let p=unify(T,T1) in solve(p$\mathcal{C}'$,p$\Gamma$)
  |$\alpha \bowtie T$  ::$\mathcal{C}'$ -> solve($\mathcal{C}'$@[$\alpha \bowtie T$], $\Gamma$)
  |$T \bowtie T1$ ::$\mathcal{C}'$ -> let p=compunify(T,T1) in solve(p$\mathcal{C}'$,p$\Gamma$)
```

Fig. 9. An algorithm to solve the constraints set

$$\mathsf{compunify}(\mathsf{end},\mathsf{end}) = \epsilon \qquad \mathsf{compunify}([T],[T']) = \mathsf{compunify}(T,T')$$
$$\mathsf{compunify}(?(\tilde{S}).T, !(\tilde{S}').T') = \mathsf{unify}(\tilde{S},\tilde{S}') \cdot \mathsf{compunify}(T,T')$$
$$\mathsf{compunify}(\&\{l_1 : T_1,\ldots,l_n : T_n\}, \oplus\{l'_1 : T'_1,\ldots,l_m : T'_m\}) = \bigcup_{l_i = l'_j} \mathsf{compunify}(T_i,T'_j)$$
$$\{l'_1,\ldots,l'_m\} \subseteq \{l_1,\ldots,l_n\}$$

Fig. 10. compunify

The compunify defined in Figure 10 (with symmetric cases) solves a $\bowtie$ equation by unifying the type of the tuples received and transmitted ($\cdot$ indicates the composition of substitutions and ϵ the empty substitution); the other cases follow the same pattern as their syntactic counterparts defined in Figure 7.

It is worth noticing that we cannot solve an equation of form $\alpha \bowtie T$ because this kind of equation does not contain any information. In these cases, the algorithm chooses to append the equation to the rest $\mathcal{C}'$ since another iteration could substitute the type variable with a more concrete type. The following proposition shows that this is always the case since sooner or later all service definitions become available, thanks to the well-formedness of P.

Proposition 1. *Let* $(\mathcal{C}, \_, \_) = \mathtt{INF}(P,\Gamma)$ *and* P *a closed process with respect to services and variables. For each constraint* $\alpha \bowtie [T] \in \mathcal{C}$*, it is possible to find in* $\mathcal{C}$ *a series of constraints that yields a substitution* $\{\alpha \mapsto T'\}$ *and* T' *is not a type variable.*

Proof. Note that constraints $\alpha \bowtie [T]$ are generated by the service invocation $\overline{v}.P$ and α is the type of the value v used for invocation. Suppose we first solve from $\mathcal{C}$ all the unification constraints. Consequently, we produce the new constraints set $\mathcal{C}'$. The remaining constraints $\alpha \bowtie [T]$ in $\mathcal{C}'$ are because α is introduced by an input binder. In this case we have an usage of the form $?(\alpha)$ to be compared with an usage of the form $!(T_1)$ otherwise compunify fails. If T_1 is not a type variable compunify returns the substitution $\{\alpha \mapsto T_1\}$. Otherwise, if $T_1 = \alpha_1$ is a type variable then it must exists (since P is closed and well formed) an equation $\alpha_1 \bowtie T_2$ and we can reiterate the reasoning to find out the desiderated substitution. The reasoning terminates and it is bounded by the number of service invocations in P.

Thanks to the previous Proposition we can show that even if there are unguarded appends, `solve` terminates.

Theorem 2. `solve` *terminates.*

Proof. The set of constraints decreases at each iteration except in the third case. By Proposition 1 there must exist a substitution that returns the concrete form of α in a finite number of steps. Let $|\mathcal{C}|$ be the total number of constraints in the set and d the number of the $\bowtie$ equations. The measure we are going to define is $|\mathcal{C}| + d!$, where $d!$ denotes the factorial of d. In fact, we need at most $|\mathcal{C}|$ steps to solve all the equality equations and at most all the permutations of d to find the correct resolution order of the $\bowtie$ equations. □

5 Running Examples Extracted from the Tool

We developed the described algorithm in OCaml [15] and in this section we show some examples of executions with the generated constraints set and the relative solution. Consider that the algorithm makes some initial work to alpha renaming the process in such a way that all bound and free names are different as implicitly expected by the `INF` function.

Example 2. We start with a classical functional flavor, factorial service. Even if this function is recursive, its typing does not require recursive types, as each session is isolated from each other. A client invokes the service and returns the result upwards. Furthermore, the example shows how nested services work.

$(\nu\texttt{fatt})$
$\qquad\texttt{fatt.(n).}$
$\qquad\texttt{if n=1 then }\langle 1\rangle\texttt{ else}$
$\qquad\quad(\nu\texttt{util})$
$\qquad\quad\texttt{util.}\overline{\texttt{fatt}}\texttt{.}\langle\texttt{sub(n,1)}\rangle\texttt{.(x).return x | }\overline{\texttt{util}}\texttt{.(x}_1\texttt{).return mul(x}_1\texttt{,n)}$
$\quad\texttt{| }\overline{\texttt{fatt}}\texttt{.}\langle 5\rangle\texttt{.(res).return res}$

First of all we need to instruct the tool, linking in the environment Γ only the types of the external functions; $\Gamma = \{\mathsf{sub} : int \times int \to int, \mathsf{mul} : int \times int \to int\}$. Now running `INF` with the previous process and $\Gamma' = \Gamma \cup \{\texttt{fatt} : \alpha_1, \texttt{n} : \alpha_6, \texttt{util} : \alpha_3, \texttt{x} : \alpha_5, \texttt{x}_1 : \alpha_4, \texttt{res} : \alpha_2\}$ yields the following constraints:

$$\begin{array}{lll}
\alpha_1 = [?(\alpha_6).!(int).\texttt{end}] & \alpha_3 = [!(\alpha_5).\texttt{end}] & \alpha_6 = int \\
\alpha_3 \bowtie [?(\alpha_4).\texttt{end}] & \alpha_1 \bowtie [!(int).?(\alpha_5).\texttt{end}] & \alpha_4 = int \\
\alpha_1 \bowtie [!(int).?(\alpha_2).\texttt{end}] & int = int &
\end{array}$$

The solving algorithm computes the right solution, $\texttt{fatt} : [?(int).!(int).\texttt{end}]$, $\texttt{n} : int$, $\texttt{x} : int$, $\texttt{x}_1 : int$, $\texttt{res} : int$, $\texttt{util} : [!(int).\texttt{end}]$.

Example 3. The second program shows how the type inference works for an invocation of a dynamically received service name.

$\quad(\nu\texttt{b})\texttt{((}\nu\texttt{a)(a.(}\mathbf{sum}\texttt{).(x,y).}\langle\texttt{add(x,y)}\rangle\texttt{ + (}\mathbf{inc}\texttt{).(x}_1\texttt{).}\langle\texttt{add(x}_1\texttt{,1)}\rangle\texttt{ | b.}\langle\texttt{a}\rangle\texttt{)|}$
$\qquad\overline{\texttt{b}}\texttt{.(z).}\overline{\texttt{z}}\texttt{.}\langle\mathbf{sum}\rangle\texttt{.}\langle 2,3\rangle\texttt{.(res))}$

This time Γ contains only the definition of `add` and $\Gamma' = \Gamma \cup \{\mathtt{b} : \alpha_1, \mathtt{a} : \alpha_4, \mathtt{x} : \alpha_5, \mathtt{y} : \alpha_6, \mathtt{x}_1 : \alpha_7, \mathtt{z} : \alpha_3, \mathtt{res} : \alpha_2\}$. `INF` returns

$$\alpha_4 = [\&\{\mathbf{sum} : ?(\alpha_6, \alpha_5).!(int).\mathtt{end}, \mathbf{inc} : ?(\alpha_7).!(int).\mathtt{end}\}] \quad \alpha_6 = int \quad \alpha_5 = int$$
$$\alpha_1 = [!(\alpha_4).\mathtt{end}] \quad \alpha_1 \bowtie [?(\alpha_3).\mathtt{end}] \quad \alpha_3 \bowtie [\oplus\{\mathbf{sum} : \{!(int, int).?(\alpha_2).\mathtt{end}\}]$$

and `solve` produces the solution

$$\begin{aligned}&\mathtt{b} : [!([\&\{\mathbf{sum} : ?(int, int).!(int).\mathtt{end}, \mathbf{inc} : ?(int).!(int).\mathtt{end}\}]).\mathtt{end}], \mathtt{x} : int,\\ &\mathtt{a} : [\&\{\mathbf{sum} : ?(int, int).!(int).\mathtt{end}, \mathbf{inc} : ?(int).!(int).\mathtt{end}\}], \mathtt{y} : int, \mathtt{x}_1 : int,\\ &\mathtt{z} : [\&\{\mathbf{sum} : ?(int, int).!(int).\mathtt{end}, \mathbf{inc} : ?(int).!(int).\mathtt{end}\}], \mathtt{res} : int\end{aligned}$$

Example 4. This example shows how an external function can input services and return services as well. In particular, the function lb has type $[!(int)] \times [!(int)] \rightarrow [!(int)]$; it inputs a couple of services and returns a service.

$$\begin{aligned}&(\nu\mathtt{loadbalance})(\nu\mathtt{a})(\nu\mathtt{b})(\mathtt{loadbalance}.\langle \mathsf{lb}(\mathtt{a},\mathtt{b})\rangle \,|\, \mathtt{b}.4 \,|\, \mathtt{a}.4)\\ &\qquad\qquad | \,\overline{\mathtt{loadbalance}}.(\mathtt{x}).\overline{\mathtt{x}}.(\mathtt{res}))\end{aligned}$$

The inferred types are $\mathtt{loadbalance} : [!([!(int).\mathtt{end}]).\mathtt{end}], \mathtt{a} : [!(int).\mathtt{end}], \mathtt{b} : [!(int).\mathtt{end}], \mathtt{x} : [!(int).\mathtt{end}], \mathtt{res} : int$

6 Conclusions and Future Work

In this paper we studied an algorithm to infer session types. This is a preliminary study and we studied only a restricted form of session types in a syntactically constrained language which does not give to the programmer the freedom to directly use session channels. In spite of these limitations, we shown that with respect to the simply typed π-calculus a context with dual interactions and choices needs a new type of equations allowing for duality.

Moreover, typical typing systems for session types are nondeterministic due to the choices, both internal and external, embedded in the types. In fact, standard rules leave the entire freedom; one can add and remove branches until both the invocation protocol and the service specification are not syntactically equivalent (modulo duality). Thus, we have proposed a set of syntax directed rules which uses the if-then-else to deterministically expand choice branches and a corresponding relation to be used in place of the syntactical equivalence. Successively, we have developed an algorithm to infer types, subdivided in two parts: constraints extractor and solver.

As a consequence, the present ideas can be adopted as a base to the enhancement of the algorithm, adding μ-types [11], extending the model with multi-parti session types [1] and studying the inference for [20].

References

1. Bonelli, E., Compagnoni, A.: Multipoint. session types for a distributed calculus. In: Proceedings of 3rd Trustworthy Global Computing, Sophia-Antipolis, France (2007)

2. Booth, D., Liu, C.: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer (2006), `http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327`.
3. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: A service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
4. Bruni, R., Mezzina, L.G.: A deadlock free type system for a calculus of services and sessions (submitted 2007), `http://www.di.unipi.it/~bruni/publications/scctype.ps.gz`
5. Bugliesi, M., Castagna, G., Crafa, S.: Access control for mobile agents: The calculus of boxed ambients. ACM Trans. Program. Lang. Syst. 26(1), 57–124 (2004)
6. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. 240(1), 177–213 (2000)
7. Chinnici, R., Haas, H., Lewis, A., Moreau, J.-J., et al.: Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts (2006), `http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327`
8. Chinnici, R., Moreau, J.-J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language (2006), `http://www.w3.org/TR/2006/CR-wsdl20-20060327`
9. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001), `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`
10. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
11. Gapeyev, V., Levin, M., Pierce, B.: Recursive subtyping revealed. J. Funct. Program. 12(6), 511–548 (2002)
12. Garralda, P., Compagnoni, A., Dezani-Ciancaglini, M.: Bass: boxed ambients with safe sessions. In: PPDP 2006: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming, pp. 61–72. ACM Press, New York (2006)
13. Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Inform. 42(2), 191–225 (2005)
14. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998 and ETAPS 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
15. Mezzina, L.G.: OCaml prototype of the type inference algorithm, `http://www.lmezzina.com/sesstypes.zip`
16. Robinson, J.A.: Logic and logic programming. Commun. ACM 35(3), 40–65 (1992)
17. Vasconcelos, V., Honda, K.: Principal typing-schemes in a polyadic -calculus (1992)
18. Vasconcelos, V.T.: Recursive types in a calculus of objects. Transactions of Information Processing Society of Japan 35(9), 1828–1836 (1994)
19. Vasconcelos, V.T., Tokoro, M.: A typing system for a calculus of objects. In: Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, London, UK, pp. 460–474. Springer, Heidelberg (1993)
20. Yoshida, N., Vasconcelos, V.: 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)