HAL open science

# Typing Progress in Communication-Centred Systems

Hugo Torres Vieira, Vasco Thudichum Vasconcelos

## To cite this version:

Hugo Torres Vieira, Vasco Thudichum Vasconcelos. Typing Progress in Communication-Centred Systems. 15th International Conference on Coordination Models and Languages (COORDINATION), Jun 2013, Florence, Italy. pp.236-250, 10.1007/978-3-642-38493-6_17 . hal-01486030

**HAL Id: hal-01486030**

**https://inria.hal.science/hal-01486030v1**

Submitted on 9 Mar 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# Typing Progress in Communication-Centred Systems

Hugo Torres Vieira and Vasco Thudichum Vasconcelos

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

**Abstract.** We present a type system for the analysis of progress in session-based communication centred systems. Our development is carried out in a minimal setting considering classic (binary) sessions, but building on and generalising previous work on progress analysis in the context of conversation types. Our contributions aim at underpinning forthcoming works on progress for session-typed systems, so as to support richer verification procedures based on a more foundational approach. Although this work does not target expressiveness, our approach already addresses challenging scenarios which are unaccounted for elsewhere in the literature, in particular systems that interleave communications on received session channels.

## 1 Introduction

In today's ever-growing cloud infrastructure of computation, communication is more and more of crucial importance. Communication is still of crucial importance even when considering multi-core machines, where processes may interact via shared memory, since such machines will inevitably have to communicate among them. It is then of vital importance to introduce mechanisms that support the development of *correct* communicating programs, given their wide dissemination, ranging from critical services, such as medical or financial, to those helping people connect with family and friends. Focusing on communication, there are at least two fundamental correctness properties one may be interested in: that interacting parties follow a given communication protocol (*fidelity*) and that the interaction does not reach a deadlock (*progress*).

Verification procedures, such as type systems, that statically ensure communicating programs enjoy the properties above can prove to be cost-effective, as they save on software maintenance by preventing bugs from the start, and can help to expedite the software development process. Several type systems have been proposed that single out programs with *correct* communication behaviour, out of which we distinguish session types, introduced by Honda et al. [8, 9]. Session types focus on ensuring *fidelity* in systems where (single-threaded) protocols are carried out between two parties, such as, e.g., a client and a server. Session types are by now widely adopted as the basis of pragmatic typing disciplines, targeting operating system design [5], middleware communication protocols [16] and distributed object-oriented programming [7], just to mention a few. Session types have also been generalised so as to consider multiparty interactions [3, 10].

Building on session types, a number of works were proposed that ensure progress (e.g., [2–4]). This has proven to be a challenging task, in particular regarding the expressiveness of the approaches, even when considering sophisticated typing mechanisms. In this paper we present a typing discipline that distils the basic ingredients necessary to prove progress, building on (classic) session types. This work is not to be viewed as an exercise on expressiveness, capturing every conceivable communication pattern: it fails to do so, even if we are able to address challenging system configurations. Instead, our (far more ambitious) goal is to introduce a foundational approach that will hopefully underpin the development of forthcoming works on progress for communication centred-programming based on session types.

At the basis of our development is the idea that a communication-centred program that enjoys progress is embodied with a natural ordering of communications (or *events*). Building on this idea, and trying to use as few ingredients as possible, we unify the notions of event and session type so as to characterise the sessions and the ordering of events in a combined and novel way. To achieve this we add to each communication in a session type an annotation that allows to identify the (abstract) communication event that is associated to the (concrete) communication action described by the session type. Then, together with a separate notion of overall ordering of events we are able to distinguish systems that enjoy fidelity and progress.

Remarkably we are able to address challenging configurations (unaccounted for elsewhere in the literature) where processes interleave communications in received session channels, for instance to communicate on other sessions or to initiate new ones. As an example, think of a (service) broker that must interleave communications with a client and a server (not to mention other local or remote resources), using (two binary) session channels that where shared (communicated) among the three parties. Our approach generalises previous work on progress in the context of conversation types [3], a session-based type system that addresses multiparty interaction. Also, differently from other session-types based approaches, our type system works directly on the (standard) $\pi$-calculus [13], exploiting notions from [3] and from a previous work on session types [17].

*Motivating examples.* We illustrate our development by visiting a couple of simple examples. Consider the system shown in Fig. 1 that specifies a basic interaction. The `Client` process creates a new name *chat* and sends it on channel *service* which is read by `Server`. The synchronisation on *service* allows the client and the server to share a private channel (*chat*) where the *session* will take place. After synchronizing on *service*, the client proceeds by receiving from channel *chat* a text message, after which, it sends another text message again on channel *chat*. The `Server` process is (continuously $\star$) waiting to receive in *service* a channel, which instantiates message parameter $y$. Upon reception, it will then send a text message on that channel, after which, to avoid waiting for the reply, it delegates the rest of the session interaction on $y$ by sending $y$ on channel *handle*. The `Slave` process is defined so as to (also continuously) receive a channel name

$$
\begin{aligned}
\texttt{Client} &\triangleq (\nu chat)service!chat.chat?s.chat!\text{“}\texttt{bye}\text{”}\\
\texttt{Server} &\triangleq \star service?y.y!\text{“}\texttt{hello}\text{”}.handle!y\\
\texttt{Slave} &\triangleq \star handle?z.z?s\\
\texttt{System} &\triangleq \texttt{Client} \mid \texttt{Server} \mid \texttt{Slave}
\end{aligned}
$$

**Fig. 1.** Greeting service code

from *handle*, and then receiving a text message in that channel. The `System` is defined as the parallel composition of the three processes.

Notice we described the system using type information (e.g., “text message”), taking *fidelity* for granted. Also, in the examples we use basic types (such as string) although our technical development focuses exclusively on *channel* types.

It is straightforward to see that the system enjoys progress, as the interaction supported by *chat* does not deadlock. However, we may already use this simple example to convey some intuition on our typing approach. Consider for instance process `Slave` that receives a channel name from *handle* and then communicates in the received channel. We may characterise this usage of channel *handle* with type $?\,(\,?\ \texttt{String}\,)$, where ? specifies reception, which may be read as “receives a channel used to receive a string”, just like in a regular session type.

We distinguish (session) interactions carried out exactly once (*linear*, no races) and service definitions that support several interactions (*shared* or unrestricted) by annotating the type of *handle* accordingly, i.e., $\mathsf{un}\,?\,(\,\mathsf{lin}\,?\ \texttt{String}\,)$, which adds to the description that *handle* can be used zero or more times while the received channel must be used exactly once (linearly).

Up to now we are using standard session type notions to characterise the usage of a channel (see [17]). Building on the standards, we add information that allows to characterise (in an abstract way) the moment in time when the communication is to take place: the *event*. We say that the communication in *handle* corresponds to some event in time $e_1$, while the reception of the string corresponds to event $e_2$, and obtain the type $e_1\,\mathsf{un}\,?\,(e_2\,\mathsf{lin}\,?\ \texttt{String}\,)$. Notice that $e_2$ necessarily takes place *after* $e_1$ (hence, $e_1$ and $e_2$ are different events), to represent which we use $e_1 \prec e_2$ (read “$e_1$ happens before $e_2$”).

We are then able to characterise processes with both information on the channel usage and on the expected overall *ordering* of events, in particular `Slave` is charaterized by the typing assumption $handle\colon e_1\,\mathsf{un}\,?\,(e_2\,\mathsf{lin}\,?\ \texttt{String}\,)$ and ordering of events $e_1 \prec e_2$. Notice that we do not refer to $z$ in the type since $z$ is a name bound to the reception ($handle?z$). However, and crucially to our approach, we do mention the event where $z$ is involved ($e_2$), and register it both in the message type $e_2\,\mathsf{lin}\,?\ \texttt{String}$ and in the event ordering $e_1 \prec e_2$. This will allow to crosscheck whether the name sent in *handle* has a corresponding type so as to keep a sound (overall) ordering. Following the same lines we may characterise process `Server` by the following typing assumptions.

$$
\begin{aligned}
handle &: e_1\,\mathsf{un}\,!\,(e_2\,\mathsf{lin}\,?\ \texttt{String}\,)\\
service &: e_3\,\mathsf{un}\,?\,(e_4\,\mathsf{lin}\,!\ \texttt{String}\,.e_2\,\mathsf{lin}\,?\ \texttt{String}\,)
\end{aligned}
$$

Notice that `Server`'s usage of channel *handle* is *dual* to that of `Slave` (`Server` outputs ! and `Slave` receives ?). Notice also that `Server` and `Slave` agree that

$$
\begin{aligned}
\texttt{Client} &\triangleq (\nu chat)service!chat.chat?s.chat!\text{“}\texttt{bye}\text{”}\\
\texttt{Proxy} &\triangleq \star service?y.(\nu s)masterservice!s.s!y\\
\texttt{Master} &\triangleq \star masterservice?z.z?y.y!\text{“}\texttt{hello}\text{”}.handle!y\\
\texttt{Slave} &\triangleq \star handle?z.z?s\\
\texttt{System} &\triangleq \texttt{Client} \mid \texttt{Proxy} \mid \texttt{Master} \mid \texttt{Slave}
\end{aligned}
$$

**Fig. 2.** Greeting via proxy service code

*handle* is to be used unrestrictedly (un). Furthermore, both processes agree on the moment in time when the communication in *handle* will take place ($e_1$). Lastly, both processes also agree on the message type $e_2\,\mathsf{lin}\,?\,\texttt{String}$, hence Server also knows that the channel sent in *handle* is involved in $e_2$.

We also have that *service* is used as an unrestricted input, associated to event $e_3$, with message type $e_4\,\mathsf{lin}\,!\,\texttt{String}.e_2\,\mathsf{lin}\,?\,\texttt{String}$. The message type captures that the name received from *service* will be used to output a string (event $e_4$) and *after* (denoted as . like in session types) used to receive a string (event $e_2$). The last part is realized via the delegation of the session channel in *handle*, where the delegated usage is given by the message type associated to *handle*.

The ordering of events that Server expects is $e_3 \prec e_4 \prec e_1 \prec e_2$, since the process first receives from *service* ($e_3$), then in the session channel ($e_4$) and then outputs in *handle* ($e_1$), delegating the reception usage of the session channel ($e_2$) which necessarily takes places after the channel is sent (hence $e_1 \prec e_2$). Notice that Server interleaves communications in the received channel $y$ and in *handle*, addressed by our characterisation based on the fact that the ordering of events (intertwined with channel types) mentions the events associated to communicated names.

As for the characterisation of Client we have that it uses *service* with the dual usage with respect to Server (Server inputs ? while Client outputs !) and expects ordering $e_3 \prec e_4 \prec e_2$ since it first outputs in *service* ($e_3$), then sequentially inputs ($e_4$) and outputs ($e_2$) in the session channel. Notice that although the session channel (*chat*) is private to the Client, the expected ordering mentions events where the private name is involved.

Using the usage and ordering information that characterise Server and Slave we may characterise the system Server | Slave. Channel usage is sound since the two usages of shared name *handle* are dual as mentioned before. The ordering of events is sound since gathering $e_1 \prec e_2$ and $e_3 \prec e_4 \prec e_1 \prec e_2$ does not introduce cycles in the overall order. Likewise for System since adding the characterisation of the Client we have consistent usages (dual in *service*) and a sound total ordering (obtained by gathering $e_3 \prec e_4 \prec e_1 \prec e_2$ and $e_3 \prec e_4 \prec e_2$). We are then able to show that System enjoys (not only *fidelity* but also) progress.

We now consider a slight variation of the previous scenario, illustrated in Fig. 2. The Client and Slave processes are exactly the same with respect to Fig. 1. Between them we find a Proxy and a Master, where the Proxy is used as an intermediary between Client and Master. The Proxy process starts a session with the client (via *service*) and then starts another session with Master (via *masterservice*), where the latter is used just to delegate the session channel

$$
\begin{array}{rlll}
P, Q ::= & \mathbf{0} & \text{(Inaction)} & \mid\ x!y.P \quad \text{(Output)} \\
\mid & P \,|\, Q & \text{(Parallel)} & \mid\ x?y.P \quad \text{(Input)} \\
\mid & (\nu x)P & \text{(Restriction)} & \mid\ \star x?y.P \quad \text{(Replicated Input)}
\end{array}
$$

**Fig. 3.** Syntax of processes

associated to the interaction with the client (notice that the client and the proxy only interact in the *service* synchronisation). The `Master` starts a session (via *masterservice*) and receives in it the session channel used to interact with the client. After that the `Master` process starts interacting with the `Client` via the received channel (from then on just like the `Server` in the previous example).

We may also single out the `System` shown in Fig. 2 using our type system to show it enjoys progress. Notice the `Master` process interleaves communications in two received names, one in a session "initiation" and the other in a inner session delegation, which presents no further challenge to our type system since both cases are handled uniformly. Such configuration is unaccounted for in the reference works on progress for sessions [2, 4].

In the rest of the paper we present our technical development, starting by the definition of the process model, followed by the presentation of the type system and associated results. To finish we discuss related and future work.

## 2 Process Model

In this section we present the syntax and semantics of the language of processes, a fragment of the $\pi$-calculus [13, 15]. The syntax of processes is given in Fig. 3, considering an infinite set of names $\Lambda$ ($x, y, \ldots \in \Lambda$). The (so-called) static fragment of the process model is given by the inaction $\mathbf{0}$ that represents a process with no behaviour, the parallel composition of processes $P \,|\, Q$ that represents a process where $P$ and $Q$ are simultaneously (concurrently) active, or the name restriction $(\nu x)P$ that represents a process that has a "private" name $x$ ($x$ is bound in $(\nu x)P$). The dynamic (active) fragment of the language is given by the communication prefixes: $x!y.P$ represents a process that outputs name $y$ in channel $x$ and then continues as specified by $P$; $x?y.P$ represents a process that receives a name from channel $x$ and then proceeds as $P$ ($y$ is bound in $x?y.P$); $\star x?y.P$ represents a replicated input, i.e., a process able to continuously receive a name from channel $x$ and proceed as $P$.

In order to keep the setting as simple as possible, we decided not to allow specifying alternative behaviour via summation, $+$. We believe however that our development can be extended to consider summation along non-surprising lines.

The semantics of the language is defined via structural congruence and reduction relations, to define which we introduce some (standard) notions. We denote by $fn(P)$ the set of free names that occur in $P$. Also, we denote by $P \equiv_\alpha Q$ that $P$ and $Q$ are equal up to a renaming of bound names. By $P\{x \leftarrow y\}$ we present the process obtained by replacing all free occurrences of $x$ in $P$ by $y$.

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \equiv_\alpha Q \implies P \equiv Q$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad P_1 \mid (\nu x)P_2 \equiv (\nu x)(P_1 \mid P_2) \;\; (x \notin fn(P))$$

**Fig. 4.** Structural congruence

$$\frac{}{x?y.P \mid x!z.Q \to P\{y \leftarrow z\} \mid Q}\text{(R-Com)} \qquad \frac{P \to Q}{(\nu x)P \to (\nu x)Q}\text{(R-New)}$$
$$\frac{}{\star x?y.P \mid x!z.Q \to \star x?y.P \mid P\{y \leftarrow z\} \mid Q}\text{(R-Rep)} \qquad \frac{P \to P'}{P \mid Q \to P' \mid Q}\text{(R-Par)}$$
$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\text{(R-Cong)}$$

**Fig. 5.** Reduction relation

Structural congruence is defined as the least congruence over processes that satisfies the rules in Fig. 4. Structural congruence allows to specify equivalent classes of processes and supports the definition of the reduction relation focusing on the interactions of the (basic) representatives of the equivalent classes.

The reduction relation over processes is given by the rules in Fig. 5, capturing how processes "evolve" via communication. Rule (R-Com) captures the interaction between an output and an input, where the output emits a name that is received in the input (notice the communicated name $z$ replaces the bound input parameter in the continuation $P$). Rule (R-Rep) follows the same lines with respect to the communication behaviour, the difference is that in the resulting state the input is again ready to further synchronise. Aiming at a simple subject congruence result, we do not consider the (original to the pi calculus) structural congruence rule that introduces parallel copies of the replicated process, defining instead unbounded behaviour via rule (R-Rep). The remaining rules close the relation under language contexts — (R-New) for name restriction and (R-Par) for parallel composition — and under structural equivalence classes.

## 3 Type System

In this section we present our type system, starting by introducing strict partial orders, a crucial notion that allows us to single out well-formed communication dependency structures of processes. We then present our type language which unifies the notions of events and of session types descriptions, and our type system where processes are characterised via their usage of channels (as usual) and of their ordering of events. Finally, we present our results, namely typing preservation under reduction (Theorem 1) and progress (Theorem 2).

The idea of ordering events to guarantee progress is not new (cf., [2, 4, 11, 12]) and seems in fact an excellent mechanism to single out sound communica-

$$
\begin{array}{llll}
p, p_1, p_2, \ldots & ::= \; ! & \text{(Output)} & \\
 & \mid \; ? & \text{(Input)} & \\
 & \mid \; \tau & \text{(Synchronisation)} & \\
L, L_1, L_2, \ldots & ::= \; \mathsf{end} & \text{(No interaction)} & \\
 & \mid \; e \,\mathsf{lin}\, p\, T.L & \text{(Session)} &
\end{array}
\qquad
\begin{array}{lll}
T, T_1, T_2, \ldots & ::= \; L & \text{(Linear)} \\
 & \mid \; e \,\mathsf{un}\, p\, T & \text{(Shared)} \\
 & & \\
\Gamma, \Gamma_1, \Gamma_2, \ldots & ::= \; \cdot & \text{(Empty)} \\
 & \mid \; \Gamma, x : T & \text{(Assumption)}
\end{array}
$$

**Fig. 6.** Syntax of types and typing contexts

tion dependency structures. In our approach, we introduce event orderings and session types in a combined and uniform way, aiming at minimising the number of ingredients required to prove progress of communicating processes. We thus find the strict partial orders defined next at the root of our development.

*Strict partial orders.* A strict (or irreflexive) partial order $\prec$ over a set $\mathcal{E}$ is a binary relation that is asymmetric (hence irreflexive) and transitive. We call $\mathcal{E}$ the *set of events*, and we let $e, e_1, e_2, \ldots$ range over $\mathcal{E}$. Furthermore we distinguish two events, and call them $\mathsf{end}$ and $\top$, that form the "leaves" of the communication dependency tree (since strict partial orders do not admit reflexive pairs, we require two elements to form the "leaf" pair of the relation).

We make use of the following notions on partial orders. The *support* of a partial order $\prec$ is the set of elements of $\mathcal{E}$ that occur in $\prec$, defined as follows.

$$\mathit{supp}(\prec) \;\triangleq\; \{e \mid \exists e_1.(e_1, e) \in \prec \vee (e, e_1) \in \prec\}$$

The support is used in our typing rules so as to allow us to pick "fresh" events, i.e., events that are not referred to by the relation. The relation obtained by *adding the least event* $e$ to $\prec$ is denoted by $e + \prec$. Notice that if $e$ is not in the support of $\prec$ and $\prec$ is a strict partial order, then so is $e + \prec$.

$$e + \prec \;\triangleq\; \prec \cup \{(e, e_1) \mid e_1 \in \mathit{supp}(\prec)\}$$

The relation obtained by *removing an element* $e$ from $\prec$ is denoted by $\prec \backslash\, e$. Notice that if $\prec$ is a strict partial order, then so is $\prec \backslash\, e$.

$$\prec \backslash\, e \;\triangleq\; \{(e_1, e_2) \mid (e_1, e_2) \in \prec \wedge e \neq e_1 \wedge e_2 \neq e\}$$

The strict partial order $\prec$ relation obtained by the *union of two strict partial orders* $\prec_1$ and $\prec_2$ is denoted by $\prec_1 \uplus \prec_2$. Notice that $\uplus$ is a partial operation (undefined when the plain union of the relations introduces cycles). We use $\uplus$ to gather the communication dependency structures of, e.g., two parallel processes.

*Types.* Having defined the notion of strict partial orders of events we proceed to the presentation of the type language whose syntax is given in Fig. 6. Our types extend session types [8, 9] with event annotations, and also exploit notions introduced in previous work on session types [17] and conversation types [3].

We use *polarities*, $p$, to capture communication capabilities: ! captures the output capability, ? captures the input capability and $\tau$ captures a synchronisation pair (cf., [3]). Types are divided in two main classes, shared and linear. The

former captures the exponential usage of channels, i.e., channels where (communication) races are admissible (intuitively, think of services that may be simultaneously provided and requested by several sites). The latter captures linear usage of channels, where no races are allowed (in the service analogy, the single-threaded protocol between server and client in a service instance). A shared type $e\,\mathsf{un}\,p\,T$ specifies a polarity $p$ that captures a communication capability, an event $e$ so as to create the association between the communication action and the abstract notion of event, and an argument type $T$ that prescribes the delegated behaviour of the communicated channel. The description of a linear type $e\,\mathsf{lin}\,p\,T.L$ follows the same lines, except for the continuation $L$ which specifies the behaviour that takes place after the action captured by $e\,\mathsf{lin}\,p\,T$. Linear types are terminated in $\mathsf{end}$, meaning no further interaction is allowed on the channel.

Notice that our types build on the notion of abstract events, differently from other related approaches (cf., [2–4]) that resort to channel names (and communication labels) to order communication actions. Notice also our types structurally resemble "classic" session types, differing in the introduction of the event identifier, crucial to our approach, the polarity annotation that allows us to avoid polarised channels, and the linear/unrestricted annotation that allows us to avoid separate typing contexts for shared and linear channels and separate typing rules for linear/unrestricted argument type of communications.

We next define some auxiliary notions over types used in our typing rules. The set of elements of $\mathcal{E}$ that occur in a type $T$ is denoted by $\mathit{events}(T)$. Notice that events in messages (argument types) are not included.

$$\mathit{events}(T) \triangleq \begin{cases} e \cup \mathit{events}(L) & \text{if } T = e\,\mathsf{lin}\,p\,T_1.L \\ \{\mathsf{end}\} & \text{if } T = \mathsf{end} \\ \{e\} & \text{if } T = e\,\mathsf{un}\,p\,T_1 \end{cases}$$

The binary relation over $\mathcal{E}$ present in a type $T$ is denoted by $T\downarrow$. If each linear prefix in $T$ has a distinct event $e$ then it is immediate that $T\downarrow$ is a strict partial order. We use $T\downarrow$ to single out the order of events prescribed by a type, which essentially is a (single) chain of events in the case that $T$ is a linear type.

$$T\downarrow \triangleq \begin{cases} e + (L\downarrow) & \text{if } T = e\,\mathsf{lin}\,p\,T_1.L \\ \{(\mathsf{end}, \top)\} & \text{if } T = \mathsf{end} \\ \{(e, \top)\} & \text{if } T = e\,\mathsf{un}\,p\,T_1 \end{cases}$$

We introduce a predicate that is true for types that do not specify pending communication actions.

$$\mathit{matched}(T) \triangleq \begin{cases} \mathit{matched}(L) & \text{if } T = e\,\mathsf{lin}\,\tau\,T_1.L \\ \mathsf{true} & \text{if } T = \mathsf{end} \text{ or } T = e\,\mathsf{un}\,?\,T_1 \\ \mathsf{false} & \mathit{otherwise} \end{cases}$$

Matched linear communication actions are captured by $\tau$ annotated types. As for shared communication actions, we focus only on unmatched output actions and thus only shared inputs are matched. We will clarify this notion in the definition of splitting (Fig. 8) and in the characterisation of active processes in the context of our main result (Theorem 2), for now it suffices to say that matched shared communications are ?-annotated.

*Typing contexts.* The syntax of typing contexts is given in Fig. 6. We assume by convention that, in a typing context $\Gamma, x\colon T$, name $x$ does not occur in $\Gamma$. Also, we use $\Gamma_{\mathsf{end}}$ to abbreviate a typing context $\cdot, x_1\colon \mathsf{end}, \ldots, x_k\colon \mathsf{end}$ for some $k \geq 0$ and $x_1, \ldots, x_k$. We introduce some auxiliary predicates over typing contexts to single out typing contexts that refer only to unrestricted and matched communications.

We denote by $\Gamma_{un}$ contexts that include only shared communications, defined as $\Gamma_{un} ::= \cdot \mid \Gamma'_{un}, x\colon \mathsf{end} \mid \Gamma'_{un}, x\colon e\,\mathsf{un}\,!\,T$. Such contexts are used to qualify the exponential resources that a replicated input may use. Since more than one copy of the continuation of a replicated input may be simultaneously active, there must be no linear behaviour present (to avoid communication races). We also exclude shared inputs in $\Gamma_{un}$ so as to avoid nested replicated inputs. Intuitively, if we admit nested service definitions then, to guarantee progress, we would also require that every service is called at least once (in such way activating all nested service definitions) or characterise progress of open systems by inserting them in the "right" context (cf., [4]). We focus on closed systems where all communications are matched, i.e., typed in *matched* contexts. We lift the matched predicate over types to typing contexts in the expected way: we write $\mathit{matched}(\cdot, x_1\colon T_1, \ldots, x_k\colon T_k)$ if $\mathit{matched}(T_i)$ for all $i$ such that $1 \leq i \leq k$.

*Splitting and conformance.* We now introduce two notions crucial to our development, namely *splitting* (inspired by [1]) that explains how behaviour can be decomposed and safely distributed to distinct parts of a process (e.g., to the branches of a parallel composition), and *conformance* that captures the desired relation between typing contexts and strict partial orders.

We say a typing context $\Gamma$ conforms to a strict partial order $\prec$, denoted by $\mathit{conforms}(\Gamma, \prec)$, if all event orderings prescribed by the types in $\Gamma$ are contained in $\prec$, thus ensuring that the events associated with the communication actions described by the types are part of the overall ordering.

$$\mathit{conforms}(\Gamma, \prec) \;\triangleq\; \begin{cases} \mathsf{true} & \text{if } \Gamma = \cdot \\ \mathit{conforms}(\Gamma_1, \prec) & \text{if } \Gamma = \Gamma_1, x\colon T \text{ and } T\!\downarrow\!\subseteq\!\prec \\ \mathsf{false} & \mathit{otherwise} \end{cases}$$

Splitting is defined for both types and typing contexts, defined via three operations over linear types, shared types and typing contexts. We write $T = T_1 \circ T_2$ to mean that type $T$ is split in types $T_1$ and $T_2$, and likewise for $\Gamma = \Gamma_1 \circ \Gamma_2$. Linear type splitting, shared type splitting and context splitting are given by the rules in Figs. 7–9. Linear type splitting supports the decomposition of a synchronised, $\tau$, session type (including continuation) in the respective dual capabilities $!, ?$, via rule (L-Dual-1) and its symmetric (L-Dual-2). Notice $L = L_1 \circ L_2$ is defined only when $\mathit{matched}(L)$. Essentially, linear type splitting allows to decompose a matched session type in its two *dual* counterparts (see, e.g., [6]).

Shared type splitting decomposes shared communication capabilities in two distinct ways, depending on whether the polarity of the type to be split is ? or !. A shared input is split in a shared input and either in an output or another input, via rules (S-In-1) and its symmetric (S-In-2). Intuitively, an input that is decomposed in two inputs allows to type processes that separately offer the

$$\frac{}{\mathsf{end} = \mathsf{end} \circ \mathsf{end}}\,(\text{L-End})$$

$$\frac{L = L_1 \circ L_2}{e\,\mathsf{lin}\,\tau\,T.L = e\,\mathsf{lin}\,!\,T.L_1 \circ e\,\mathsf{lin}\,?\,T.L_2}\,(\text{L-Dual-1})$$

$$\frac{L = L_1 \circ L_2}{e\,\mathsf{lin}\,\tau\,T.L = e\,\mathsf{lin}\,?\,T.L_1 \circ e\,\mathsf{lin}\,!\,T.L_2}\,(\text{L-Dual-2})$$

**Fig. 7.** Linear type splitting

$$\frac{p \in \{?, !\}}{e\,\mathsf{un}\,?\,T = e\,\mathsf{un}\,?\,T \circ e\,\mathsf{un}\,p\,T}\,(\text{S-In-1})$$

$$\frac{p \in \{?, !\}}{e\,\mathsf{un}\,?\,T = e\,\mathsf{un}\,p\,T \circ e\,\mathsf{un}\,?\,T}\,(\text{S-In-2})$$

$$\frac{}{e\,\mathsf{un}\,!\,T = e\,\mathsf{un}\,!\,T \circ e\,\mathsf{un}\,!\,T}\,(\text{S-Out})$$

**Fig. 8.** Shared type splitting

$$\frac{}{\cdot = \cdot \circ \cdot}\,(\text{C-Empty})$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon T = \Gamma_1, x\colon T \circ \Gamma_2}\,(\text{C-Left})$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon T = \Gamma_1 \circ \Gamma_2, x\colon T}\,(\text{C-Right})$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = T_1 \circ T_2}{\Gamma, x\colon T = \Gamma_1, x\colon T_1 \circ \Gamma_2, x\colon T_2}\,(\text{C-Split})$$

**Fig. 9.** Context splitting

input capability (e.g., a service that is provided by two distinct sites), and an input that is decomposed in an output and an input allows to type processes that offer the dual communication capabilities (e.g., a service provider and a service client). A shared output is split in two shared outputs — rule (S-Out) — which, intuitively, allows to type processes that offer the output capability separately (e.g., like two service clients). Input capabilities may be further split so as to "absorb" several output capabilities and be distributed in several input capabilities, and output capabilities may also be further split to be distributed in several output capabilities. Notice type splitting (both linear and shared) preserves the argument types so as to guarantee the dual communication actions agree on the type of the communication.

Context splitting allows to split a context in two distinct ways: context entries either go into the left or the right outgoing contexts — rules (C-Left) and its symmetric (C-Right) — or they go in both contexts — rule (C-Split). The latter form lifts the (type) behaviour distribution to the context level, while the former allows to delegate the entire behaviour to a part of the process, leaving no behaviour to the other part. To lighten notation we use $\Gamma_1 \circ \Gamma_2$ to represent $\Gamma$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$ (if such $\Gamma$ exists). Notice that, given $\Gamma_1$ and $\Gamma_2$, there is at most one $\Gamma$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$.

*Typing system.* We may now present our type system which characterises processes in terms of their usage of channels and of their overall ordering of events, as captured by judgement $\Gamma; \prec \vdash P$ where $\Gamma$ describes channel usage and $\prec$ gives the ordering of events. We say process $P$ is well-typed if $\Gamma; \prec \vdash P$ is the conclusion of a derivation using the rules in Fig. 10.

$$\frac{}{\Gamma_{\mathsf{end}}; \{(\mathsf{end}, \top)\} \vdash \mathbf{0}} \quad \text{(T-Inact)}$$

$$\frac{\Gamma_1; \prec_1 \vdash P \qquad \Gamma_2; \prec_2 \vdash Q}{\Gamma_1 \circ \Gamma_2, \prec_1 \uplus \prec_2 \vdash P \,|\, Q} \quad \text{(T-Par)}$$

$$\frac{\Gamma, x\colon T; \prec \vdash P \qquad \mathit{matched}(T)}{\Gamma; \prec \vdash (\nu x)P} \quad \text{(T-New)}$$

$$\frac{\Gamma, x\colon L, y\colon T; \prec \vdash P \qquad e \notin \mathit{supp}(\prec)}{\Gamma, x\colon e\,\mathsf{lin}\,?\,T.L; e + \prec \vdash x?y.P} \quad \text{(T-LIn)}$$

$$\frac{\Gamma, x\colon L; \prec \vdash P \qquad e \notin (\mathit{supp}(\prec) \cup \mathit{events}(T))}{(\Gamma, x\colon e\,\mathsf{lin}\,!\,T.L) \circ y\colon T; e + (\prec \uplus T\downarrow) \vdash x!y.P} \quad \text{(T-LOut)}$$

$$\frac{\Gamma_{un}, y\colon T; \prec \vdash P \qquad e \notin \mathit{supp}(\prec)}{\Gamma_{un}, x\colon e\,\mathsf{un}\,?\,T; e + \prec \vdash \star x?y.P} \quad \text{(T-UIn)}$$

$$\frac{\Gamma; \prec \vdash P \qquad e \notin (\mathit{supp}(\prec) \cup \mathit{events}(T))}{(\Gamma, x\colon e\,\mathsf{un}\,!\,T) \circ y\colon T; e + (\prec \uplus T\downarrow) \vdash x!y.P} \quad \text{(T-UOut)}$$

**Fig. 10.** Typing rules

We comment on the rules in Fig. 10. Rule (T-Inact) types the inactive process with a context that associates $\mathsf{end}$ to (any set) of channel names and with the "leaf" ordering (the relation with just one pair $(\mathsf{end}, \top)$). Rule (T-Par) types parallel composition by typing each branch with a slice of the context, obtained via splitting, and with a sub-ordering (such that the union of the sub-orderings is a strict partial order). So, the two branches of the parallel composition may freely refer different channels but they must agree in a sound overall ordering. Rule (T-New) types name restriction by typing the restricted name with a matched type (no unmatched communications). Notice the ordering expected for the interactions in the restricted name is kept in the conclusion, so as to characterise the abstract communication dependencies of the process, which includes (an abstraction of) the communication dependencies of bound names.

Communication prefixes are typed in separate rules depending on the type of the subject of the communication — notice however that mixing linear and shared types in the same typing context avoids introducing rules that depend on the type of the object of the communication. Rule (T-LIn) types the input on a channel $x$ with linear usage by typing the continuation process considering the continuation session type $L$ for $x$, the argument type $T$ for the input variable $y$ and ordering $\prec$. We single out a fresh event $e$ with respect to the continuation ($e \notin \mathit{supp}(\prec)$) that is used to specify the type of the input, together with the respective ? polarity, argument type $T$ and continuation $L$. We build a new order from $\prec$ by setting $e$ as the least element (given by $e+\prec$) since any communication in the continuation depends on the input (hence is greater than $e$). Notice that the communications in the continuation include the ones that involve the received name, characterised by $T$ and ordered by $\prec$. Notice also that $\prec$ is recorded in the conclusion, so as to (also) capture the communication dependencies involving

the received name. This is crucial to our approach so as to address processes that interleave communications in received names.

Rule (T-LOut) types the output in a channel $x$ with linear usage by typing the continuation process with the continuation session type $L$ and ordering $\prec$. The conclusion records the type of the output using a fresh event $e$ with respect to the continuation (ordered by $\prec$) and also with respect to the type delegated in the communication $T$ ($e \notin \mathit{supp}(\prec) \cup \mathit{events}(T)$). The (linear) session type in the conclusion is thus specified using $e$, the argument type $T$, the respective ! polarity and continuation $L$. Event $e$ is also recorded in the ordering of the output as the minimum event $e + (\prec \uplus T\downarrow)$, since any communication in the continuation, along with any delegated communication capabilities, depends on the output (hence, are greater than $e$). We use $T\downarrow$ to extract the (chain of) events prescribed by $T$. The conclusion records the delegated type $T$ via splitting $(\Gamma, x\colon e\,\mathsf{lin}\,!\,T.L) \circ (y\colon T)$ as $y$ may be used (dually to $T$) in $\Gamma$.

The description of rule (T-UOut) follows the same lines, the only differences is that a shared type captures the output and there is no continuation usage for channel $x$. We rule out uses of $x$ in the continuation to exclude processes that offer the input in the continuation of an output (at the cost of excluding processes that perform more than one shared output over the same channel in sequence). Our rationale for shared communications is that at least one (replicated) shared input matches all corresponding outputs, so the input cannot be activated *after* the output (to avoid cluttering the rules this led us to also exclude two outputs in sequence, a configuration which is not problematic per se). Rule (T-UIn) types shared inputs that are necessarily replicated, so as to support the rationale that a shared input is able to match all respective inputs. Since more than one copy of the continuation of the input may be simultaneously active we require the resources *shared* by all copies to be shared outputs ($\Gamma_{un}$).

The reason why we exclude (nested) shared inputs, as explained earlier, is to avoid the situation where a shared output is blocked due to a shared input (of lesser order) that is blocking the matching shared input (of greater order) which is not matched. To avoid forcing that all shared inputs are matched we exclude nested shared inputs in general. Notice however that the argument type $T$ may be linear or shared, in which case $T$ may actually specify a shared input (a nested input that is activated via interaction). Since the behaviour of the name received in the input is only "published" via a corresponding output, this particular case of nested shared inputs is naturally supported.

The main restriction of the presented work is the absence of a general form of recursion, which, conceivably, involves considering the repetition of the overall ordering throughout the unfolding, handled e.g., via a dedicated typing context that we believe can be engineered in conformance to our approach.

*Results.* We may now present our results, namely that typing is preserved under reduction (Theorem 1) and that a specific class of well-typed processes (those where all communications are matched) enjoy progress (Theorem 2). We start by mentioning some auxiliary results, namely that we may show that conformance is ensured between the typing context and the strict partial order in all derivations,

a sanity check that ensures the conditions imposed by our rules (e.g., picking freshness of events) are enough to keep conformance invariant in a derivation. We may also show two standard results used in the proof of Theorem 1, namely that typing is preserved under structural equivalence and under name substitution.

Before presenting our first main result (Theorem 1) we introduce two auxiliary notions that characterise reduction of contexts and of strict partial orders. As expected from a behavioural type system, as processes evolve so must the types that characterise the processes. Reduction for contexts is defined as follows.

$$\cdot \to \cdot \qquad \Gamma, x\colon e\,\mathsf{lin}\,p\,T.L \to \Gamma, x\colon L \qquad \Gamma_1 \to \Gamma_2 \implies \Gamma_1, x\colon T \to \Gamma_2, x\colon T$$

A context reduces if it holds an assumption on a linear type prefix, which reduces to the continuation so as to mimic the analogous behaviour in processes. Also, the empty context reduces (to the empty context) so as to capture synchronisations in processes on restricted channels and on channels with shared usage, thus introducing reflexivity in context reduction since no change is required to capture such synchronisations. Reduction for partial orders is defined as follows.

$$\prec \to \prec \qquad e \in \mathit{supp}(\prec) \implies \prec \to \prec \setminus e$$

Strict partial order reduction is also reflexive. This allows to capture synchronisations that *depend* on shared inputs. Notice that the ordering for shared inputs is kept invariant via reduction (since the replicated process is kept throughout reduction), thus capturing synchronisations that depend on shared inputs (as they will take place repeatedly for each activation of the continuation of the shared input). Reduction is also defined by removing an event of the ordering, so as to capture *one shot* synchronisations. Since such synchronisations may depend on shared outputs, they are not necessarily associated with the minimum event in the ordering. We may now present our first main result, where $\Gamma_1; \prec_1 \to \Gamma_2; \prec_2$ denotes $\Gamma_1 \to \Gamma_2$ and $\prec_1 \to \prec_2$.

**Theorem 1 (Preservation).** *If $\Gamma_1; \prec_1 \vdash P_1$ and $P_1 \to P_2$ then $\Gamma_1; \prec_1 \to \Gamma_2; \prec_2$ and $\Gamma_2; \prec_2 \vdash P_2$.*

The proof follows by induction on the length of the derivation of $P_1 \to P_2$ in expected lines. The theorem says that typing is preserved under process reduction, up to a reduction in the context and ordering. Fidelity is an immediate consequence of Theorem 1, as usual. We now turn our attention to the result on progress. In order to define "live" processes (processes that should reduce) we introduce the (standard) notion of *active contexts*, noted $\mathcal{C}[\cdot]$, defined as $\mathcal{C}[\cdot] ::= \cdot \mid (P \mid \mathcal{C}[\cdot]) \mid (\nu x)\mathcal{C}[\cdot]$. A context $\mathcal{C}[\cdot]$ is a process with a hole $\cdot$ under a number of parallel compositions and restrictions. We say a process $P$ is *active* if it has a (non-replicated) communication prefix in an active context, defined as follows $\mathit{active}(P) \triangleq \exists \mathcal{C}[\cdot], x, y, Q.\ P \equiv \mathcal{C}[x!y.Q] \vee P \equiv \mathcal{C}[x?y.Q]$. Notice the definition of active process rules out replicated inputs. So, we consider *stable* processes (processes that do not reduce but are not errors) to be processes where a number of (replicated) shared inputs are active. We now state our second main result that says an active and well-typed (*matched*) process reduces.

**Theorem 2 (Progress).** *If $\mathit{active}(P)$, $\Gamma; \prec \vdash P$ and $\mathit{matched}(\Gamma)$ then $P \to P'$.*

The proof follows by induction on the size of $\prec$. The proof invariant is that for every event either there is a synchronisation pair of *lesser* order or every active prefix of lesser order is a replicated (shared) input. Theorem 2 attests our typing discipline ensures progress of active processes, including processes that interleave communications on received channels.

## 4 Concluding Remarks

We have presented a typing discipline for the analysis of progress in session-based communication-centred systems. Our work exploits notions introduced in [3] (e.g., the $\tau$ polarity) and in [1] (e.g., the splitting relation), allowing to type systems specified in standard $\pi$-calculus. This is in contrast with related approaches, where session channels are equipped with polarities (see, e.g., [6]) or where channels have two endpoints (see, e.g., [17]), or or where sessions are established via specialised initiation primitives (see, e.g., [9]). Also, we uniformly handle communication *of* linear and shared channels (when they are the object of the communication) via $\mathsf{lin}$ and $\mathsf{un}$ annotations introduced in [17]. However, this is not the case for communications *on* linear and shared channels (when they are the subject of the communication).

A cornerstone of our development is the progress analysis technique introduced in [3], where message types already specify the orderings expected for the communicated names, thus providing the basic support for the interleaving of communications on received names. We depart from [3] by unifying the channel usage and event ordering in a single type analysis. Moreover, our orderings build on abstract events and do not refer channel identities (nor labels) differently from [2–4], which allows us to relate events in received names (via an abstract event) with others. This is crucial to address the interleaving of received names in a more general way, allowing us to address scenarios out of reach of the above mentioned works [2–4]. By combining session types and events in the same type language, inspired by [14], we are able to rely on usual session-based reasoning. Our approach differs from the preliminary ideas presented in [14] that combines session types with a typing discipline that relies on type simulation [11], in that our verification system is completely syntax driven and does not rely on extra-imposed conditions (neither on type simulation nor on model-checking).

In our approach, the sound communication dependency structure is captured in a minimal way, via a (strict) partial order of events, which combined with the event-equipped session types, allow us to single out systems that enjoy progress. We acknowledge that our development does not address full-fledged recursion. However, the principles we use can conceivably be lifted to consider recursion, considering the repetition of the event ordering (handled by a dedicated typing context, as usual) or (well-founded) infinite orderings, an engineering exercise we leave to future work. We also plan to use the ideas presented in this paper to type progress in multiparty conversations.

On a pragmatic (vital) level, we may show that the type checking procedure is decidable (considering bound names are type annotated) and we are confident

that a type inference procedure can be extracted from our type system. While decidability attests the type system is worth mentioning, type inference makes it more interesting. It supports the verification of systems without burdening the development process, thus contributing to a cost-effective increase of reliability.

**Acknowledgments** We acknowledge support of the project PTDC/EIA-CCO/117513/2010 and thank Pedro Baltazar and Luís Caires for fruitful discussions.

## References

1. Baltazar, P., Caires, L., Vasconcelos, V.T., Vieira, H.T.: A type system for flexible role assignment in multiparty communicating systems. In: TGC 2012, Proceedings. LNCS, Springer (2012), to appear.
2. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: CONCUR 2008, Proceedings. LNCS, vol. 5201, pp. 418–433. Springer (2008)
3. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. 411(51-52), 4399–4440 (2010)
4. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: TGC 2007, Proceedings. LNCS, vol. 4912, pp. 257–275. Springer (2007)
5. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. In: EuroSys 2006, Proceedings. pp. 177–190. ACM (2006)
6. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. 42(2-3), 191–225 (2005)
7. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: POPL 2010, Proceedings. pp. 299–312. ACM (2010)
8. Honda, K.: Types for dyadic interaction. In: CONCUR 1993, Proceedings. LNCS, vol. 715, pp. 509–523. Springer (1993)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP 1998, Proceedings. LNCS, vol. 1381, pp. 122–138. Springer (1998)
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, Proceedings. pp. 273–284. ACM (2008)
11. Kobayashi, N.: A type system for lock-free processes. Inf. Comput. 177(2), 122–159 (2002)
12. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: STOC 1980, Proceedings. pp. 70–81. ACM (1980)
13. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I and II. Inf. Comput. 100(1), 1–77 (1992)
14. Padovani, L.: From lock freedom to progress using session types. In: PLACES 2013, Proceedings (2013), to appear.
15. Sangiorgi, D., Walker, D.: The $\pi$-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
16. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the behavior of software components using session types. Fundam. Inform. 73(4), 583–598 (2006)
17. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. 217, 52–70 (2012)