# A Type System for Client Progress in a Service-Oriented Calculus[⋆]

Lucia Acciai and Michele Boreale

Dipartimento di Sistemi e Informatica, Università di Firenze.
{lacciai,boreale}@dsi.unifi.it

**Abstract.** We introduce a type system providing a guarantee of *client progress* for a fragment of CaSPiS, a recently proposed process calculus for service-oriented applications. The interplay of sessioning and data-orchestration primitives makes the design of a type system for CaSPiS challenging. Our main result states that in a well-typed CaSPiS system, and in absence of divergence, any client invoking a service is guaranteed not to get stuck during the execution of a conversation protocol because of inadequate service communication capabilities.
**Keywords:** process calculi, service-oriented computing, pi-calculus, type systems

## 1 Introduction

Recent years have seen the emergence of web-based applications composed by several loosely coupled components, often referred to as web services, relying on message-passing as the sole means of cooperation. This technological shift has in turn led to the formulation of a new computational paradigm underpinning the construction of such applications and known as *Service Oriented Computing* (*SOC*). Equipping SOC with rigorous semantic foundations is the subject of a very active research area. We just mention here the SENSORIA project [17], a large, EU-funded research initiative aiming at the development of a comprehensive approach to the engineering of SOC software systems, starting from rigorous methodological foundations.

CaSPiS (*Calculus of Sessions and Pipelines*, [2]) is a language currently being considered in SENSORIA as a candidate core calculus for SOC programming. CaSPiS design, influenced both by Cook and Misra's Orc [8] and by the pi-calculus [16], is centered around the notions of *session* and of *pipeline*. In CaSPiS, these concepts, and the related programming primitives, are viewed as natural tools for structuring client-service interaction and orchestration, the following description of CaSPiS is partly adopted from [2].

In CaSPiS, service definitions and invocations are written like (nullary) input and output prefixes in CCS, thus we have:

$$s.P \quad \text{and} \quad \overline{s}.Q$$

⋆ Research supported by the Project FET-GC II IST-2005-16004 SENSORIA.

where $s$ is the name of the service. There is an important difference, though, as the bodies $P$ and $Q$ are not quite continuations, but rather protocols that, within a session, govern interaction between (instances of) the client and the server. As an example

$$\mathit{currency\_converter}.(x).\langle x * r\rangle \quad \text{and} \quad \overline{\mathit{currency\_converter}}.\langle \mathit{amount}\rangle.(y).\langle y\rangle^{\uparrow}$$

are respectively: a service that once called waits for an amount expressed in euros and then sends back its counter-value in US dollars, computed according to an exchange rate $r$; and a client that passes argument *amount* to the service, then waits for the counter-value and returns this value as a result.

A session is generated as the result of a service invocation and represents an ongoing conversation between a client and a service. In the variant of CaSPiS we consider, a session is written $[P|||Q]$, with $P$ and $Q$ the communicating protocols running at the client and at the service side, respectively. For instance, synchronization of the client and of the service described above triggers a new session

$$[\langle \mathit{amount}\rangle.(y).\langle y\rangle^{\uparrow} \ ||| \ (x).\langle x * r\rangle] \ .$$

Here, after one reduction step, the counter-value $x * r$ will be computed by the service protocol and then sent to the client:

$$[(y).\langle y\rangle^{\uparrow} \ ||| \ \langle \mathit{amount} * r\rangle] \ \rightarrow \ [\langle \mathit{amount} * r\rangle^{\uparrow} \ ||| \ \mathbf{0}] \ .$$

The remaining activity will be performed by the client side, which will emit $\mathit{amount} * r$ outside the session. In fact, values can be returned outside a session to the enclosing environment using the return operator, $\langle\cdot\rangle^{\uparrow}$. These values can be used to start new activities. To orchestrate flows of data arising from different sessions, CaSPiS provides the programmers with a *pipe* operator, written $P > Q$. As an example, pipes allow to pass the results produced by one service invocation in $P$ onto the next service $Q$ in a given chain of invocations; or to wait for the results produced by two concurrent invocations before invoking a third service. For instance, what follows is a client that invokes the service *currency_converter* and then checks if the amount is available on his bank account:

$$\overline{\mathit{currency\_converter}}.\langle \mathit{amount}\rangle.(y).\langle y\rangle^{\uparrow} > (z).\overline{\mathit{check\_bank\_availability}}.\langle z\rangle \ .$$

Very often, client-service interactions in a SOC scenario comprise not only the exchange of messages between the two main parties, but also invocation of subsidiary services. The results produced by these subsidiary invocations are used in the main (top level) session. For this reason, CaSPiS allows service invocations to be placed inside sessions, hence giving rise to hierarchies of invocations and nested sessions. As an example, suppose the exchange rate from euros to US dollars in the example above is not fixed and that service *currency_converter* calls service *exchange_rates* for obtaining the up to date rate as described below.

$$\mathit{currency\_converter}.\Big((x).\overline{\mathit{exchange\_rates}}.\big(\langle\text{“€/\$”}\rangle.(z)\langle z\rangle^{\uparrow}\big) > (r).\langle x * r\rangle\Big) \ .$$

Interaction of the client above and the new version of the $currency\_converter$ service will lead to

$$\Big[(y).\langle y\rangle^{\uparrow} \mid\mid\mid \Big([\langle\text{“€/\$”}\rangle.(z)\langle z\rangle^{\uparrow}) \mid\mid\mid R] > (r).\langle amount * r\rangle\Big)\Big]$$

where $R$ is the interaction protocol of service $exchange\_rates$. Once $R$ gets “€/$” message, it provides the up to date exchange rate $rate$, the innermost session passes this value through the pipeline and the whole process reduces to

$$\Big[(y).\langle y\rangle^{\uparrow} \mid\mid\mid \Big([\mathbf{0} \mid\mid\mid R'] > \langle amount * rate\rangle\Big)\Big]$$

and then to

$$\Big[\langle amount * rate\rangle^{\uparrow} \mid\mid\mid \Big([\mathbf{0} \mid\mid\mid R'] > \mathbf{0}\Big)\Big] .$$

The presence of pipes and nested sessions makes the dynamics of a CaSPiS session quite complex: it is substantially different from simple type-regulated interactions as found in the pi-like languages of, e.g. [11,10], or in the finite-state contract languages of [5,4,6].

The present paper is a contribution towards developing programming techniques for safe client-service interaction in a SOC scenario. Technically, we offer a type system for CaSPiS that provides guarantees of *client progress*. In practice, this means that in a well-typed CaSPiS system, and in absence of divergence, any client invoking a service is guaranteed, during the execution of a conversation protocol, not to get stuck because of inadequate service communication capabilities. More generally, we hope that some of the concepts we discuss here may be further developed and find broader applications in the future.

There are three key aspects involved in the design of our type system. A first aspect concerns *abstraction*: types focus on flows of I/O value-types and ignore the rest (actual values, service calls, ...). Specifically, types take the form of CCS-like terms describing I/O flows of processes. In fact, a tiny fragment of CCS, with no synchronization and restriction, is employed, where the role of atomic actions is played by basic types. A second aspect concerns *compliance* of client protocols with service protocols, which is essential to avoid deadlocks. In the type system, the operational abstractions provided by types are employed to effectively check client-service compliance. To this purpose, types are required to account for process I/O behaviour quite precisely. Indeed, approximation might easily result into ignoring potential client-service deadlocks. A final aspect concerns the nesting of sessions. A session at a lower level can exercise *effects* on the upper level, say the level of any enclosing session. To describe this phenomenon, we follow [3,14] and keep track, in the type system, of the behaviour at both the current level and at the level of a (fictitious) enclosing session. This results in type judgments of the form $P : [\mathsf{S}]\mathsf{T}$, where $\mathsf{S}$ is the the current-level type and $\mathsf{T}$ is the upper-level effect of $P$. Note that the distinction between types and effects we make here is somehow reminiscent of the type-and-effects systems of [18], with the difference that our effects are very simple (sequences of outputs) and are exercised on an upper level of activity rather than on a shared memory.

The version of CaSPiS considered in this paper differs from the “official” one in [2] in one important respect: we restrict our attention to the case where values can be returned outside a session only on the client side (the same restriction applies to the language considered in [3]). The theoretical reasons for doing so will be discussed in the concluding section. From a practical point of view, this limitation means that, once a session is started, for the service there will be no "feedback" of sort as to what is going on inside the session. This is somehow consistent with the idea that services should be stateless entities.

*Related work.* Our work is mainly related to Bruni and Mezzina's [3] and to Lanese et al.'s [14]. In these papers, type systems for languages affine to CaSPiS are put forward. In particular, the language considered in [3] is essentially CaSPiS with the restriction discussed above. The language of [14], SSCC, differs from CaSPiS essentially because streams, rather than pipes, are provided for data orchestration of different activities. We share with [3,14] the two-level types technique. In some important aspects, though, our system differs from theirs, resulting into a gain of simplicity and generality. These aspects we discuss below. First, we take advantage of the restriction that values can be returned only to the client and adopt a new syntax and operational semantics for sessions that spares us the necessity of explicit session names and the annoying "balancing" conditions on them (see also [2]). Second, our type system does not suffer from certain heavy restrictions of [3,14], like for example, forcing either of the two components in a parallel composition to have a null effect. Also, the client-service compliance relation we adopt is more flexible than the bare complementarity relation inherited from session-types disciplines employed in [3,14]. Finally, our client-progress theorem is an immediate consequences of two natural properties of the system (subject reduction, type safety). In particular, we do not have to resort to a complex system of obligations and capabilities *a l* Kobayashi [13,12], like [3] does. This is a benefit partly of a precise operational correspondence between processes and types and partly of our new syntax for sessions. Note that, in [14], synchronization problems related to data streams prevent achieving a deadlock-freeness result.

Both CaSPiS and SSCC evolved from *SCC* (*Serviced Centered Calculus*) [1], a language that arose from a joint effort of various partners involved in the SENSORIA consortium. The original proposal turned out later to be unsatisfactory in some important respects. In particular, SCC had no dedicated mechanisms for data orchestration and came equipped with no type system. These problems motivated the proposal of a few evolutions of SCC. As mentioned above, SSCC is stream-oriented, in that values produced by sessions are stored into dedicated queues, accessible by their names, while CaSPiS relies solely on pipes. Another evolution of SCC is the language in [19], featuring message-passing primitives for communication in all directions (within a session, from inside to outside and vice-versa).

*Structure of the paper.* The rest of the paper is organized as follows. In Section 2 we present CaSPiS$^-$, the variant of CaSPiS we will consider. *Client progress*, the property we wish to capture with our system, is also defined. A language of types is introduced in Section 3, while a type system is presented in Section 4. Results about the type system are discussed in Section 5, culminating in Corollary 1, which asserts that well-typed processes enjoy the client progress property. We conclude with a few remarks concerning the limitation of our system and further work in Section 6.

| | | | |
|---|---|---|---|
| $P, Q ::= \sum_{i\in I} \pi_i.P_i$ | Guarded Summation | $\pi ::= (x : \mathsf{b})$ | Input Prefix |
| $\mid \langle u\rangle^{\uparrow}$ | Return | $\mid \langle u\rangle$ | Output Prefix |
| $\mid s.P$ | Service Definition | | |
| $\mid \overline{u}.P$ | Service Invocation | | |
| $\mid [P \Vert\!\vert Q]$ | Session | | |
| $\mid P > Q$ | Pipeline | | |
| $\mid P \vert Q$ | Parallel Composition | | |
| $\mid (\nu s)P$ | Restriction | | |
| $\mid *P$ | Replication | | |

**Table 1.** Syntax of CaSPiS$^-$.

## 2 Processes

### 2.1 Syntax and semantics

We introduce below the variant of CaSPiS, which we christen CaSPiS$^-$, that we have chosen as a target language for our type system.

*Syntax.* We presuppose the following disjoint sets: a set $\mathcal{B}$ of *base values*, a countable set $\mathcal{N}$ of *service names* ranged over by $n, s, \ldots$ and a countable set $\mathcal{V}$ of *variables*, ranged over by $x, y, \ldots$. In the following, we let $u$ be a generic element of $\mathcal{N} \cup \mathcal{B} \cup \mathcal{V}$ and $v$ be a generic element of $\mathcal{N} \cup \mathcal{B}$. We presuppose a set $\mathcal{B}t$ of *base types*, $\mathsf{b}, \mathsf{b}', \ldots$ which include name *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. We finally presuppose a generic base-typing relation, mapping base values and service names to base types, written $v : \mathsf{b}$, with the obvious proviso that service names are mapped to sorts and base values are mapped to the remaining base types.

The syntax of the calculus is reported in Table 1. Input prefixes are annotated with types $\mathsf{b}$, which are associated to input variables. In service definitions and invocations, $s.P$ and $\overline{s}.Q$, processes $P$ and $Q$ are the protocols followed respectively by the service and client side. As in [2], the grammar defined in Table 1 should be considered as a run-time syntax. In particular sessions $[P \Vert\!\vert Q]$ can be generated at run-time, upon service invocation, but a programmer is not expected to explicitly use them. In $[P \Vert\!\vert Q]$ processes $P$ and $Q$ represent respectively the rest of the client and the service protocol to be executed. The free and bound names and variables of a term are defined as expected. In the following, we suppose each bound name in a process different from free, and we identify terms up to alpha-equivalence. We denote by $\mathrm{fn}(P)$, resp. $\mathrm{fv}(P)$, the set of free names, resp. variables, of $P$, and indicate with $\mathcal{P}$ the set of closed terms, that is, the set of process terms with no free variables. In what follows, we abbreviate the empty summation by $\mathbf{0}$.

CaSPiS$^-$ is essentially the close-free fragment of the calculus in [2], but for a major difference: in CaSPiS$^-$ sessions are one-sided. In particular, sessions are executed on the client side and all returned values are available only at this side. This simplification allows us to dispense with session names and balancing conditions on them – see [2] – which are necessary when the two sides of a sessions are distinct and far apart. Practically, this limitation means that services in CaSPiS$^-$ cannot return values and are stateless. Another, minor difference from [2] is that here returns are asynchronous. Finally, for the sake of simplicity we do not consider structured values and expressions, which can be easily accommodated.

*Semantics.* The operational semantics of the calculus is given in terms of a *labelled transition relation*, $\xrightarrow{\lambda}$ defined as the least relation generated by axioms in Table 2. Labels $\lambda$ can be of the following form: input $(v)$, output $\langle v\rangle$ or $(\nu\hat{s})\langle s\rangle$ – where $(\nu\hat{s})$ indicates that the restriction $(\nu s)$ may or may not be present –, return $\langle v\rangle^\uparrow$ or $(\nu\hat{s})\langle s\rangle^\uparrow$, service definition $(\nu\tilde{n})s\langle R\rangle$, service invocation $\overline{s}(R)$ and synchronization $\tau$. It is worth noticing that service definitions are persistent, (DEF), and only (synchronous) in-session value-passing is allowed, (S-COM$_l$) and (S-COM$_r$). As already stated, sessions are one-sided, (CALL), and possible returns arising from the service protocol $Q$ are ignored – there is no symmetric rule of (S-RET). Note that in (P-PASS) we have used an optional restriction $(\nu\hat{n})$ to indicate that the passed value might be a bound service name. Finally, note the run-time type check in (IN), which avoids type mismatch between the received object and the expected base type. From a computational point of view, this rule should not be particularly worrying, since we are only considering checks on base values. Note that in pi-like process calculi, static checks on channels are often sufficient to avoid such type mismatches – see e.g. the sorting system of [15]. In CaSPiS$^-$, this solution is not viable as communication takes place freely inside sessions. In fact, an alternative to run-time checks would be assigning "tags" to I/O actions, to regulate data-exchange inside sessions, which would essentially amount to re-introducing a channel-based discipline, which is not our main concern here. Note that this issue does not arise in [3], because, as discussed in the Introduction, their type system discards the parallel composition of two or more outputs inside sessions.

We shall often refer to a silent move $P \xrightarrow{\tau} P'$ as a *reduction*; $P \Rightarrow P'$ and $P \xRightarrow{\lambda} P'$ mean respectively $P \xrightarrow{\tau}{}^* P'$ and $P \xrightarrow{\tau}{}^* \xrightarrow{\lambda} \xrightarrow{\tau}{}^* P'$.

## 2.2 Client progress property

The client progress property will be defined in terms of an error predicate. Informally, an error occurs when the client protocol of an active session tries to send to or receive a value from the service side, but the session as a whole is blocked. This is formalized by the predicate $\rightarrow_{\mathrm{ERR}}$ defined below. In the definition, we rely on two standard notions, structural congruence and contexts, briefly introduced below. *Structural congruence*, $\equiv$, is defined as the least congruence over (open) processes preserved by substitutions and satisfying the axioms in Table 3. In the vein of [2], the laws in Table 3 comprise the structural rules for parallel composition and restriction from the pi-calculus, plus some extra scope extension laws for pipelines and sessions.

*Contexts*, $C[\cdot], C'[\cdot], \ldots$, are process terms with a hole; we shall indicate with $C[P]$ the process obtained by replacing the hole with $P$. The notion of context can be generalized to $n$-holes contexts as expected. We say a context is *static* if its hole is not under

$$(\textsc{In})\ \frac{v : \mathsf{b}}{(x : \mathsf{b}).P \xrightarrow{(v)} P[{}^{v}\!/\!_{x}]} \qquad (\textsc{Out})\ \frac{}{\langle v\rangle.P \xrightarrow{\langle v\rangle} P} \qquad (\textsc{Ret})\ \frac{}{\langle v\rangle^{\uparrow} \xrightarrow{\langle v\rangle^{\uparrow}} \mathbf{0}}$$

$$(\textsc{Rep})\ \frac{P| * P \xrightarrow{\lambda} P'}{*P \xrightarrow{\lambda} P'} \qquad (\textsc{Def})\ \frac{}{s.P \xrightarrow{s\langle P\rangle} s.P} \qquad (\textsc{Call})\ \frac{}{\overline{s}.P \xrightarrow{\overline{s}(Q)} [P|||Q]}$$

$$(\textsc{Sync}_l)\ \frac{P \xrightarrow{(\nu\tilde{n})s\langle R\rangle} P' \quad Q \xrightarrow{\overline{s}(R)} Q'}{P|Q \xrightarrow{\tau} (\nu\tilde{n})(P'|Q')} \qquad (\textsc{S-Ret})\ \frac{P \xrightarrow{(\nu\hat{v})\langle v\rangle^{\uparrow}} P'}{[P|||Q] \xrightarrow{(\nu\hat{v})\langle v\rangle} [P'|||Q]}$$

$$(\textsc{S-Pass}_l)\ \frac{P \xrightarrow{\lambda} P' \quad \lambda ::= \tau \,|\, \overline{s}(Q) \,|\, (\nu\tilde{n})s\langle Q\rangle}{[P|||Q] \xrightarrow{\lambda} [P'|||Q]} \qquad (\textsc{S-Com}_l)\ \frac{P \xrightarrow{(v)} P' \quad Q \xrightarrow{(\nu\hat{v})\langle v\rangle} Q'}{[P|||Q] \xrightarrow{\tau} (\nu\hat{v})[P'|||Q']}$$

$$(\textsc{S-Sync}_l)\ \frac{P \xrightarrow{(\nu\tilde{n})s\langle R\rangle} P' \quad Q \xrightarrow{\overline{s}(R)} Q'}{[P|||Q] \xrightarrow{\tau} (\nu\tilde{n})[P'|||Q']} \qquad (\textsc{Sum})\ \frac{\pi_j.P_j \xrightarrow{\lambda} P_j \quad j \in I \quad |I| > 1}{\sum_{i\in I} \pi_i.P_i \xrightarrow{\lambda} P_j}$$

$$(\textsc{P-Pass})\ \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq (\nu\hat{v})\langle v\rangle}{P > Q \xrightarrow{\lambda} P' > Q} \qquad (\textsc{P-Sync})\ \frac{P \xrightarrow{(\nu\hat{v})\langle v\rangle} P' \quad Q \xrightarrow{(v)} Q'}{P > Q \xrightarrow{\tau} (\nu\hat{v})(P' > Q|Q')}$$

$$(\textsc{Par}_l)\ \frac{P \xrightarrow{\lambda} P' \quad \mathrm{fn}(Q) \cap \mathrm{bn}(\lambda) = \emptyset}{P|Q \xrightarrow{\lambda} P'|Q} \qquad (\textsc{R-Pass})\ \frac{P \xrightarrow{\lambda} P' \quad n \notin \mathrm{n}(\lambda)}{(\nu n)P \xrightarrow{\lambda} (\nu n)P'}$$

$$(\textsc{Open})\ \frac{P \xrightarrow{\lambda} P' \quad \lambda ::= (\nu\tilde{a})s\langle R\rangle \,|\, (\nu\tilde{a})\langle n\rangle \,|\, (\nu\tilde{a})\langle n\rangle^{\uparrow} \quad s \neq n \quad n \in \mathrm{fn}(\lambda)}{(\nu n)P \xrightarrow{(\nu n)\lambda} P'}$$

Symmetric versions of (S-SYNC$_l$), (PAR$_l$), (S-PASS$_l$), (S-SYNC$_l$) and (S-COM$_l$) are not displayed.

**Table 2.** Labeled Semantics

the scope of a dynamic operator (input and output prefixes, replication, service definitions and invocations, and the right-hand side of a pipeline). In essence, active subterms in a process $P$ are those surrounded by a static context.

**Definition 1 (error).** *$P \rightarrow_{\mathrm{ERR}}$ if and only if whenever $P \equiv C[[Q|||R]]$, with $C[\cdot]$ static, and $Q \xrightarrow{\lambda}$, with $\lambda ::= (v) \,|\, (\nu\hat{v})\langle v\rangle$, then $[Q|||R] \overset{\lambda'}{\not\rightarrow}$, with $\lambda' ::= \tau \,|\, \overline{s}(P')$.*

A process guarantees client progress if it is error-free at run-time.

**Definition 2 (client progress).** *Let be $P \in \mathcal{P}$. We say $P$* guarantees client progress *if and only if whenever $P \Rightarrow P'$ then $P' \not\rightarrow_{\mathrm{ERR}}$.*

| | | |
|---|---|---|
| $(P\|Q)\|R \equiv P\|(Q\|R)$ | $P\|Q \equiv Q\|P$ | $P\|\mathbf{0} \equiv P$ |
| $(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$ | $*P \equiv *P\|P$ | |
| $(\nu n)P\|Q \equiv (\nu n)(P\|Q)$ | $(\nu n)P > Q \equiv (\nu n)(P > Q)$ | if $n \notin \mathrm{fn}(Q)$ |
| $[Q\|\|\|((\nu n)P)] \equiv (\nu n)[Q\|\|\|P]$ | $[((\nu n)P)\|\|\|Q] \equiv (\nu n)[P\|\|\|Q]$ | if $n \notin \mathrm{fn}(Q)$ |

**Table 3.** Structural Congruence

The above definition of error may seem too liberal, as absence of error does not actually guarantee progress of the session if $[Q|||R] \xrightarrow{\overline{s}(P')}$ and service $s$ is not available. In fact, we are interested in processes where such situations do not arise: we call these processes well-formed, and define them formally below. First, we need a notion of $s$-receptive process, a process where a service definition for the service name $s$ is available under a static context, hence is active.

**Definition 3 ($s$-receptive process).** *Let be $P$ an (open) process. $P$ is* $s$-receptive *if $s \in \mathrm{fn}(P)$ and $P \equiv C[s.R]$ for some static $C[\cdot]$ not binding $s$.*

**Definition 4 (well-formed process).** *Let be $P \in \mathcal{P}$. $P$ is* well-formed *if and only if for each $s \in \mathrm{fn}(P)$ $P$ is $s$-receptive and whenever $P \equiv C[(\nu s)Q]$ process $Q$ is $s$-receptive.*

Well-formedness is preserved by reductions.

**Lemma 1.** *Let $P$ be well-formed. $P \Rightarrow P'$ implies $P'$ is well-formed.*

The following lemma ensures that, in well-formed processes, each active service call can be immediately served, thus substantiating our previous claim that our definition of error is adequate for well-formed processes.

**Lemma 2.** *Let $P \in \mathcal{P}$ be well-formed. If $P \equiv C[\overline{s}.Q]$, with $C[\cdot]$ static, then either $C[\cdot] = C_0\big[\, C_1[\cdot] \,|\, C_2[s.R] \,\big]$, $C[\cdot] = C_0\big[\, [C_1[\cdot]|||C_2[s.R]] \,\big]$ or $C[\cdot] = C_0\big[\, [C_1[s.R]|||C_2[\cdot]] \,\big]$, for some static contexts $C_0[\cdot]$, $C_1[\cdot]$ and $C_2[\cdot]$, and for some process $R$.*

## 3 Types

In this section we introduce syntax and semantics of types, essentially a fragment of CCS corresponding to BPP processes [7].

The set $\mathcal{T}$ of types is defined by the grammar in Table 4. Recall that $\mathsf{b}, \mathsf{b}', \ldots$ range over base types in $\mathcal{B}t$, including sorts. Notice that, like in [3], we need not nested session types in our system, because in order to check session safety it is sufficient to check local, in-session communications. In what follows we abbreviate with $\mathsf{0}$ the empty summation type.

The semantics of types is described in terms of a labelled transition relation, $\xrightarrow{\alpha}$, derived from the axioms in Table 5. It is worth noticing that input and output prefixes, $?\mathsf{b}$

$$\begin{array}{llll} \mathsf{T},\mathsf{S},\mathsf{U},\mathsf{V} ::= \sum_i \alpha_i.\mathsf{T}_i & \text{Guarded Summation} & \alpha ::= !\mathsf{b} & \text{Output Prefix} \\ \quad \mid \mathsf{T} \mid \mathsf{T} & \text{Interleaving} & \quad \mid ?\mathsf{b} & \text{Input Prefix} \\ \quad \mid *\mathsf{T} & \text{Replication} & & \end{array}$$

**Table 4.** Syntax of types.

$$(\textsc{Sum-t})\ \frac{j \in I}{\sum_{i \in I} \alpha_i.\mathsf{T}_i \xrightarrow{\alpha_j} \mathsf{T}_j} \qquad (\textsc{Par-t}_l)\ \frac{\mathsf{T} \xrightarrow{\alpha} \mathsf{T}'}{\mathsf{T}|\mathsf{S} \xrightarrow{\alpha} \mathsf{T}'|\mathsf{S}}$$

$$(\textsc{Par-t}_r)\ \frac{\mathsf{S} \xrightarrow{\alpha} \mathsf{S}'}{\mathsf{T}|\mathsf{S} \xrightarrow{\alpha} \mathsf{T}|\mathsf{S}'} \qquad (\textsc{Rep-t})\ \frac{\mathsf{T}|*\mathsf{T} \xrightarrow{\alpha} \mathsf{T}'}{*\mathsf{T} \xrightarrow{\alpha} \mathsf{T}'}$$

**Table 5.** Labeled transition system of types.

and $!\mathsf{b}$, cannot synchronize with each other – we only have interleaving in this fragment of CCS.

The basic requirement for ensuring client progress is *type compliance* between client and service protocols involved in sessions, defined below. In the following, we indicate with $\overline{\alpha}$ the *coaction* of $\alpha$: $\overline{?\mathsf{b}} = !\mathsf{b}$ and $\overline{!\mathsf{b}} = ?\mathsf{b}$. This notation is extended to sets of actions as expected. Moreover, we indicate with $I(\mathsf{S})$ the set of initial actions $\mathsf{S}$ can perform: $I(\mathsf{S}) = \{\alpha \mid \exists \mathsf{S}' : \mathsf{S} \xrightarrow{\alpha} \mathsf{S}'\}$. Type compliance is defined co-inductively and guarantees that, given two compliant types $\mathsf{S}$ and $\mathsf{T}$, either $\mathsf{S}$ is stuck, or there is at least one action from $\mathsf{S}$ matched by a coaction from $\mathsf{T}$. Formally:

**Definition 5 (type compliance).** *Let be $\mathsf{S}, \mathsf{T} \in \mathcal{T}$.* Type compliance *is the largest relation on types such that whenever $\mathsf{S}$ is compliant with $\mathsf{T}$, written $\mathsf{S} \propto \mathsf{T}$, it holds that either $I(\mathsf{S}) = \emptyset$ or $K = I(\mathsf{S}) \cap \overline{I(\mathsf{T})} \neq \emptyset$ and for each $\alpha \in K$ and each $\mathsf{S}'$ and $\mathsf{T}'$ such that $\mathsf{S} \xrightarrow{\alpha} \mathsf{S}'$ and $\mathsf{T} \xrightarrow{\overline{\alpha}} \mathsf{T}'$, it holds that $\mathsf{S}' \propto \mathsf{T}'$.*

## 4 A type system for client progress

In this section we introduce a type system that ensures client progress, that is, ensures that sessions cannot block as long as the client's protocol is willing to do some action.

The type system is along the lines of those in [3,14] and is reported in Table 6. We presuppose a mapping *ob* from sorts $\{\mathcal{S}, \mathcal{S}', \ldots\}$ to types $\mathcal{T}$, with the intended meaning that if $ob(\mathcal{S}) = \mathsf{T}$ then names of sort $\mathcal{S}$ represent services whose abstract protocol is $\mathsf{T}$. We take $s : \mathsf{T}$ as an abbreviation of $s : \mathcal{S}$ and $ob(\mathcal{S}) = \mathsf{T}$ for some $\mathcal{S}$. A *context* $\Gamma$ is a finite partial mapping from types to variables. For $u$ a service name, a base value or a

$$\text{(T-OUT)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \Gamma \vdash u : \mathsf{b}}{\Gamma \vdash \langle u \rangle.P : [!\mathsf{b}.\mathsf{S}]\mathsf{T}} \qquad \text{(T-RES)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T}}{\Gamma \vdash (\nu a)P : [\mathsf{S}]\mathsf{T}}$$

$$\text{(T-INP)}\ \frac{\Gamma, x : \mathsf{b} \vdash P : [\mathsf{S}]\mathsf{T}}{\Gamma \vdash (x : \mathsf{b}).P : [?\mathsf{b}.\mathsf{S}]\mathsf{T}} \qquad \text{(T-PAR)}\ \frac{\Gamma \vdash P : [\mathsf{S}_1]\mathsf{T}_1 \quad \Gamma \vdash Q : [\mathsf{S}_2]\mathsf{T}_2}{\Gamma \vdash P|Q : [\mathsf{S}_1|\mathsf{S}_2](\mathsf{T}_1|\mathsf{T}_2)}$$

$$\text{(T-RET)}\ \frac{\Gamma \vdash u : \mathsf{b}}{\Gamma \vdash \langle u \rangle^{\uparrow} : [0]!\mathsf{b}} \qquad \text{(T-SUM)}\ \frac{\forall i \in I :\ \Gamma \vdash \pi_i.P_i : [\alpha_i.\mathsf{S}_i]\mathsf{T} \quad |I| \neq 1}{\Gamma \vdash \sum_{i \in I} \pi_i.P_i : [\sum_{i \in I} \alpha_i.\mathsf{S}_i]\mathsf{T}}$$

$$\text{(T-DEF)}\ \frac{s : \mathsf{V} \quad \Gamma \vdash P : [\mathsf{V}]0}{\Gamma \vdash s.P : [0]0} \qquad \text{(T-CALL)}\ \frac{\Gamma \vdash u : \mathsf{V} \quad \Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \mathsf{S} \propto \mathsf{V}}{\Gamma \vdash \overline{u}.P : [\mathsf{T}]0}$$

$$\text{(T-REP)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T}}{\Gamma \vdash *P : [*\mathsf{S}] * \mathsf{T}} \qquad \text{(T-SESS)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{U} \quad \Gamma \vdash Q : [\mathsf{T}]0 \quad \mathsf{S} \propto \mathsf{T}}{\Gamma \vdash [P|||Q] : [\mathsf{U}]0}$$

$$\text{(T-PIPE)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \Gamma \vdash Q : [?\mathsf{b}.\mathsf{U}]\mathsf{V} \quad \mathrm{monomf}(\mathsf{S}, \mathsf{b}) \quad \mathrm{NoSum}(\mathsf{S})}{\Gamma \vdash P > Q : [\mathsf{S} \bowtie \mathsf{U}](\mathsf{T}|\mathsf{S} @ \mathsf{V})}$$

**Table 6.** Rules of the type system

variable, we take

$$\Gamma \vdash u : \mathsf{T}$$

to mean either that $u = s : \mathsf{T}$, or $u = v : \mathsf{b}$ or $u = x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) = \mathsf{T}$. Type judgments are of the form $\Gamma \vdash P : [\mathsf{S}]\mathsf{T}$, where $\Gamma$ is a context, $P$ is a possibly open process with $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$ and $\mathsf{S}$ and $\mathsf{T}$ are types. Informally, $\mathsf{S}$ and $\mathsf{T}$ represent respectively the *in-session*, or *internal*, and the *external* types of $P$. The first one describes inputs and outputs $P$ can perform at the current session level, while the second one represents the outputs $P$ can perform at the parent level – which correspond to $P$'s returns. As already discussed in the Introduction, the external type $\mathsf{T}$ describes the effects produced outside the enclosing session, that is the effects visible one level up.

Rule (T-DEF) checks that the internal type of the service protocol corresponds to the type expected by the sorting system; moreover, the rule requires the absence of external effects, hence, as discussed earlier, no returns are allowed on the service protocol. Concerning rules (T-CALL) and (T-SESS), it is worth noticing that the premises ensure compliance between client and service internal types. Rule (T-SUM) requires that each summand exposes the same external type: intuitively, sums are resolved as internal choices from the point of view of an enclosing session, hence which branch is chosen should not matter as for the external effect. Finally, rule (T-PIPE) deserves some explanations. We put some limitations on the types of the pipeline operands. First, the right-hand process $Q$ is a single, input-prefixed process of type $?\mathsf{b}.\mathsf{U}$, ready to receive a value. Second, we make sure, through predicate $\mathrm{NoSum}(\mathsf{S})$, that the left-hand $P$'s type does not contain any summation. Third, we make sure, through predicate $\mathrm{monomf}(\mathsf{S},\mathsf{b})$, that the type of the left-hand side of a pipeline is "monomorphic", that is, contains only outputs of the given type $\mathsf{b}$. Formal definition of $\mathrm{NoSum}(\mathsf{S})$ and $\mathrm{monomf}(\mathsf{S},\mathsf{b})$ are obvious and omitted. We will come back to these restrictions in Remark 1.

The auxiliary functions $\bowtie$ and $@$ are used to build respectively the internal and the external type of $P > Q$ starting from the types of $P$ and $Q$. In essence, both $\mathsf{S} \bowtie \mathsf{U}$ and $\mathsf{S} @ \mathsf{V}$ spawn a new copy of type $\mathsf{U}$ and $\mathsf{V}$, respectively, in correspondence of each output prefix in $\mathsf{S}$. The main difference is that in $@$ inputs in $\mathsf{S}$ are discarded, while in $\bowtie$ they are preserved. This because $\mathsf{S}$ is an internal type, hence its actions cannot be observed from the external viewpoint. Formally, $\mathsf{S} \bowtie \mathsf{U}$ and $\mathsf{S} @ \mathsf{V}$ are inductively defined on the structure of $\mathsf{S}$ as follows.

$$\begin{array}{rclrcl}
!\mathsf{b}.\mathsf{S} \bowtie \mathsf{U} &=& \mathsf{U}|(\mathsf{S} \bowtie \mathsf{U}) & !\mathsf{b}.\mathsf{S} @ \mathsf{U} &=& \mathsf{U}|(\mathsf{S} @ \mathsf{U}) \\
?\mathsf{b}.\mathsf{S} \bowtie \mathsf{U} &=& ?\mathsf{b}.(\mathsf{S} \bowtie \mathsf{U}) & ?\mathsf{b}.\mathsf{S} @ \mathsf{U} &=& \mathsf{S} @ \mathsf{U} \\
*\mathsf{S} \bowtie \mathsf{U} &=& *(\mathsf{S} \bowtie \mathsf{U}) & *\mathsf{S} @ \mathsf{U} &=& *(\mathsf{S} @ \mathsf{U}) \\
(\mathsf{S}_1|\mathsf{S}_2) \bowtie \mathsf{U} &=& (\mathsf{S}_1 \bowtie \mathsf{U})|(\mathsf{S}_2 \bowtie \mathsf{U}) & (\mathsf{S}_1|\mathsf{S}_2) @ \mathsf{U} &=& (\mathsf{S}_1 @ \mathsf{U})|(\mathsf{S}_2 @ \mathsf{U})
\end{array}$$

Note that $\mathrm{NoSum}(\mathsf{S})$ ensures the absence of summations on the internal type $\mathsf{S}$, hence we intentionally omit definitions of $\bowtie$ and $@$ for this case.

*Example 1 (pipelines).* Consider the process $P$ below, which calls two services *ansa* and *bbc*, supposed to reply by sending a newspage of type $\mathsf{news}$, returns an acknowledgment of type $\mathsf{ack}$, sends the received news by e-mail to address $a$ and outputs an acknowledgment:

$$\begin{aligned}
P \triangleq (\nu\, callnews)\Big( &\big(callnews.\big(\overline{ansa}.(y : \mathsf{news}).\langle y\rangle^{\uparrow} \mid \overline{bbc}.(y : \mathsf{news}).\langle y\rangle^{\uparrow}\big) \\
&\mid \overline{callnews}.(w : \mathsf{news}).(z : \mathsf{news}).\langle (w \cdot z)\rangle^{\uparrow}\big) \\
&\quad > (x : (\mathtt{news} \times \mathtt{news})).\big(\langle \mathsf{ack}\rangle^{\uparrow} \mid \overline{Email}.\langle (x, a)\rangle.(y : \mathsf{ack}).\langle y\rangle^{\uparrow}\big)\Big)
\end{aligned}$$

where we suppose service $Email$ is defined elsewhere with associated protocol of the expected type $?(\mathsf{news} \times \mathsf{news} \times \mathsf{eAddr}).!\mathsf{ack}$.
Suppose $callnews : !\mathsf{news} \,|\, !\mathsf{news}$. Then the left hand side of the pipeline is of type $[!(\mathsf{news} \times \mathsf{news})]0$ and the right one of type $[?(\mathsf{news} \times \mathsf{news}).!\mathsf{ack}]!\mathsf{ack}$. Hence, given that $!(\mathsf{news} \times \mathsf{news}) \bowtie !\mathsf{ack} = !\mathsf{ack}$ and $!(\mathsf{news} \times \mathsf{news}) @ !\mathsf{ack} = !\mathsf{ack}$, the whole process $P$ has associated type $[!\mathsf{ack}]!\mathsf{ack}$.

*Remark 1 (summations and pipelines).* We discuss here the necessity of banning summations on both side of pipelines. Suppose summations on the left hand side are allowed and consider e.g. the following process

$$P \triangleq \big((x : \mathsf{int}).(\langle x\rangle \mid \langle x\rangle \mid \langle x\rangle^{\uparrow}) + (y : \mathsf{int}).(\langle y\rangle \mid \langle y\rangle^{\uparrow})\big) > (w : \mathsf{int}).\langle w\rangle^{\uparrow} .$$

It is clear that

$$\begin{aligned}
(x : \mathsf{int}).(\langle x\rangle \mid \langle x\rangle \mid \langle x\rangle^{\uparrow}) + (y : \mathsf{int}).(\langle y\rangle \mid \langle y\rangle^{\uparrow}) &: [?\mathsf{int}.(!\mathsf{int} \,|\, !\mathsf{int}) + ?\mathsf{int}.!\mathsf{int}]!\mathsf{int} \\
(w : \mathsf{int}).\langle w\rangle^{\uparrow} &: [?\mathsf{int}]!\mathsf{int} .
\end{aligned}$$

And by definition of $\bowtie$ and $@$, $P : [\mathsf{S}]\mathsf{T}$ with

$$\begin{aligned}\mathsf{S} &\triangleq (?\mathsf{int}.(!\mathsf{int} \mid !\mathsf{int}) + ?\mathsf{int}.!\mathsf{int}) \bowtie \mathsf{0} && = \mathsf{0}\\ \mathsf{T} &\triangleq !\mathsf{int} \mid \big((?\mathsf{int}.(!\mathsf{int} \mid !\mathsf{int}) + ?\mathsf{int}.!\mathsf{int})\, @ !\mathsf{int}\big) && = !\mathsf{int} \mid \big((!\mathsf{int} \mid !\mathsf{int}) + !\mathsf{int}\big)\ .\end{aligned}$$

But $\mathsf{T}$ contains a non-guarded summation, hence $\mathsf{T} \notin \mathcal{T}$.

Similarly, suppose that summations at top level are allowed on the right-hand side of pipelines, like in

$$Q \triangleq \langle 1 \rangle > \big((x : \mathsf{int}).((z : \mathsf{int}).R_1 \mid (w : \mathsf{int}).R_2) \ + \ (y : \mathsf{int}).R_3\big)$$

the internal type associated to $Q$ is

$$!\mathsf{int} \bowtie \big((?\mathsf{int}.\mathsf{T}_{R_1} \mid ?\mathsf{int}.\mathsf{T}_{R_2}) + \mathsf{T}_{R_3}\big) \ = \ (?\mathsf{int}.\mathsf{T}_{R_1} \mid ?\mathsf{int}.\mathsf{T}_{R_2}) + \mathsf{T}_{R_3}$$

which contains a non-guarded summation. In fact, we might type sums with distinct input prefixes (external determinism only). In such a manner, each output performed by the left-hand side must be deterministically associated to one choice on the right one and no summation would arise by $\bowtie$. We have preferred to restrict our attention to pipelines where the right-hand side does not contain summations at top level for the sake of simplicity.

Let us now discuss some important differences with [3], relative to how pipelines and parallel compositions are managed. In typing a pipeline, Bruni and Mezzina require that the left-hand side be a single output if the right-hand side contains more than a single input (or the vice-versa). As discussed, we only require absence of summations on the left-hand side. E.g. in [3] the process $(x).\big(*\langle x \rangle\big) > (y).\overline{s}.\langle y \rangle.(z).\langle z \rangle^{\uparrow}$, which receives a value and uses it to call service $s$ an unbounded number of times, is not well-typed, while it is in our system. Concerning parallel composition, they require that either of the two components has a null type. This means that, e.g. a process invoking two services in parallel, and then return something, like in Example 1, are not well typed in their system. Concerning sessions, in [3] the authors decide to keep the two-sided structure of the original calculus, but ignore all effects on the service side. From the point of view of expressiveness, this is essentially equivalent to using one-sided sessions, like we do.

## 5 Results

The first step towards proving that well-typed processes guarantee client progress is establishing the usual subject reduction property (Proposition 1). Next, we prove that if a type is not stuck, the associated process is not stuck either (Proposition 2). Finally, type safety (Theorem 1), stating that a well typed process cannot immediately generate an error, is sufficient to conclude.

In the following, we denote by $\Gamma \vdash^n P : [\mathsf{S}]\mathsf{T}$ a type judgment whose derivation from the rules in Table 6 has depth $n$. Moreover, we abbreviate $\emptyset \vdash P : [\mathsf{S}]\mathsf{T}$ as $P : [\mathsf{S}]\mathsf{T}$. Finally, we say $P \in \mathcal{P}$ is *well-typed* if $P : [\mathsf{S}]\mathsf{T}$ for some $\mathsf{S}$ and $\mathsf{T}$.

**Lemma 3 (substitution).** *If $\Gamma, x : \mathsf{b} \vdash^n P : [\mathsf{S}]\mathsf{T}$ and $v : \mathsf{b}$ then $\Gamma \vdash^m P[v/x] : [\mathsf{S}]\mathsf{T}$, with $m \leq n$.*

**Lemma 4.** *1. Whenever $\mathsf{S} \xrightarrow{?\mathsf{b}} \mathsf{S}'$ then $\mathsf{S} \bowtie \mathsf{T} \xrightarrow{?\mathsf{b}} \mathsf{S}' \bowtie \mathsf{T}$.*
*2. Whenever $\mathsf{S} \xrightarrow{!\mathsf{b}} \mathsf{S}'$ then $\mathsf{S} \bowtie \mathsf{T} = \mathsf{T}|\mathsf{S}' \bowtie \mathsf{T}$.*
*3. Whenever $\mathsf{S} \bowtie \mathsf{T} \xrightarrow{?\mathsf{b}} \mathsf{V}$ then either $\mathsf{S} \xrightarrow{?\mathsf{b}} \mathsf{S}'$ and $\mathsf{V} = \mathsf{S}' \bowtie \mathsf{T}$ or $\mathsf{S} \xrightarrow{!\mathsf{b}'} \mathsf{S}'$, $\mathsf{T} \xrightarrow{?\mathsf{b}} \mathsf{T}'$ and $\mathsf{V} = \mathsf{T}'|\mathsf{S}' \bowtie \mathsf{T}$.*
*4. Whenever $\mathsf{S} \bowtie \mathsf{T} \xrightarrow{!\mathsf{b}} \mathsf{V}$ then $\mathsf{S} \xrightarrow{!\mathsf{b}'} \mathsf{S}'$, $\mathsf{T} \xrightarrow{!\mathsf{b}} \mathsf{T}'$ and $\mathsf{V} = \mathsf{T}'|\mathsf{S}' \bowtie \mathsf{T}$.*

**Lemma 5.** *1. Whenever $\mathsf{S} \xrightarrow{?\mathsf{b}} \mathsf{S}'$ then $\mathsf{S} @ \mathsf{T} = \mathsf{S}' @ \mathsf{T}$.*
*2. Whenever $\mathsf{S} \xrightarrow{!\mathsf{b}} \mathsf{S}'$ then $\mathsf{S} @ \mathsf{T} = \mathsf{T}|\mathsf{S}' @ \mathsf{T}$.*
*3. Whenever $\mathsf{S} @ \mathsf{T} \xrightarrow{!\mathsf{b}} \mathsf{V}$ then $\mathsf{S} \xrightarrow{!\mathsf{b}'} \mathsf{S}'$, $\mathsf{T} \xrightarrow{!\mathsf{b}} \mathsf{T}'$ and $\mathsf{V} = \mathsf{T}'|\mathsf{S}' @ \mathsf{T}$.*

**Proposition 1 (subject reduction).** *Suppose $P : [\mathsf{S}]\mathsf{T}$. Then*

1. *whenever $P \xrightarrow{(v)} P'$, for some $v : \mathsf{b}$, then $\mathsf{S} \xrightarrow{?\mathsf{b}} \mathsf{S}'$ and $P' : [\mathsf{S}']\mathsf{T}$;*
2. *whenever $P \xrightarrow{(\nu\hat{v})\langle v\rangle} P'$, for some $v : \mathsf{b}$, then $\mathsf{S} \xrightarrow{!\mathsf{b}} \mathsf{S}'$ and $P' : [\mathsf{S}']\mathsf{T}$;*
3. *whenever $P \xrightarrow{\langle v\rangle^{\uparrow}} P'$, with $v : \mathsf{b}$, then $\mathsf{T} \xrightarrow{!\mathsf{b}} \mathsf{T}'$ and $P' : [\mathsf{S}]\mathsf{T}'$;*
4. *whenever $P \xrightarrow{(\nu\tilde{n})s\langle Q\rangle} P'$ then $P' : [\mathsf{S}]\mathsf{T}$;*
5. *whenever $P \xrightarrow{\overline{s}(Q)} P'$, with $s : \mathsf{U}$ and $Q : [\mathsf{U}]0$, then $P' : [\mathsf{S}]\mathsf{T}$;*
6. *whenever $P \xrightarrow{\tau} P'$ then $P' : [\mathsf{S}]\mathsf{T}$.*

*Proof.* The proof is straightforward by induction on the derivation of $P : [\mathsf{S}]\mathsf{T}$ and proceeds by distinguishing the last tying rule applied. The most interesting case is (T-SESS), which we examine below (concerning other cases, note that case (T-INP) relies on Lemma 3 and (T-PIPE) relies on Lemma 3, 4 and 5).

**(T-SESS):** by $[P|||Q] : [\mathsf{S}]0$ and the premises of the rule, we get $P : [\mathsf{T}]\mathsf{S}$, $Q : [\mathsf{U}]0$ and $\mathsf{T} \propto \mathsf{U}$. We distinguish various cases, depending on the rule applied for deducing $[P|||Q] \xrightarrow{\lambda}$.

**(S-RET):** $\lambda = (\nu\hat{v})\langle v\rangle$ and by the premises of the rule and (S-RET), it must be $P \xrightarrow{\langle v\rangle^{\uparrow}} P'$. Suppose $v : \mathsf{b}$. Hence, by applying the inductive hypothesis to $P$, we get $P' : [\mathsf{T}]\mathsf{S}'$, with $\mathsf{S} \xrightarrow{!\mathsf{b}} \mathsf{S}'$, and $[P'|||Q] : [\mathsf{S}']0$, by (T-SESS).

**(S-PASS$_l$):** by the premises of the rule, we get $P \xrightarrow{\lambda} P'$, and by applying the inductive hypothesis to $P$, we get $P' : [\mathsf{T}]\mathsf{S}$. Therefore, $[P'|||Q] : [\mathsf{S}]0$, by (T-SESS).

**(S-COM$_l$):** $\lambda = \tau$ and by the premises of the rule, we get $P \xrightarrow{(v)} P'$ and $Q \xrightarrow{\langle v\rangle} Q'$. Suppose that $v : \mathsf{b}$. By applying the inductive hypothesis to both $P$ and $Q$, we get $\mathsf{T} \xrightarrow{?\mathsf{b}} \mathsf{T}'$, $\mathsf{U} \xrightarrow{!\mathsf{b}} \mathsf{U}'$, $P : [\mathsf{T}']\mathsf{S}$ and $Q : [\mathsf{U}']0$. Moreover, by definition of $\propto$ it must be $\mathsf{T}' \propto \mathsf{U}'$. Hence, by (T-SESS), $[P'|||Q'] : [\mathsf{S}]0$.

**(S-SYNC$_l$):** $\lambda = \tau$ and by the premises of the rule, we get $P \xrightarrow{(\nu\tilde{n})s\langle R\rangle} P'$, $Q \xrightarrow{\overline{s}(R)} Q'$, $P' : [\mathsf{T}]\mathsf{S}$ and $Q' : [\mathsf{U}]0$. Therefore, by (T-SESS), we get $[P|||Q] : [\mathsf{S}]0$.

**(S-Pass$_r$), (S-Com$_r$), (S-Sync$_r$):** the proof proceeds in a similar way.

**Proposition 2.** *Suppose $P : [\mathsf{S}]\mathsf{T}$. Then:*

1. *whenever $\mathsf{S} \xrightarrow{\alpha}$ then $P \xrightarrow{\lambda}$ with $\lambda ::= \tau \mid \overline{s}(Q) \mid \lambda'$ and either $\lambda' = (v)$, if $\alpha =?\mathsf{b}$, or $\lambda' = (\nu\hat{v})\langle v\rangle$, if $\alpha =!\mathsf{b}$, for some $v : \mathsf{b}$;*
2. *whenever $\mathsf{T} \xrightarrow{!\mathsf{b}}$ then $P \xrightarrow{\lambda}$ with $\lambda ::= \tau \mid \langle v\rangle^{\uparrow} \mid (v') \mid (\nu\hat{v}')\langle v'\rangle \mid \overline{s}(Q)$, for some $v : \mathsf{b}$.*

*Proof.* The proof is straightforward by induction on the derivation of $P : [\mathsf{S}]\mathsf{T}$ and proceeds by distinguishing the last typing rule applied. For the first result, the most interesting cases are (T-Sess) and (T-Pipe).

**(T-Sess):** by $[P|||Q] : [\mathsf{S}]0$ and the premises of the rule, we get $P : [\mathsf{T}]\mathsf{S}$, $Q : [\mathsf{U}]0$ and $\mathsf{S} \propto \mathsf{U}$.
By applying the inductive hypothesis to $P$, given that it must be $\alpha =!\mathsf{b}$ for some $\mathsf{b}$, we get $P \xrightarrow{\lambda}$ with $\lambda ::= \tau \mid \langle v\rangle^{\uparrow} \mid (v') \mid (\nu\hat{v}')\langle v'\rangle \mid \overline{s}(Q)$, for some $v : \mathsf{b}$.
If $P \xrightarrow{\tau}$, then $[P|||Q] \xrightarrow{\tau}$, by (S-Pass$_l$). Similarly, if $P \xrightarrow{\overline{s}(Q)}$, then $[P|||Q] \xrightarrow{\overline{s}(Q)}$, by (S-Pass$_l$).
If $P \xrightarrow{\langle v\rangle^{\uparrow}}$ then $[P|||Q] \xrightarrow{\langle v\rangle}$ with $v : \mathsf{b}$, by (S-Ret).
Otherwise, by Proposition 1 (subject reduction) and $\lambda ::= (v') \mid (\nu\hat{v}')\langle v'\rangle$ for some $v' : \mathsf{b}'$, we get $\mathsf{T} \xrightarrow{\alpha}$, with $\alpha ::=?\mathsf{b}' \mid !\mathsf{b}'$. Hence, by $\propto$, $\mathsf{U} \xrightarrow{\overline{\alpha}}$ and by applying the inductive hypothesis to $Q$ we get either $Q \xrightarrow{\overline{\lambda}}$, $Q \xrightarrow{\tau}$, or $Q \xrightarrow{\overline{s}(Q)}$. In the first case, either (S-Com$_l$) or (S-Com$_r$) can be applied for deducing $[P|||Q] \xrightarrow{\tau}$. In both the second and the third case, rule (S-Pass$_r$) can be applied for deducing either $[P|||Q] \xrightarrow{\tau}$ or $[P|||Q] \xrightarrow{\overline{s}(Q)}$.

**(T-Pipe):** by $P > Q : [\mathsf{S} \bowtie \mathsf{U}]\mathsf{T}|\mathsf{S} @ \mathsf{V}$ and the premises of the rule, we get $P : [\mathsf{S}]\mathsf{T}$, $Q : [?\mathsf{b}'.\mathsf{U}]\mathsf{V}$, $\mathrm{NoSum}(\mathsf{S})$ and $\mathrm{monomf}(\mathsf{S},\mathsf{b}')$.
Suppose $\alpha =?\mathsf{b}$. By Lemma 4, $\mathsf{S} \bowtie \mathsf{U} \xrightarrow{?\mathsf{b}}$ implies either $\mathsf{S} \xrightarrow{?\mathsf{b}}$ or $\mathsf{S} \xrightarrow{!\mathsf{b}'}$ and $\mathsf{U} \xrightarrow{?\mathsf{b}}$.
In both cases, by applying the inductive hypothesis to $P$ we get $P \xrightarrow{\lambda}$, with $\lambda ::= \tau \mid (v) \mid (\nu\hat{v}')\langle v'\rangle \mid \overline{s}(Q)$, for some $v : \mathsf{b}$ and $v' : \mathsf{b}'$. Therefore, either $P > Q \xrightarrow{\tau}$, $P > Q \xrightarrow{(v)}$ or $P > Q \xrightarrow{\overline{s}(Q)}$, by (P-Pass) and (P-Sync).
Suppose $\alpha =!\mathsf{b}$. By Lemma 4, $\mathsf{S} \xrightarrow{!\mathsf{b}'}$, $\mathsf{S} \bowtie \mathsf{U} = \mathsf{S}' \bowtie \mathsf{U}|\mathsf{U}$ and $\mathsf{U} \xrightarrow{!\mathsf{b}}$. Hence, again by applying the inductive hypothesis to $P$, we get either $P > Q \xrightarrow{\tau}$ or $P > Q \xrightarrow{\overline{s}(Q)}$, by either (P-Pass) or (P-Sync).

Concerning the second result, the most interesting case is (T-Pipe) and proceeds by applying Lemma 5 instead of Lemma 4 as shown for the previous case.

The following theorem is the main result of the paper.

**Theorem 1 (type safety).** *Suppose $P$ is well typed. Then $P \not\to_{\mathrm{ERR}}$.*

*Proof.* Suppose by contradiction that $P \rightarrow_{\mathrm{ERR}}$. This means that $P \equiv C[[P_1 \| P_2]]$, $P_1 \xrightarrow{\lambda}$, with $\lambda ::= (v) \mid (\nu \hat{v})\langle v \rangle$ and $[P_1 \| P_2] \overset{\lambda'}{\not\rightarrow}$, with $\lambda' ::= \tau \mid \overline{s}(R)$.

Given that $P$ is well typed, by induction on $C[\cdot]$ we can prove that $[P_1 \| P_2]$ is well-typed too, hence there are suitable $\mathsf{S}, \mathsf{T}$ and $\mathsf{U}$ such that $P_1 : [\mathsf{S}]\mathsf{T}$, $P_2 : [\mathsf{U}]0$ and $\mathsf{S} \propto \mathsf{U}$.

Now, by $P_1 \xrightarrow{\lambda}$, for some $\lambda$, and by Proposition 1 (subject reduction), we deduce that there is a suitable $\alpha ::= ?\mathsf{b} \mid !\mathsf{b}$ such that $\mathsf{S} \xrightarrow{\alpha}$. Hence $I(\mathsf{S}) \neq \emptyset$. By definition of $\propto$, we get $I(\mathsf{S}) \cap \overline{I(\mathsf{U})} \neq \emptyset$. That is, there is at least one $\alpha$ such that $\mathsf{S} \xrightarrow{\alpha}$ and $\mathsf{U} \xrightarrow{\overline{\alpha}}$.

Suppose $\alpha = ?\mathsf{b}$ (the case when $\alpha = !\mathsf{b}$ is similar). By Proposition 2, we have $P_1 \xrightarrow{\lambda}$ and $P_2 \xrightarrow{\lambda'}$, with $\lambda ::= (v) \mid \tau \mid \overline{s}(Q)$ and $\lambda' ::= (\nu \hat{v})\langle v \rangle \mid \tau \mid \overline{s}(Q')$, for a suitable $v : \mathsf{b}$. Now, if either $\lambda$ or $\lambda'$ is a $\tau$ or a service call, we get a contradiction, because we would get a transition for $[P_1 \| P_2]$ violating $P \rightarrow_{\mathrm{ERR}}$. The only possibility we are left with is $\lambda = (v)$ and $\lambda' = (\nu \hat{v})\langle v \rangle$, but this would imply $[P_1 \| P_2] \xrightarrow{\tau}$, contradicting again $P \rightarrow_{\mathrm{ERR}}$.

**Corollary 1 (client progress).** *Suppose $P$ is well typed. Then $P$ guarantees client progress.*

*Proof.* By Proposition 1 and Theorem 1.

*Example 2.* Consider the system $Sys$ below, composed by:

- a directory of services $D$, which upon invocation offers a set of services $\tilde{s}_i$. We suppose that each service definition $s_i.P_i$ is well typed;
- a client $C$ that asks $S$ service to compute the summation of two integers and outputs its value;
- a service $S$, that, upon invocation: (1) asks $D$ for the name of an available service of type $\mathcal{S}_{sum}$, with $ob(\mathcal{S}_{sum}) = ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}$ – that is a service capable of receiving two integers and computing and outputting their sum; (2) invokes the received service and gets the result of the computation; and, (3) passes this value to its client.

$$
\begin{aligned}
D &\triangleq (\nu \tilde{s}_i)(dir.\textstyle\sum_i \langle s_i \rangle \mid \prod_{s_i} s_i.P_i)\\
C &\triangleq \overline{sum}.\langle 2 \rangle.\langle 3 \rangle.(w : \mathsf{int})\langle w \rangle^{\uparrow}\\
S &\triangleq sum.(z : \mathsf{int}).(y : \mathsf{int}).\\
&\qquad\qquad \Big(\overline{dir}.\big((x : \mathcal{S}_{sum}).\langle x \rangle^{\uparrow}\big) > (y : \mathcal{S}_{sum}).\overline{y}.\big(\langle z \rangle.\langle y \rangle.(w' : \mathsf{int}).\langle w' \rangle^{\uparrow}\big)\Big)\\
Sys &\triangleq C|(\nu\, dir)(S|D)\ .
\end{aligned}
$$

The whole system is well typed assuming $sum : ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}$ and $dir : \sum_i !\mathcal{S}_i$, with $s_i : \mathcal{S}_i$ for each $i$ ($Sys : [!\mathsf{int}]0$) and, as expected,

$$
Sys \Rightarrow\equiv [\langle 5 \rangle^{\uparrow} \| \mathbf{0}] \mid (\nu\, dir)(S|D)\ .
$$

*Example 3 (divergence).* Let us show a simple example of a process that is well typed but diverges. Let $s$ be a service with associated type $!\mathsf{b}$ and let be $Q = \overline{s}.(x : \mathsf{b}).\langle x\rangle^{\uparrow}$.

It is easy to see that $(x : \mathsf{b}).\langle x\rangle^{\uparrow} : [?\mathsf{b}]!\mathsf{b}$, (T-RET) and (T-IN), $?\mathsf{b} \propto !\mathsf{b}$ and $\Gamma \vdash \overline{s}.(x : \mathsf{b}).\langle \mathsf{b}\rangle^{\uparrow} : [!\mathsf{b}]\mathsf{0}$, (T-CALL).

Note also that by (SYNC):

$$Q|s.Q \to [((x : \mathsf{b}).\langle x\rangle^{\uparrow})|||Q]|s.Q \to [((x : \mathsf{b}).\langle x\rangle^{\uparrow})||| \left([((x : \mathsf{b}).\langle x\rangle^{\uparrow})|||Q]\,\right)]|s.Q \to \cdots$$

hence $Q \mid s.Q$ diverges.

More complex types are needed for avoiding such kind of divergences. E.g. types extended with service calls, and an extended type system for ensuring termination and livelock freedom, in the style of [9,13]. We let these extensions as future works.

## 6 Conclusion

We have presented a type system ensuring client progress for well typed $\mathsf{CaSPiS}^{-}$ processes. While capturing an interesting class of services, the system we propose suffers from an important limitation with respect the language in [2]: $\mathsf{CaSPiS}^{-}$ does not allow values produced inside a session to be returned to the service. Overcoming this limitation would imply allowing non-null effects in the body $P$ of a service definition $s.P$, at the same time labelling those effects as "potential" – as they are to be exercised only if and when $s$ is invoked.

It would also be important to account in a type-theoretic framework another for another important feature offered by the language in [2]: the possibility of explicitly closing sessions and handling the corresponding compensation actions.

Although the compliance relation we make use of already offers some flexibility on the client side, it would be interesting to extend the type system with subtyping on service protocols. This would imply, in the first place, understanding when two service protocols in CaSPiS can be considered as *conformant*, that is, equivalent from the point of view of any client. To this purpose, a good starting point is represented by the theories of [5,6], which provide notions of conformance for contracts, that is, service protocols.

## References

1. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., 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, Vienna (2006).
2. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. Technical Report, Università di Firenze (2008). Available from `http://rap.dsi.unifi.it/sensoria/`.
3. Bruni, R., Mezzina, L.G.: A deadlock free type system for a calculus of services and sessions. Submitted (2007).
4. Carpineti, S., Laneve, C.: A Basic Contract Language for Web Services. In: Sestoft, P. (eds.) ESOP 2006, LNCS, vol. 3924, pp. 197–213, Springer, Vienna (2006).

5. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Vienna (2006).
6. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: POPL 2008, pp. 261–272, ACM Press, San Francisco (2008).
7. Christensen, S., Hirshfeld, Y., Moller, F.: Bisimulation equivalence is decidable for basic parallel processes. In: Best, E. (eds) CONCUR 1993. LNCS, vol. 715, pp. 143–157, Springer, Hildesheim (1993).
8. Cook, W.R, Misra, J.: Computation Orchestration: A Basis for Wide-Area Computing. Journal of Software and Systems Modeling, 2006. `http://www.cs.utexas.edu/~wcook/projects/orc/`.
9. Deng, Y., Sangiorgi, D.: Ensuring Termination by Typability. Information and Computation 204(7), 1045–1082, 2006.
10. Gay, S., Hole, M.: Subtyping for session types in the pi-calculus. Acta Informatica 42(2), 191–225, 2005.
11. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (eds) ESOP 1998. LNCS 1381, pp. 22–138, Lisbon (1998).
12. Kobayashi, N.: A New Type System for deadlock-Free Processes. In: Baier, C., Hermanns, H. (eds) CONCUR 2006. LNCS 4137, pp. 233–247, Bonn (2006).
13. Kobayashi, N.: A type system for lock-free processes. Information and Computation 177(2), 122-159, 2002.
14. Lanese, I., Martins, F., Vasconcelos, V.T., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing. In: Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), pp. 305–314, IEEE Press, London (2007).
15. Milner, R.: The polyadic $\pi$-calculus: a tutorial.Logic and Algebra of Specification, 203–246, Springer, 1993.
16. Milner, R., Parrow, J., Walker, D.: A calculus of Mobile Processes, part I and II. Information and Computation 100, 1–40 and 41–78, 1992.
17. SENSORIA: Software Engineering for Service-Oriented Overlay Computers. FET-GC II IST-2005-16004 EU Project. `http://www.sensoria-ist.eu/`.
18. Talpin, J.P., Jouvelot, P.: The Type and Effect Discipline. Information and Computation 111(2), 245–296, 1994.
19. Vieira, H.T., Caires, L., Seco, J.C.: The Conversation Calculus: a Model of Service Oriented Computation. To appear in ESOP, 2008.