Sessions and Session Types: an Overview

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro

Dipartimento di Informatica, Università di Torino
corso Svizzera 185, 10149 Torino (Italy)
{dezani,deliguoro}@di.unito.it

Abstract. We illustrate the concepts of sessions and session types as they have been developed in the setting of the π-calculus. Motivated by the goal of obtaining a formalisation closer to existing standards and aiming at their enhancement and strengthening, several extensions of the original core system have been proposed, which we survey together with the embodying of sessions into functional and object-oriented languages, as well as some implementations.

Key words: Process calculi, Type Systems, Service Oriented Computing.

1 Introduction

The rapid growth of web technologies and of service oriented programming is promoting a fruitful interaction between research communities and standards organizations, with the aim of designing languages and systems for communication centred computations based on a sound theoretical footing.

Session types are one of the formalisms that have been proposed to structure interaction and reason over communicating processes and their behaviour. They appeared in [THK94] and subsequently in [HVK98], where the issue of formalising in a type system the concept of session was framed in the (polyadic) π-calculus with types. The basic idea is to introduce a new form of polymorphism which allows the typing of channel names by structured sequences of types, abstractly representing the trace of the usage of the channels.

The apparently weak constraint constituted by typing channels with session types, while disregarding the interleaved usage of the channels themselves within the process term, is however sufficient to detect subtle errors in the implementation of communication protocols. In fact it reveals to be the right setting where concepts developed for the π-calculus or in general for process algebras can be combined: we think of error freeness checked via typability, of internal mobility which nicely captures the idea of private conversations, of linearity and type duality which enforce the mirroring of the channel usage into its type, and of channel transmission, at the very basis of the π-calculus, to model service delegation.

Since then a substantial body of research has been carried out: to better understand the potentiality of the proposed calculi, as it is the case of the introduction of subtyping polymorphism for session types in [GH05]; to strengthen the expressive power of session type systems with respect to relevant computational properties like progress and deadlock-freedom in [DCdLY08], building over ideas of [Kob06]; to widen the scenarios which can be modelled in the calculus, stepping to multiparty sessions instead of just dyadic ones [HYC08], or to detect realizable choreographies via a system of global session types in [CHY07]; to propagate the session type technology to existing programming languages as in [VGR06] for the functional paradigm or in [CCDC+09] for the object-oriented one, providing implementations and applications.

Session types are by no means the only proposal for a theoretical foundation of communication centred programming which has been based on process algebras. Service calculi as well as protocol descriptions called "contracts" have been devised (for which see the references in Section 6) and in some cases the relations with session types have been investigated, although much remains to be done. The comparisons of the superficially different formalisms enlightening common underlying concepts will hopefully improve the language design and the programming practice for communication based computing.

In the present paper we will survey all these aspects mainly informally, by means of examples or just providing pointers to the literature. We begin in Section 2 with session types in their global versus local formulation, though this is a recent development: this is where the basic concepts and formalisms are presented. Section 3 overviews the numerous extensions for the original system which have been proposed to gain expressivity and to catch stronger computational properties. Section 4 is devoted to the embedding of sessions and their typings into the functional and object-oriented programming paradigms. In Section 5 we report on implementations of sessions and session types which use mainstream programming languages. Finally in Section 6, we quickly review formalisms and calculi which appear to be close to session type systems and to their goals.

2 Basic Concepts and Systems

In networking a *session* is a logic unit of information exchange between two or more communicating agents. The essential concern of a session is to specify the topic of conversation as well as the sequence and direction of the communicated messages. This has been formalized as a type system for a dialect of Milner's π-calculus in a series of papers by Honda and others [THK94,HVK98,YV07], and recently extended to express ideas from W3C-CDL (`http://www.w3.org/TR/ws-cdl-10/`), a language for choreography. To look at sessions and session types in their latest incarnation, we follow [CHY07,HYC08], where sessions are described at different levels. At the global level they are abstract specifications of globally available services (called *interactions* in [CHY07]), whose types are global session types, or simply *global types*. At the local level they are protocols

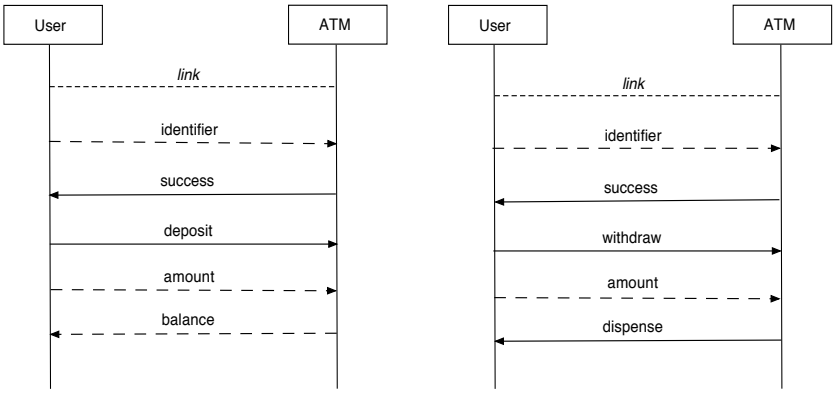

Fig. 1. UML sequence diagrams of some User-ATM interactions.

described in the participant perspective: the local session types or just *session types*, which can be assigned to *end-point* processes, the actual participants of the interaction. These two levels are related to each other: the global processes (that can be thought of as choreographies) and the global types should project to the local ones, where end-points play the role of the actual implementations of the specified system.

To illustrate these concepts and their formal representation, let us consider the following protocol which describes a simplified interaction between a customer (User) and an automated teller machine (ATM)[1]:

- First the User communicates her/his identifier to the ATM;
- The ATM answers with either success or failure.
  - In the first case the User can ask either for doing a deposit or a withdraw.
    - For a deposit the User communicates an amount, and waits for a balance.
    - For a withdraw first the User communicates an amount and then the ATM answers with either dispense or overdraft.
  - If the answer is failure, then the interaction terminates.

Two possible interactions are described in the UML sequence diagrams in Figure 1. Note that identifier, amount and balance are row data and have been represented by dashed arrows, while success, failure, deposit, withdraw, dispense and overdraft are labels used to choose between different options, shown in the diagrams by solid arrows.

[1] The example of the interaction among User, ATM and Bank comes from [HVK98], and it has been used by several authors. We adapt this example also to illustrate the subsequent developments and variations of the original system.

Following [CHY07][2] a global description of this interaction is as follows:

```
User ⟶ ATM : identifier.
ATM ⟶ User :
    { success : User ⟶ ATM :
                { deposit :    User ⟶ ATM : amount.
                               ATM ⟶ User : balance.
                               end
                [] withdraw :  User ⟶ ATM : amont.
                               ATM ⟶ User :
                                   { dispense : end
                                   [] overdraft : end
                                   }
                }
    [] failure : end
    }
```
(1)

The arrows User ⟶ ATM and ATM ⟶ User represent the direction of the message, which in the first line is simply an identifier. The alternatives between the possible answers success or failure by the ATM (and similarly in the subsequent lines, where branching actions are described) are grouped by curly brackets and separated by []. In the last case the protocol terminates, while in the first one it goes on with nested choices, by choosing among deposit and withdraw. In the first case the User is expected to send the amount and to wait for the balance from the ATM. In the case of withdraw instead, after sending the amount the User will receive either a dispense or an overdraft message from the ATM.

The global type of the current interaction can be simply obtained from the global description replacing data by their types:

```
User ⟶ ATM : String.
ATM ⟶ User :
    { success : User ⟶ ATM :
                { deposit :    User ⟶ ATM : Real.
                               ATM ⟶ User : Real.
                               end
                [] withdraw :  User ⟶ ATM : Real.
                               ATM ⟶ User :
                                   {dispense : end
                                   [] overdraft : end
                                   }
                }
     [] failure : end
    }
```
(2)

[2] In [CHY07,HYC08] global descriptions of interaction are more informative than ours, since they also specify the initiation and the channel names on which data and choice labels are communicated.

Now let us look at the User view of this interaction, which can be described by the *session type*:

$$
\begin{array}{l}
!\,\mathsf{String}.\ \&\{\,\mathsf{success} : \oplus\{\,\mathsf{deposit} : \ !\,\mathsf{Real}.\ ?\,\mathsf{Real}.\ \mathsf{end} \\
\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{withdraw} : !\,\mathsf{Real}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \&\{\,\mathsf{dispense} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{overdraft} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \\
\qquad\qquad\qquad\quad \} \\
\qquad\quad [\!]\ \ \mathsf{failure} : \mathsf{end} \\
\qquad\quad \}
\end{array}
\tag{3}
$$

With respect to the global type (2), the local view of User is obtained by removing the arrows, being the source and the target of the messages determined, and by using ! or ? when the User is the sender or the receiver of a message, respectively; then by writing $\oplus$ when the User *selects* a choice and & when she/he offers a *branching* of choices.

Similarly we get the type for the ATM view; however, being this symmetric w.r.t. the User's view, the same result can be obtained form (3) by interchanging ! with ? and $\oplus$ with &:

$$
\begin{array}{l}
?\,\mathsf{String}.\ \ \oplus\{\,\mathsf{success} : \ \&\{\,\mathsf{deposit} : \ \ ?\,\mathsf{Real}.\ !\,\mathsf{Real}.\ \mathsf{end} \\
\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{withdraw} : ?\,\mathsf{Real}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \oplus\{\,\mathsf{dispense} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{overdraft} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \\
\qquad\qquad\qquad\quad \} \\
\qquad\quad [\!]\ \ \mathsf{failure} : \mathsf{end} \\
\qquad\quad \}
\end{array}
\tag{4}
$$

The types (3) and (4) show a typical *duality* of session types, which is at the basis of *communication safety* and *session fidelity* properties of the processes typable in the session type systems. Communication safety is the extension to a sequence of communications of the standard correctness property of simple types for (polyadic) π-calculus (see [SW01] Chap. 6), namely that only data of the expected type are exchanged. Session fidelity is instead a typical feature of session type systems, where special names, called *session channels*, may carry messages of different types but in a specific sequence. To illustrate the point let us see how a process implementing the User agent might look:

$$
\begin{array}{l}
\bar{a}(k).\ k\,!\,\mathsf{identifier}. \\
k\ \&\{\,\mathsf{success} : \mathsf{if}\ \cdots\ \mathsf{then}\ k \oplus \ \ \mathsf{deposit} : k\,!\,\mathsf{amount}.\ k\,?\,(x).\ \mathbf{0} \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ k \oplus\ \ \mathsf{withdraw} : k\,!\,\mathsf{amount}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\ k\ \&\{\,\mathsf{dispense} : \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{overdraft} : \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \\
\qquad\quad \mathsf{failure} : \ \mathbf{0} \\
\qquad\quad \}
\end{array}
\tag{5}
$$

where $\cdots$ represent local computations.

On the other hand a process modelling the ATM could be:

$$\begin{array}{l} a(h).\ h\ ?\ (x). \\ \mathsf{if}\ \cdots\ \mathsf{then}\ h\ \oplus \mathsf{success}:\ \ h\&\{\ \mathsf{deposit}: h\ ?\ (y).\ \ \cdots\ h\ !\ \mathsf{balance}.\ \mathbf{0} \\ \qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{withdraw}: h\ ?\ (z). \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{if}\ \cdots \mathsf{then}\ h\ \oplus \mathsf{dispense}: \cdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ h\ \oplus \mathsf{overdraft}: \cdots \\ \qquad\qquad\qquad\qquad\qquad\quad \} \\ \qquad \mathsf{else}\ h\ \oplus \mathsf{failure}: \mathbf{0} \end{array} \tag{6}$$

The name a is a *session name* (simply a name in [HVK98]) and the prefixes $a(h)$ and $\bar{a}(k)$ are used for *session initiation*, named accept and receive in the same work; the names k, h are bound names in the respective bodies representing the *session channels*. As soon as the session begins, the session channels are substituted by a new name κ, with a superscript polarity[3] $p \in \{+, -\}$, marking the two end points of the running session, as it is apparent from the rule for *session initiation*:

$$(\bar{a}(k).P) \mid (a(h).Q) \longrightarrow (\nu\kappa)(P\{\kappa^+/k\} \mid Q\{\kappa^-/h\}).$$

The *polarised channels* $\kappa_1^p, \kappa_2^q, \ldots$ are the medium of all messages between the participants of a session. In fact they are used to *receive and send values*:

$$(\kappa^p\ !\ v.P) \mid (\kappa^{\bar{p}}\ ?\ (x).Q) \longrightarrow P \mid Q\{v/x\}$$

(where $\bar{+} = -$ and $\bar{-} = +$), and to perform *select/branching actions*[4]:

$$(\kappa^p\ \oplus\ \ell_i:\ P) \mid (\kappa^{\bar{p}}\ \&\{\ell_1 : Q_1\ [\!]\ \cdots\ [\!]\ \ell_n : Q_n\}) \longrightarrow P \mid Q_i, \quad (1 \leq i \leq n).$$

Communication safety and session fidelity (simply called “error freeness” in [HVK98]) are induced by a typing discipline, assigning session types $S, S', \ldots$ to session channels, and the type $[S]$ to a session name carrying a session channel typable by S or by $\overline{S}$. So typing judgements are of the shape[5]:

$$\Gamma \vdash P \triangleright \Delta$$

where Γ associates types to session names and variables, while Δ associates session types to session channels. In the sequel we do not report rules for the polarised channels (that are essentially the same), just for simplicity.

[3] The use of polarities has been proposed in [GH05], and adapted to the original syntax in [YV07].

[4] W.r.t. [HVK98], we write $\kappa^p\ \oplus\ \ell_i:\ P$ for $\kappa^p \triangleleft \ell_i;\ P$ and $\kappa^q\ \&\{\ell_1 : Q_1\ [\!] \cdots [\!]\ \ell_n : Q_n\}$ for $\kappa^q\ \triangleright \{\ell_1 : Q_1\ [\!]\ \cdots\ [\!]\ \ell_n : Q_n\}$, just to keep closer process and the type syntax in the present informal exposition.

[5] The syntax of judgments and rules of the session type system is from [YV07] §3, omitting the bases Θ, of use for process definitions that we disregard in this survey.

The typing rules for session initiation assure that the channels bound by session names have exactly the session types prescribed (writing $\overline{S}$ for the dual of S):

$$\frac{\Gamma, a : [S] \vdash P \triangleright \Delta, k : S}{\Gamma, a : [S] \vdash a(k).P \triangleright \Delta} \qquad \frac{\Gamma, a : [S] \vdash P \triangleright \Delta, k : \overline{S}}{\Gamma, a : [S] \vdash \bar{a}(k).P \triangleright \Delta}$$

The assumption $a : [S]$ declares that the session name a is able to open a session whose session channel k has type S. The session type S is constructed along the use of its subject k in the process P, i.e.:

$$\frac{\Gamma, x : T \vdash P \triangleright \Delta, k : S'}{\Gamma \vdash k\ ?\ (x).P \triangleright \Delta, k :? T.S'}$$

whose dual is derived by the rule:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S'' \quad \Gamma \vdash v : T}{\Gamma \vdash k\ !\ v.P \triangleright \Delta, k :! T.S''}$$

Because of these rules, the type $?\,T.S'$ in the conclusion of the first rule tells that over the channel k there will be an input of a value of type T, and then the conversation will continue according to S'; similarly the type $!\,T.S''$ in the conclusion of the second rule tells that the session over k begins with output of a value of type T, and then it continues according to S''. By this we have that $!\,T.S'' = \overline{?\,T.S'}$, provided that $S'' = \overline{S'}$.

Note that to reflect the usage of the session channel in its session type, an almost linear discipline is imposed to the typings Δ. In particular the axiom $\Gamma \vdash \mathbf{0} \triangleright \Delta$ (where $\mathbf{0}$ is the inactive process) requires that Δ associates only the session type end (the type of the completed sessions) to channels. As a consequence weakening of the typing Δ is not admissible but for typings of this form.

We omit the rules for typing selection, branching and parallel composition, which can be found for instance in [YV07].

The actual strength of the π-calculus w.r.t. CCS and similar process algebras consists in the ability to send and receive names. We have seen above that the formalism chosen for the endpoint calculus is essentially a dialect of the π-calculus, extended with session initiation and selection/branching primitives. We will discuss now how a restricted (and more structured) form of mobility allows to express *delegation* in the scenario of sessions and session types.

Consider the more complex version of the User-ATM protocol in Figure 2, which further includes the Bank. The point here is that, to complete its protocol, the ATM asks the Bank to deposit or to withdraw the required amount from the proper bank account. This is accomplished by opening a new session between the ATM and the Bank, which is the agent that ultimately is expected to send

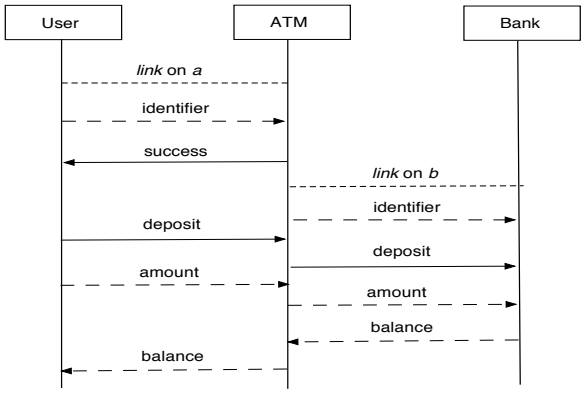

Fig. 2. UML sequence diagram of a User-ATM and Bank interaction.

or to receive the amount determined by the User:

$$
\begin{array}{l}
a(h).\ h\ ?\ (x). \\
\mathsf{if}\ \ldots\,\mathsf{then} \\
\quad h\ \oplus \mathsf{success} : \bar{b}(k).\ k\ !\ x.\ h\&\{\ \mathsf{deposit} : k \oplus \mathsf{deposit} : \\
\qquad\qquad\qquad\qquad\qquad\qquad h\ ?\ (y).\ k\ !\ y.\ k\ ?\ (z).\ h\ !\ z.\ \mathbf{0} \\
\qquad\qquad\qquad\qquad\qquad [\!]\ \mathsf{withdraw} : k \oplus \mathsf{withdraw} : \\
\qquad\qquad\qquad\qquad\qquad\qquad h\ ?\ (t).\ k\ !\ t. \\
\qquad\qquad\qquad\qquad\qquad\qquad k\&\{\ \mathsf{dispense} : h\ \oplus \mathsf{dispense} : \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{overdraft} : h\ \oplus \mathsf{overdraft} : \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \}\ \\
\qquad\qquad\qquad\qquad\qquad \}\ \\
\mathsf{else}\ h\ \oplus \mathsf{failure} : \mathbf{0}
\end{array}
\tag{7}
$$

The service name $\bar{b}$ is used to require a connection to the Bank, and uses the session channel k. Its first use is to send to the Bank the identifier, received on x from the User. Then the ATM plays just the role of a forwarder between the User and the Bank and vice versa. A quite different approach, however, would be to *delegate* (say just after authentication) all the ATM job to the Bank by:

$$
\begin{array}{l}
a(h).\ h\ ?\ (x). \\
\mathsf{if}\ \cdots\ \mathsf{then}\ h\ \oplus \mathsf{success} : \bar{b}(k).\ k\ !\ x.\ k\ !\ h.\ \mathbf{0} \\
\qquad\quad \mathsf{else}\ h\ \oplus \mathsf{failure} : \mathbf{0}
\end{array}
\tag{8}
$$

In the process (8) the session channel h, which is supposed to carry the conversation with the User, is passed along k to the Bank, that will continue the interaction directly with the User. This is however transparent to the User, who is unaware of the fact that the opposite endpoint is now held by some different partner.

Delegation is achieved by allowing higher-order sessions, i.e. by allowing to send channels over channels[6]:

$$(\kappa^p \,!\, \kappa_1^q.P) \mid (\kappa^{\bar{p}} \,?\, (h).Q) \longrightarrow P \mid Q\{\kappa_1^q/h\}$$

How is this reflected in the type system? Is typing able to guarantee to the User that either interaction with the non delegating ATM (7) or with the delegating ATM (8) will always comply with the protocol formalized by the type? As a matter of fact both these issues are addressed by suitably typing the channel exchanges. The rule for the sending process is:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S_1}{\Gamma \vdash k \,!\, h.\, P \triangleright \Delta, k : !\, S_2.S_1, h : S_2}$$

where h is a fresh name. Because of this the new channel h cannot occur in P, even if it is credited of the (arbitrarily complex) usage described in S_2. This is essential for session fidelity to hold: looking at the example (8), if the ATM could save an occurrence of h that could be used after having been sent to the Bank, then the conversation with the User would be ambiguously directed either to the ATM or to the Bank, and the interaction might end up in some unexpected way. For example the process

$$(\kappa^+ \,!\, \kappa_1^+.\kappa_1^+ \,!\, \mathsf{true}.\mathbf{0}) \mid (\kappa^- \,?\, (h).h \,!\, \mathsf{false}.\mathbf{0}) \mid (\kappa_1^- \,?\, (x).P)$$

reduces to

$$(\kappa_1^+ \,!\, \mathsf{true}.\mathbf{0}) \mid (\kappa_1^+ \,!\, \mathsf{false}.\mathbf{0}) \mid (\kappa_1^- \,?\, (x).P)$$

where the linearity of the channel κ_1^+ is lost. The last process can non deterministically give either $(\kappa_1^+ \,!\, \mathsf{false}.\mathbf{0}) \mid P\{\mathsf{true}/x\}$ or $(\kappa_1^+ \,!\, \mathsf{true}.\mathbf{0}) \mid P\{\mathsf{false}/x\}$, so no communication protocol is respected.

On the other hand the receiving process will bind a session channel h:

$$\frac{\Gamma \vdash Q \triangleright \Delta, k : S_1, h : S_2}{\Gamma \vdash k \,?\, (h).Q \triangleright \Delta, k : ?\, S_2.S_1}$$

It is indeed essential that the actual usage in Q of the channel h is controlled by the type S_2, which suffices to guarantee that the delegated session will continue as expected by the partner. This implies that, while the type of k obviously changes, the session type of the delegated session in (8) remains the same as in the case of (7) without delegation.

[6] Observe that, since channel names can only be introduced by the initiation of a session, where they occur within the scope of the restriction operator ν, the communicated names are always private, that is only "internal mobility" is permitted (see [SW01], Chap. 5.7). However in [Bor98] it is shown that the internal π-calculus has the same expressive power, up to barbed-bisimulation, as the asynchronous π-calculus, which in turn is known to encode the full π-calculus: see [SW01], Chap. 5.5.

By admitting recursive definitions of processes, also protocols of unbounded sequences of actions can be expressed.

We remark that while global types have straightforward projections into session types, this fails on the process side. Although this is not the case of our examples, the projection map sending global interactions into end-point processes is quite complex. In fact it is a partial map which is defined only if the given interaction satisfies *connectedness*, *well-threadedness* and *coherence* conditions, as they are detected via a further refinement of the global typing system (for more details see [CHY07]).

The interested reader wishing a more technical presentation of the basics of session types might consult [Vas09a], where Vasconcelos presents a reconstruction of session types in a linear π-calculus with a restriction operator binding at the same time two variables and establishing that they are the two end-points of communications.

3 Extensions

In this section we discuss, mainly through schematic examples, some extensions of sessions and session types that allow to increase their expressivity and consequently to widen their applications.

3.1 Extensions of the Calculus

Correspondence Assertions. In the example (7) of the User-ATM-Bank sketched in the previous section, a malicious ATM' could send to the Bank an amount of money different from that communicated by the User, and consequently altering the balance obtained from the Bank:

$$\begin{array}{l} \mathsf{ATM}' = \\ a(h).\ \cdots \bar{b}(k).\ \cdots \\ \qquad\qquad \mathsf{deposit} : h\ ?\ (y).\ k\ !\ y - 10.\ \ k\ ?\ (z).\ h\ !\ z + 10. \\ \qquad\qquad \cdots \end{array} \tag{9}$$

This change is transparent to the typing, since it does not modify the communication protocol. In order to cope with such kind of misbehaviour, in [BCG05] Bonelli et al. incorporate *correspondence assertions* in the theory of session types. In particular to detect the misbehaviour of ATM' one is enabled to include two correspondence assertions (which are tagged tuples of expressions) into the codes of the User and of the Bank, intended to state that values of both the amount and the balance are the same:

$$\begin{array}{l} \mathsf{User}' = \bar{a}(h).\ \cdots\ h\ !\ \mathsf{amount}.\ h\ ?\ (x).\ \mathsf{cBegin}\ \langle \mathsf{amount}, x\rangle.\ \cdots \\ \mathsf{Bank}' = b(k).\ \cdots\ k\ ?\ (y).\ k\ !\ \mathsf{balance}.\ \mathsf{cEnd}\ \langle y, \mathsf{balance}\rangle.\ \cdots \end{array}$$

Then the type system can discover the malicious behaviours of the ATM' since in the type checking of the process $\mathsf{User}' \mid \mathsf{ATM}' \mid \mathsf{Bank}'$ the tuples $\langle \mathsf{amount}, x\rangle$ and $\langle y, \mathsf{balance}\rangle$, paired by the keywords cBegin and cEnd, do not match.

In general type systems with session types and correspondence assertions can be used to check:

- source of information,
- whether data is propagated as specified across multiple parties,
- if there are unspecified communications between parties, and
- if *the data being exchanged have been modified* by the code in some unexpected way.

Multiparty Sessions. In a multiparty session we can have any number of participants. So a multiparty session forms a unit of structured interactions among many participants which follow a prescribed scenario specified as a global type signature. Multiparty sessions were first designed in [HYC08], but we follow the syntax of [BCD$^+$08], being closer to that one used here for dyadic sessions. For example a global type describing the User-ATM-Bank interaction with three participants is:

$$
\begin{array}{l}
\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : \quad \mathsf{String}. \\
\mathsf{ATM} \longrightarrow \{\mathsf{User}, \mathsf{Bank}\} : \\
\{\ \mathsf{success} : \mathsf{User} \longrightarrow \mathsf{Bank} : \\
\qquad\qquad\qquad\qquad \{\ \mathsf{deposit} : \quad \mathsf{User} \longrightarrow \mathsf{Bank} : \mathsf{Real}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Bank} \longrightarrow \mathsf{User} : \mathsf{Real}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{end} \\
\qquad\qquad\qquad\qquad [\!]\ \mathsf{withdraw} : \mathsf{User} \longrightarrow \mathsf{Bank} : \mathsf{Real}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Bank} \longrightarrow \{\mathsf{User}, \mathsf{ATM}\} : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\ \mathsf{dispose} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \mathsf{overdraft} : \mathsf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \\
\qquad\qquad\qquad\qquad \} \\
[\!]\ \mathsf{failure} : \mathsf{end} \\
\}
\end{array}
\tag{10}
$$

In this context the arrow does not just indicate the direction of a message: $\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : \mathsf{String}$ expresses that the User sends *the same* String to the ATM and to the Bank by means of a unique action. Differently than in the dyadic case, when projecting the global type:

$$\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : \mathsf{String}$$

we have to take into account the roles to which the single actions are projected, giving the slightly more verbose session types:

$$! \langle \{\mathsf{ATM}, \mathsf{Bank}\}, \mathsf{String} \rangle \qquad ? \langle \mathsf{User}, \mathsf{String} \rangle \qquad ? \langle \mathsf{User}, \mathsf{String} \rangle$$

for respectively the User, the ATM and the Bank.

On the process side the session initialization primitives declare the role of the single participants (labelled by a natural number), but for one (distinguished by

the over-bar on the service name) which being the last one declares the overall number of participants. For example, writing the initial actions of each partner in columns which are separated by the parallel composition operator we get for the previous example:

$$\begin{array}{c|c|c} a[1](k_1). & a[2](k_2). & \bar{a}[3](k_3). \\ k_1\,!\,\langle\{2,3\},\mathsf{id}\rangle. & k_2\,?\,\langle 1,x\rangle. & k_3\,?\,\langle 1,y\rangle. \\ \ldots & \ldots & \ldots \end{array}$$

where each communication specifies either the set of the receivers or the sender.

Concurrent Constraints. Following the approach of [BM07,BM08] the paper [CDC09] proposes a calculus which combines concurrent constraints, name passing and sessions. Public and private constraints specify the requirements of session participants to open new interactions and to conduct them. More precisely the primitives for session initiation allow the programmer to specify a set of constraints whose satisfaction is necessary for starting the session interaction. For example a service could offer different times and prices:

$$a\{\mathsf{deliveryTime} = 3 \mid \mathsf{price} = 10\}(k).\ldots$$
$$a\{\mathsf{deliveryTime} = 5 \mid \mathsf{price} = 7\}(k).\ldots$$

so that a rushed client $\bar{a}\{\mathsf{deliveryTime} \leq 4\}(h).\ldots$ will choose the first option; a thrifty client $\bar{a}\{\mathsf{price} \leq 9\}(h).\ldots$ will take the second one; finally a too demanding client $\bar{a}\{\mathsf{deliveryTime} \leq 4 \mid \mathsf{price} \leq 9\}(h).\ldots$ will refuse the connection at all.

In this calculus we have:

- a *fusion* mechanism that explicitly represents, through the notion of constraint, relations involving private and public names,
- *symmetric data communication* both in input and in output, achieved via the introduction of constraints between channel names.

A simple example showing how communication is realised by fusion - i.e. just by creating a new constraint and putting it in parallel with the process continuations - is:

$$\kappa^{+}(\mathsf{amount}).P \mid \kappa^{-}(x).Q \longrightarrow P \mid Q \mid \mathsf{amount} = x$$

The main technical problem is to preserve the linearity of session channel usage in presence of delegation and constraints.

Lopez et al. [LPO10] encode *a timed extension of multi-party sessions* [HYC08] *into the timed process calculus with concurrent constraints* of [OV08]. The timed extension explicitly includes information on session duration, allows for declarative preconditions within session initiations, and features a construct for session abortion. Since the processes of [OV08] can be interpreted as linear temporal logic formulas, the given encoding allows to verify properties of structured communications.

Code Mobility. Mostrous and Yoshida propose in [MY07,MY09] a calculus of sessions in which processes can be sent and received, i.e. a calculus of sessions with *higher-order processes*. The advantage is to avoid many remote interactions. For example the ATM could send a process to the Bank so that the Bank could directly interact with the User. [MY09] discusses also how actions can be permuted in order to increase efficiency.

The main challenge of this approach is the preservation of the linear use of session channels while allowing instantiation of names into executable code.

Exceptions. Carbone et al. in [CHY08] propose a notion of exceptions for sessions which they call *interactional exceptions*. These exceptions demand not only local but also coordinated actions between session participants. The main features of the proposed calculus typed by sessions with exceptions are:

- *flexibility*: exceptions are allowed at any point of a conversation;
- *consistency*: messages in normal and exception conversations are not mixed-up;
- *safety*: communications inside sessions take place linearly and without communication mismatch.

Resource Access Control through Delegation. Capecchi et al. in [CCDR09] enrich the calculus of multiparty sessions with security levels of participants and data. A suitable type system assures that each participant can only receive data of security levels less than or equal to its own security level. For example in a well-typed protocol involving a Customer, a Seller and a Bank, the "secret" credit card number of the Customer is communicated to the Bank, but not to the Seller. This is realised also by making delegation explicit in the typing of the delegated session channel. Typing prevents any leak of information due to selection/branching too.

3.2 Extensions of the Typing

Subtyping. The idea of subtyping, coming from the typed λ-calculus, is that any value of a certain type can be safely placed in a context expecting a value of some more general type: this principle is called *subsumption* (for a handy and clear explanation of the concepts of subtyping and subsumption see [Bru02], Chap. 5). In the setting of the π-calculus, where only names have a type, the subsumption rule takes a dual form (also called *narrowing*): if a type T' describes a more general kind of data than T, written $T \leq T'$, then any name typable by T' in the process P can safely be typed by T in the same process. This is sound with respect to communication safety because for example, an ATM which accepts a Real amount of money can safely communicate with a User who sends an Int amount of money, which is formally expressed by postulating $\mathsf{Int} \leq \mathsf{Real}$ and by deriving $?\mathsf{Int} \leq ?\mathsf{Real}$.

The concept of subtyping, originally conceived for input/output types (see [SW01] Chap. 7, where the covariance/contravariance of input and output actions - respectively - is explained) has been extended to session types by Gay and

Hole in [GH05]. An ATM which offers on a channel both deposit and withdraw can safely communicate through that channel with any User willing just to do a deposit action, which can be expressed by:

$$\&\{\mathsf{deposit} : S_1\} \leq \&\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\}.$$

On the contrary a User who is willing to do a deposit through a certain channel will comply with any environment ready to interact over that channel with someone either asking for a deposit or for a withdraw:

$$\oplus\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\} \leq \oplus\{\mathsf{deposit} : S_1\}.$$

To formalize this in the type system, let us consider the following rule[7]:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S' \quad S \leq S'}{\Gamma \vdash P \triangleright \Delta, k : S}$$

Then if we type the ATM by $k : \&\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\}$ in the premise, we know that it is offering both actions, so that in particular it will do with just deposit, as stated in the conclusion. On the other hand if we know from the premise that the User will do just a deposit, *a fortiori* she/he will be correctly communicate with an ATM accepting either a deposit or a withdraw selection action, which is spelled out in the conclusion.

Summarizing:

- input is covariant,
- output is contra-variant,
- branching is covariant in the number of branches,
- selection is contra-variant in the number of branches,
- both branching and selection are covariant in the continuation types.

This has the remarkable consequence that, if S, S' are session types and $S \leq S'$, then $\overline{S'} \leq \overline{S}$.

Subtyping enhances expressivity of typing with session types since it allows:

- *refinement of participants* without invalidating type-correctness of the overall system,
- *participants to follow different protocols* which are nevertheless compatible according to the subtype relation.

Bounded Polymorphism. A more precise and flexible specification of protocols is obtained in [Gay07] by introducing bounded polymorphism. In particular *a choice of type in one message may affect the types of future messages.* For example

[7] This rule is only admissible in the system studied in [GH05], where a more syntax directed presentation is indeed preferred.

$$\&\{\ \mathsf{opp}\ (\mathsf{Int} \leq X \leq \mathsf{Complex}) : ?\ X.\ !\ X.\ \mathsf{end}$$
$$\cdots$$
$$\}$$

is the type of a calculator which offers an opposite operator working on all numbers whose type is between Int and Complex, returning a number of the same type. A User typed by

$$\oplus\{\ \mathsf{opp} : !\ \mathsf{Real}.\ ?\ \mathsf{Real}.\ \mathsf{end}\}$$

could safely engage a session with such a calculator.

Progress. A very useful property is that once a session is started, the participants will be able to complete all the necessary communications without getting in a deadlock. This property - usually called progress - has been studied for several calculi; in particular Kobayashi has developed very refined techniques for the π-calculus [Kob98,Kob02,Kob05,Kob07].

Session types already assure deadlock-freeness inside single sessions. If distinct sessions do not overlap, then after a session initiation the process is never blocked. This is no longer true if a process contains two or more interleaved sessions: in fact if a session includes another one, then the outermost session might start and wait forever if the innermost session does not find a partner. For example when running the process (7), the session between the User and the ATM opened by a is blocked if there is no Bank hearing on b. In an open scenario we can assume that it is always possible to find the required partners, and therefore we do not consider this kind of cases as deadlocks. There are however situations which cannot be solved by adding suitable partners. A very simple kind of deadlock occurs when two sessions are wrongly interleaved. Consider for example the following typable process:

$$\begin{array}{c|c} a\ (k). & \bar{a}\ (k'). \\ b\ (h). & \bar{b}\ (h'). \\ k\ !\ 2. & h'\ !\ \mathsf{true}. \\ h\ ?\ (x) & k'\ ?\ (y) \\ \cdots & \cdots \end{array}$$

After the two session initiations we get:

$$\begin{array}{c|c} \kappa_a^+\ !\ 2. & \kappa_b^+\ !\ \mathsf{true}. \\ \kappa_b^-\ ?\ (x) & \kappa_a^-\ ?\ (y) \\ \cdots & \cdots \end{array}$$

which is blocked as soon as input and output actions are synchronous. Allowing asynchronous output does not avoid this kind of blocks in general, as it is shown for instance by:

$$\begin{array}{c|c} a\ (k). & \bar{a}\ (k'). \\ b\ (h). & \bar{b}\ (h'). \\ h\ ?\ (x). & k'\ ?\ (y) \\ k\ !\ 2 & h'\ !\ \mathsf{true} \\ \cdots & \cdots \end{array}$$

More interesting examples of deadlocks involve delegation. Type systems assuring progress are discussed in [DCdLY08] for dyadic sessions and in [BCD$^+$08] for multiparty sessions. The key ideas of these works are:

- to take advantage of *nested sessions,*
- to infer the *order of channel usage* for interleaved sessions (following [Kob05]),
- to *forbid "self-delegation"* (opposite polarities of the same session channel cannot be put in sequence).

Action Permutation. As it is well known, in asynchronous π-calculus inputs are blocking while outputs are not (see [SW01], Chap. 5). This asymmetry is the starting point of the work in [HMY09], where Honda et al. propose to *execute outputs before inputs* when possible for increasing efficiency. This change of order is realized by means of an appropriate subtyping theory, which allows automatic action permutation for multiparty sessions while assuring communication safety and session fidelity. Notably action permutation is tricky in presence of recursion and selection/branching.

3.3 Other Extensions

Semantic Subtyping. Semantic subtyping, as proposed in [CF05], is based on the interpretation of types as the sets of their inhabitants, so that subtyping turns out to be set inclusion. Type constructors are indeed interpretable as plain set theoretic operations, so that boolean combinators have their natural meaning.

In [CDCGP09] Castagna et al. propose a theory of session types in which the choices are done on the basis of the type of the messages exchanged. The standard choices through labels are then particular cases in which each label is typed with a singleton type.

An example is the process:

$$\begin{array}{l} \&\{\ k\,?\,(x : \mathsf{Int}).\, k\,!\,-x.\,\mathbf{0} \\ \quad\ k\,?\,(y : \mathsf{Bool}).\, k\,!\,\neg y.\,\mathbf{0} \\ \}\end{array}$$

which, when receiving either an Int or a Bool value, replies differently: in case of an Int number it answers with its opposite; on receiving a Bool value it answers with its negation. The type of the channel k in this process is therefore:

$$\begin{array}{l} \&\{\ ?\mathsf{Int}.\,!\mathsf{Int}.\,\mathsf{end} \\ \quad\ ?\mathsf{Bool}.\,!\mathsf{Bool}.\,\mathsf{end} \\ \}\end{array}$$

Consider now the slightly different type:

$$\begin{array}{l} \&\{\ ?\mathsf{Real}.\,!\mathsf{Nat}.\,\mathsf{end} \\ \quad\ ?\mathsf{Int}.\,!\mathsf{Bool}.\,\mathsf{end} \\ \}\end{array} \tag{11}$$

It can be assigned to a channel which, when receiving a Real number replies with a Nat number, and when getting an Int number answers with a Bool value. Being Int a (semantic) subtype of Real, when the channel receives an Int it can react either by sending a value of type Bool, or, by viewing the integer as a Real, a value of type Nat. A session type which is dual of this type can naturally use boolean operators within type syntax:

$$\begin{array}{l} \oplus\{\ !(\mathsf{Real} \wedge \neg \mathsf{Int}).\ ?\mathsf{Nat}.\ \mathsf{end} \\ \quad\ \ !\mathsf{Int}.\ ?(\mathsf{Nat} \vee \mathsf{Bool}).\ \mathsf{end} \\ \}\end{array}$$

This type says that if the channel sends a Real number which is not an Int number, it will receive a Nat number; but if it sends an Int number, then it will receive either a Nat number or a Bool value. In this way one can obtain a finer description of behaviours within the formalism of session types. Note that also the types $!(\mathsf{Real} \wedge \neg \mathsf{Int}).\ ?\mathsf{Nat}.\ \mathsf{end}$ and $!\mathsf{Int}.\ ?(\mathsf{Nat} \vee \mathsf{Bool}).\ \mathsf{end}$ are dual of (11): namely duality is not involutive in this theory.

In [CDCGP09] also duality is defined semantically: two session types are dual if no conversation on a private channel shared by two processes which follow the prescriptions of these two types ever gets stuck.

In this scenario, where types play a computational role, *session types can be interpreted as the sets of their dual types* and the semantics of boolean combinators is set theoretical. This interpretation of session types gives a semantic subtyping relation, since it is safe to replace a channel with another one when every dual of the replacing channel is also a dual of the replaced one.

Hennessy-Milner Logic. Berger et al. in [BYH08] present an extension of Hennessy-Milner Logic suitable to capture the behaviours of session participants. The basic concept of this logic is the *hypothetical parallel composition* formula $A \gg B$, which means: if a process satisfying A is put in parallel with a process that satisfies this formula, then the resulting process will satisfy B. For example the process

$$P \equiv a(k).\ k\ \oplus\ \mathsf{opp}:\ k\ !\ 2.\ k\ ?\ (x).\ h\ !\ x.\mathbf{0}$$

offers a session initiation on the service name a binding the session channel k, then along k it selects the label opp, sends the integer 2, and receives an input which is bound to the variable x. Eventually it sends over the channel h the value of the variable x. Let Q be a process which offers a session initiation on the service name $\bar{a}$ binding the session channel k'; then using the session channel k' it offers a branch labelled opp, receives an input which is bound to the variable y and then sends the opposite of y. It is clear that the process obtained by putting P and Q in parallel may reduce to a process which sends -2 on the channel h. This is expressed in the logic language by saying that P has the property $A \gg h\ !\ -2\ \mathsf{true}$, where

$$A = \forall y^{\mathsf{Int}}.\bar{a}(k').\ k'\ \&\ \mathsf{opp}:\ k'\ ?\ (y).\ k'\ !\ -y.\ \mathsf{true}.$$

4 Session Embedding in Programming Paradigms

In the previous sections sessions have been considered in the context of the π-calculus. In this section instead we will briefly overview how sessions can be incorporated into two mainstream programming paradigms, i.e. the functional and the object-oriented ones. In this way one can achieve powerful type systems which are suited to programming practice, while retaining the benefit of a sound theoretical foundation.

4.1 Functional Paradigm

Vasconcelos et al. [VGR06,Vas09b] transfer the concept of session and session type to a multi-threaded functional language with side-effecting input/output operations. This shows that static checking of session types can be fruitfully added to a language such as Concurrent ML [Rep99] or Concurrent Haskell [JGF96]. For example a functional version of the process (6) would be:

$$
\begin{array}{lll}
a\ \ h & = \mathsf{let}\ x = \mathsf{receive}\ h\ \mathsf{in} & \\
 & \quad \mathsf{if}\ \cdots \mathsf{then\ select\ success\ on}\ h\ \mathsf{case}\ h\ \mathsf{of}\ \{\ \mathsf{deposit}\ \Rightarrow \cdots & \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{withdraw} \Rightarrow \cdots \} & \\
 & \qquad \mathsf{else\ select\ failure\ on}\ h &
\end{array}
$$

Characteristics of this embedding are:

- *the operations on channels are independent terms* rather than prefixes of processes,
- *the communication is asynchronous*,
- *typing is enhanced by subtyping*, which also allows anticipation of outputs with respect to inputs.

In the recent paper [GV10] Gay and Vasconcelos simplify and extend previous work by giving an operational semantics with buffered channels and by proving that *the session type of a channel gives an upper bound on the necessary size of the buffer*. A novel form of subtyping between standard and linear function types reduces the burden of linear typing on the programmer, by allowing standard function types to be inferred by default and converted to linear types if necessary.

4.2 Object-Oriented Paradigm

Moose. Moose (Multi-threaded Object-Oriented calculus with Sessions) is a multi-threaded object-oriented calculus augmented with session primitives, which supports session names as parameters of methods, spawning, iterative sessions and delegation (see [DCDMY09] and the references there). Progress is enhanced by *spawning a new thread when a session channel is received*: in this way self-delegation never happens. *Choice is made on the basis of the class of the object being sent/received* instead of using labels. Through bounded polymorphism the class of a received object may affect the class of the objects which will be sent.

SAM. The design of the SAM (Sessions Amalgamated with Methods) calculus originates from the comparison between sessions and methods in [CCDC+09]. From this comparison a new notion of session is derived, which subsumes the notion of method. In SAM *classes have fields and sessions*, session bodies are selected on the ground of object classes, and channels are created only at run time when sessions are called. Invocation takes place on an object, say a customer asking to withdraw money from a particular ATM machine, and execution of the corresponding session takes place immediately and concurrently with the requesting thread. The body is defined in the class of the receiving object, namely in the class implementing the ATM of our example, and any number of communications interleaved with computations is possible.

For example the ATM class might contain the declaration of a session:

```
void   ?String . . . atmserver
                     {String x := receive;
                      . . .
                     }
```

where void is the return type of the session, ?String . . . is the session type shown in example (4), atmserver is the session name and the code between brackets is the session body - in this case a translation of the process (6). A User can then call this session on an ATM object by:

```
new  ATM . atmserver {send (identifier);
                        . . .
                     }
```

where the code between brackets is in this case a translation of the process (5). Notably there are no channels in the source code, and only polarised channels will be generated at run time.

Delegation is limited since it does not support an initial and a final dialogue before and after the delegation itself. Expressiveness of typing in SAM has been enhanced with union types [BCDC+08] and generics [CCDC+09].

Session Object Calculus. Mostrous and Yoshida propose in [MY08] an extension of Abadi and Cardelli imperative object calculus (see [AC96]) with sessions, naturally integrating session based choices with method invocations. The main features of this typed calculus are:

- *objects can be spawned, updated and cloned*,
- communication is asynchronous,
- subtyping enjoys the minimal subtyping property.

Modular Session Types as Dynamic Interfaces. In the object-oriented calculus of [VGR+09] the availability of methods depends on object states: object interfaces are dynamic. *Each class has a session type which provides a global specification of the availability of methods at each state.* The typing of a method specifies pre- and

post-conditions for its object states and static typing guarantees that methods are only called when they are available. A key feature is that the state of an object may depend on the result of a method whose return type is an enumeration. Inheritance is included; a subtyping relation on session types characterises the relationship between method availability in a subclass and in its superclass.

Building on [VGR+09], Gay et al. show in [GVR+10] that *a session can be modularised by dividing it into distinct methods that can be called separately.* A key idea is to allow a channel to be stored in a field of an object. Several methods can operate on the same channel, thus allowing to effectively encapsulate channels into objects, while retaining the usual object-oriented development practice.

5 Implementations

Naturally implementations of sessions and session types require their embedding in the used languages, so that it is not surprising that implementations have been done using functional and object-oriented languages, even if to the time among the works surveyed in the previous section just [GVR+10] (see the last paragraph of Section 4.2) is implemented by Bica (see the last paragraph of Section 5.2).

On the contrary it is worthwhile to notice the interplay between the theory sketched in Sections 2 and 3 and the actual implementations of sessions and session types mentioned below. For example the Haskell implementation by Sackman and Eisenbach (see Section 5.1) has first realised the action permutation studied then by Mostrous, Yoshida and Honda (see the last paragraph of Section 3.2). Multiparty sessions were first implemented in Scribble (see Section 5.2) and then formalised by Honda, Yoshida, and Carbone (see Section 3.1).

5.1 Functional Languages

The first implementation of sessions and session types was done by Neubauer and Thiemann [NT04] into *Haskell*. The core of this and of the following implementations into Haskell is the definition of a *session monad*. Type classes with functional dependencies model the progression of the current state of the channel. Functions with polymorphic parameters model client and server side of a communication with one specification.

Sackman and Eisenbach give in [SE08] an implementation of sessions as a standard Haskell library. This implementation presents a monadic API to the programmer. In particular the $SMonad$ type class is a type indexed monad: it allows to represent a computation from a state to another one which additionally produces a value, where the two states can have different types. Since session types are encoded into Haskell types, no preprocessor, external type checker or modification to the Glasgow Haskell compiler are required.

At the address `http://www.agusa.i.is.nagoya-u.ac.jp/person/sydney/full-sessions.html` one can find a tool providing session type inference in Haskell using Haskell type-level programming.

Bhargavan et al. describe in [BCD+09] a compiler from high-level multiparty session descriptions to custom cryptographic protocols coded as ML modules. In the generated code each participant has strong security guarantees for all her/his messages against any adversary that may control both the network and some participants to the session.

5.2 Object-Oriented Languages

The language Sing# [FAH+06] is a variant of C# which combines session types with ownership types [CNP01], supports message-based communication via a designed heap area (shared memory), and allows interfaces between OS-modules to be described as message passing conversations.

SJ [HYH08] is an extension of Java with syntax for session types and structured communication operations. The main features of SJ are asynchronous message passing, delegation, session subtyping, interleaving, class downloading, and failure handling. The compilation-runtime framework of SJ maps session abstraction onto underlying transports, and guarantees communication safety through static and dynamic session type checking. A User coded into SJ could be:

```
s.request( ); s.send(identifier);
             s.inbranch( ) {case success:  if (···){s.outbranch(deposit); ...}
                                           else {s.outbranch(withdraw); ...}
                           }
                           {case failure: }
```

Scribble (`http://sourceforge.net/projects/pi4scribble/`) is a language for describing global (choreography) and local (service end-point) behaviour. Extensible tools are provided, both as stand alone applications and as Eclipse plugins, to edit the language, to perform validation and to export specifications to other formalisms.

Bica (`http://gloss.di.fc.ul.pt/bica/`) implements the type system of [GVR+10]. This implementation comprises an extension to the Java 5 compiler that checks conventional Java source code against session type specifications for classes (included in Java annotations). The extension touches the type checker only: if a program satisfies the more stringent type system of [GVR+10], then code is generated as usual. Bica is implemented with Polyglot.

6 Related Concepts and Formalisms

The present section quickly surveys on formalisms which look to be closely related to sessions and session types, and it contains some pointers to the literature.

6.1 Generic Process Types

Igarashi's and Kobayashi's generic type system (GTS: see [IK04]) is a powerful framework from which one can obtain as instances a variety of type systems for the π-calculus guaranteeing strong properties like deadlock and race-freedom. Not surprisingly also systems of session types can be formalised into

GTS [Kob07,GGR08] although via non trivial translations. However, as observed by Gay et al. in [GGR08], this does not invalidate the usefulness of session types mainly because:

1. *session types are valuable for program design*,
2. session types have been developed for calculi/languages different from π-calculus,
3. proofs of type soundness for session types are fairly straightforward,
4. type checking algorithms for session types cannot be easily obtained via translation, since GTS does not yield an algorithm automatically.

6.2 Contracts

Contracts are behavioural descriptions of Web services [MB03]. In [CGP09] Castagna et al. formalise contracts by means of a sublanguage of CCS without τ (see [DH87]), namely with both external and internal choice, but not including the parallel operator. For example a contract for the ATM process (6) would be:

$$\begin{aligned}\mathsf{Login}.(\quad & \overline{\mathsf{Success}}. \quad (\mathsf{Deposit}.\mathsf{Amount}.\overline{\mathsf{Balance}}.\mathbf{0} \; + \\ & \qquad\qquad \mathsf{Withdraw}.\mathsf{Amount}.(\overline{\mathsf{Dispense}}.\cdots \oplus \overline{\mathsf{Overdraft}}.\cdots)) \\ & \oplus \; \overline{\mathsf{Failure}}.\mathbf{0})\end{aligned}$$

Names and co-names model the input and output actions, respectively; the external choice $+$ is a selection by the ATM counterpart, while the internal choice $\oplus$ represents decisions by the ATM itself.

The main difference between contracts and session types is that contracts record the overall behaviour of a process, while session types project this behaviour onto the private channels that a process uses.

A prominent feature of the theory of contracts is the subcontract relation: if σ is a subcontract of τ, written $\sigma \preceq \tau$, then any client which is satisfied with a service described by σ, will comply with a service described by τ, since the latter possibly includes more capabilities than those described in σ.

In [LP08] Laneve and Padovani give two encodings, from contracts to session types and from session types to contracts[8]. It is also shown that, if $\sigma \preceq \tau$, then the translation of σ is a subtype of the translation of τ in the sense of [GH05].

As remarked in [BCdL10], however, when allowing session delegation, the direct formalisation of the idea that a subcontract can be the description of some "shorter interaction" (as it is in [CGP09]), leads to the collapse of the subtyping relation; this can be avoided at the price of considering subtyping and subcontract as different notions.

The distance between contracts and session types has been narrowed in [CP09] by defining a theory of contracts with explicit channels, so that delegation becomes expressible.

Padovani in [Pad09] presents session types roughly speaking as projections of contracts. The main contributions of this work are:

[8] These encodings are however far more complex than what the last example seems to suggest.

– session types are generalised to processes similar to value-passing CCS,
– session types can be composed by a parallel composition operator (as conversation types, see Section 6.3),
– participants can use channels for communicating after delegating them.

A last remark is that there is a clear similarity between global types and session types on one side and choreography and contracts (as defined in [BZ07]) on the other side. We think that such a relation should be further investigated in order to gain a deeper view of both formalisms.

6.3 Conversation Calculus

The conversation calculus (see [CV09] and the references there) organizes behaviour around places of conversation, which slightly resemble Boxed Ambients [BCC04]. The conversation types record the overall behaviour of processes and assure progress, while accounting for dynamical join/leave of a possibly unanticipated number of participants.

An example of [CV09] showing how the conversation calculus takes advantage of localities is the following composition of two conversation contexts, named *Buyer* and *Seller*:

$$\begin{array}{l} \mathit{Buyer} \blacktriangleleft [\ \textbf{new}\ \mathit{Seller} \cdot\ \texttt{startBuy}\ \Leftarrow\ \texttt{buy!prod.}\ \texttt{price?}(v)\ ]\ \mid \\ \mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ \mid\ \textbf{def}\ \ \texttt{startBuy}\ \Rightarrow \texttt{buy?}(\mathit{prod}).\ \texttt{askPrice}^{\uparrow}!\mathit{prod}. \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{readVal}^{\uparrow}?(v).\ \texttt{price!}v\ ] \end{array}$$

The code $\textbf{new}\ \mathit{Seller} \cdot\ \texttt{startBuy}\ \Leftarrow$ calls the service `startBuy` located at *Seller*. This system reduces to

$$\begin{array}{l} (\nu\ c)(\ \mathit{Buyer}\ \blacktriangleleft [c\ \blacktriangleleft [\ \texttt{buy!prod.}\ \texttt{price?}(v)\ ]\ ]\ \mid \\ \qquad \mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ \mid c\ \blacktriangleleft [\texttt{buy?}(\mathit{prod}).\ \texttt{askPrice}^{\uparrow}!\mathit{prod}. \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{readVal}^{\uparrow}?(v).\ \texttt{price!}v\ ]\ ]\ ) \end{array}$$

where c is the fresh name of the new created conversation context. The code in the *Buyer* side of c sends a product and receives a price, both in the current conversation c. The code in the *Seller* side of c first receives a product in the conversation c, then it consults the database *PriceDB* by means of the messages superscripted by $\uparrow$ which are targeted to the parent conversation (*Seller*), and finally it sends a price in the conversation c.

6.4 Calculi for Web Services

The work on this subject is documented by a large and rapidly growing body of literature, which cannot be accounted for shortly in the present survey. Therefore we just mention three calculi that look more closely related to sessions and session types: the Service Centred Calculus (SCC) [BBC+06], the Calculus of Sessions and Pipelines (CaSPiS) [BBDNL08], and the Calculus for Orchestration of Web Services (COWS) [LPT07]. Common features of these calculi are:

– a clear *distinction between users and services*,
– that *services are permanent*,
– that sessions can only be nested,
– the presence of operators for *explicit closure of sessions*,
– that values can be communicated from an inner session to an outer one (*pipeline*).

For instance, the SCC process $\mathsf{succ} \Rightarrow (x)x + 1$ models a service that, received an integer, gives its successor. A client for this service will be written $\mathsf{succ}\ \{(y)(z)\ \mathsf{return}\ z\} \Leftarrow 5$: after the invocation both x and y are bound to the argument 5, the client waits for a value from the server and the received value (in this case 6) is substituted for z and hence returned as the result of the service invocation.

7 Conclusions

Session types allow the framing of newly emerged issues, in the world of communication centred programming and web services, into the mainstream of type theories and systems, familiar from the functional and object-oriented languages theory and practice. As it is inherent to such approach, they are based on abstractions which just approximate the desired goal of detecting and certifying that certain desirable properties are satisfied by given pieces of code or system specifications. This has however the advantage of being a well-understood technique, that can be implemented efficiently and, by the way, embodied into compilers or software development tools assisting programmers and system designers. On the other hand this should be contrasted with the modelling of processes into process algebras, where powerful but unfeasible concepts of equivalence are used to abstract from implementation details and to distil a precise notion of behaviour.

We think that, as it happens in other fields, it is a matter of balance between expressivity and feasibility, which can be reached only via a deeper understanding of the involved concepts and of their intrinsic complexity. This seems to be the reason why session types look like processes, or why processes - possibly involving few computational combinators - are often thought of as specifications, rather than as concrete implementations. Because of these reasons we think that the work of comparing session types and related systems with other process-based formalisms is worthy and might be fruitful to step to a new generation of calculi and reasoning tools apt to the emerging challenges of the world-wide computing.

Acknowledgments. We gratefully thank Giuseppe Castagna, Ilaria Castellani, Vasco Vasconcelos, Simon Gay, and Luca Padovani for their comments and suggestions on an early draft of the present paper.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[BBC+06] Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Vasconcelos, and Gianluigi Zavattaro. SCC: a Service Centered Calculus. In *WS-FM 2006*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[BBDNL08] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and Pipelines for Structured Service Programming. In *FMOODS'08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[BCC04] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Access Control for Mobile Agents: The Calculus of Boxed Ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[BCD+08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[BCD+09] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *CSF'09*, pages 124–140. IEEE Computer Society, 2009.

[BCDC+08] Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and Union Types for Object Oriented Programming. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 659–680. Springer, 2008.

[BCdL10] Franco Barbanera, Sara Capecchi, and Ugo de' Liguoro. Typing Asymmetric Client-Server Interaction. In *FSEN'09*, volume 5961 of *LNCS*, pages 97–112. Springer, 2010.

[BCG05] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.

[BM07] Marzia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP'07*, volume 4421 of *LNCS*, pages 18 –32. Springer, 2007.

[BM08] Marzia Buscemi and Ugo Montanari. Open Bisimulation for the Concurrent Constraint Pi-Calculus. In *ESOP'08*, volume 4960 of *LNCS*, pages 254–268. Springer, 2008.

[Bor98] Michele Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.

[Bru02] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[BYH08] Martin Berger, Nobuko Yoshida, and Kohei Honda. Completeness and Logical Full Abstraction in Modal Logics for Typed Mobile Processes. In *ICALP'08*, volume 5126 of *LNCS*, pages 99–111. Springer, 2008.

[BZ07] Mario Bravetti and Gianluigi Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *SC'07*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.

[CCDC+09] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science*, 410:142–167, 2009.

[CCDR09] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani, and Tamara Rezk. Session Types for Access and Information Flow Control. available at `http://www.di.unito.it/~dezani/ccdr.pdf`, 2009.

[CDC09] Mario Coppo and Mariangiola Dezani-Ciancaglini. Structured Communications with Concurrent Constraints. In *TGC'08*, volume 5474 of *LNCS*, pages 104–125. Springer, 2009.

[CDCGP09] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of Session Types. In *PPDP'09*, pages 219–230. ACM Press, 2009.

[CF05] Giuseppe Castagna and Alain Frisch. A Gentle Introduction to Semantic Subtyping. In *PPDP'05,* pages 198-208, ACM Press (full version) and *ICALP'05,* LNCS n. 3580, pages 30-34, Springer (summary), 2005. Joint ICALP-PPDP keynote talk.

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009. article n.19, pages 51.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Interactional Exceptions for Session Types. In *CONCUR'08*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[CNP01] David Clarke, James Noble, and John Potter. Simple Ownership Types for Object Containment. In *ECOOP'01*, volume 2072 of *LNCS*, pages 53–76. Springer, 2001.

[CP09] Giuseppe Castagna and Luca Padovani. Contracts for Mobile Processes. In *CONCUR'09*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.

[CV09] Luis Caires and Hugo Torres Vieira. Conversation Types. In *ESOP'09*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.

[DCdLY08] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[DCDMY09] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Session Types for Object-Oriented Languages. *Information and Computation*, 207(5):595–641, 2009.

[DH87] Rocco De Nicola and Mattew Hennessy. CCS Without τ's. In *TAPSOFT'87*, volume 249 of *LNCS*, pages 138–152. Springer, 1987.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys'06*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[Gay07] Simon Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2007.

[GGR08] Simon Gay, Nils Gesbert, and António Ravara. Session Types as Generic Process Types. In *PLACES'08*, pages 16–21, 2008. available at `http://gloss.di.fc.ul.pt/places08/Places08Proceedings.pdf`.

[GH05] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[GV10] Simon Gay and Vasco Vasconcelos. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[GVR+10] Simon Gay, Vasco Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Caldeira. Modular Session Types for Distributed Object-Oriented Programming. In *POPL'10*, pages 299–312. ACM Press, 2010.

[HMY09] Kohei Honda, Dimitris Mostrous, and Nobuko Yoshida. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP'09*, LNCS 5502, pages 316–332. Springer, 2009.

[HVK98] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[IK04] Atsushi Igarashi and Naoki Kobayashi. A Generic Type System for the Pi-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[JGF96] Simon Peyton Jones, Andy Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL'96*, pages 295–308. ACM Press, 1996.

[Kob98] Naoki Kobayashi. A Partially Deadlock-Free Typed Process Calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[Kob02] Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177:122–159, 2002.

[Kob03] Naoki Kobayashi. Type Systems for Concurrent Programs. In *Formal Methods at the Crossroads*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.

[Kob05] Naoki Kobayashi. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informatica*, 42(4–5):291–347, 2005.

[Kob06] Naoki Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[Kob07] Naoki Kobayashi. Type Systems for Concurrent Programs. Extended version of [Kob03], Tohoku University, 2007.

[LP08] Cosimo Laneve and Luca Padovani. The Pairing of Contracts and Session Types. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 681–700, 2008.

[LPO10] Hugo López, Jorge Pérez, and Carlos Olarte. Towards a Unified Framework for Declarative Structured Communications. In *PLACES'09*, number 17 in EPTCS, pages 1–16, 2010.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[MB03] Greg Meredith and Steve Bjorg. Contracts and Types. *Communications of the ACM*, 46(10):41–47, 2003.

[MY07] Dimitris Mostrous and Nobuko Yoshida. Two Sessions Typing Systems for Higher-Order Mobile Processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

[MY08] Dimitris Mostrous and Nobuko Yoshida. A Session Object Calculus for Structured Communication-Based Programming. available at `http://www.doc.ic.ac.uk/~mostrous/sesobj.pdf`, 2008.

[MY09] Dimitris Mostrous and Nobuko Yoshida. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In *TLCA'09*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.

[NT04] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL'04*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[OV08] Carlos Olarte and Frank Valencia. Universal Concurrent Constraint Programming: Symbolic Semantics and Applications to Security. In *SAC'08*, pages 145–150. ACM, 2008.

[Pad09] Luca Padovani. Session Types at the Mirror. In *ICE'09*, volume 12, pages 71–86. EPTCS, 2009.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[SE08] Matthew Sackman and Susan Eisenbach. Session Types in Haskell (Updating Message Passing for the 21st Century). available at `http://pubs.doc.ic.ac.uk/session-types-in-haskell/session-types-in-haskell.pdf`, 2008.

[SW01] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[Vas09a] Vasco Vasconcelos. Fundamentals of Session Types. In *SFM'09*, volume 5569 of *LNCS*, pages 158–186. Springer, 2009.

[Vas09b] Vasco Vasconcelos. Session Types for Linear Multithreaded Functional Programming. In *PPDP'09*, pages 1–6. ACM Press, 2009.

[VGR06] Vasco Vasconcelos, Simon Gay, and António Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science*, 368:64–87, 2006.

[VGR+09] Vasco Vasconcelos, Simon Gay, António Ravara, Niels Gesbert, and Alexandre Caldeira. Dynamic Interfaces. In *FOOL'09*, 2009. available at `http://www.cs.cmu.edu/~aldrich/FOOL09/vasconcelos.pdf`.

[YV07] Nobuko Yoshida and Vasco Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In *SecReT'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.