A Distributed Object-Oriented language with Session types ⋆

Mariangiola Dezani-Ciancaglini[1], Nobuko Yoshida[2],
Alexander Ahern[2], and Sophia Drossopoulou[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computing, Imperial College London

Abstract. In the age of the world-wide web and mobile computing, programming communication-centric software is essential. Thus, programmers and program designers are exposed to new levels of complexity, such as ensuring the correct composition of communication behaviours and guaranteeing deadlock-freedom of their protocols.

This paper proposes the language $\mathcal{L}_{doos}$, a simple distributed object-oriented language augmented with session communication primitives and types. $\mathcal{L}_{doos}$ provides a flexible object-oriented programming style for structural interaction protocols by prescribing channel usages within signatures of distributed classes.

We develop a typing system for $\mathcal{L}_{doos}$ and prove its soundness with respect to the operational semantics. We also show that in a well-typed $\mathcal{L}_{doos}$ program, there will never be a connection error, a communication error, nor an incorrect completion between server-client interactions. These results demonstrate that a consistent integration of object-oriented language features and session types can statically check the consistent composition of communication protocols.

1 Introduction

In distributed systems, physically separated (and potentially mobile) computational entities cooperate or compete by passing code and data to one another. Existing theoretical foundations, which have been successful in sequential programming (as structured programming [9] and type disciplines for programming languages [23]) require non-trivial extensions for the distributed setting. Several new issues arise in this setting, including how to structure communication-based software, how to guarantee security concerns such as confidentiality and integrity, and how to identify correct behaviour of concurrent programs so that we can safely discuss (for example) optimisation of distributed software.

The scenario we are considering in the present paper is a set of users at different locations interacting by means of *object-oriented* code. Distributed objects are one of the most popular programming paradigms in today's computing environments [20], naturally extending the sequential message-passing-oriented paradigm of objects. In current

⋆ Work partially supported by the Royal Society, by EU within the FET - Global Computing initiative, project DART IST-2001-33477, and by EPSRC Advanced Fellowship (GR/T03208/01) and EPSRC GR/R33465/02, GR/S55538/01 and GR/T04724/01.

practice, however, code is often written in terms of bare socket-based communications [21]; it consists of isolated method invocations and returns, and there is no way to ascertain that the code conforms to the intended structure of interaction.

Therefore, the quest for frameworks to enable the expression of *structured interaction*, and for ways to assure the safety of the resulting *interaction protocols* based on that structure, are concerns of paramount importance.

Session types, first introduced in [15], can specify protocols of communication by describing the sequence and types of entities read on a channel. For example, the session type $!\mathbf{int}.!\mathbf{int}.?\mathbf{bool}.\mathbf{end}$ expresses that two **int**-values will be sent, then a **bool**-value is expected as an input, and finally that the protocol is completed. Thus, session types provide a natural way to specify the communication behaviour of a piece of software, and allow verification that several pieces of software are safely composed.

Session types have been widely used to describe protocols in different settings, *i.e.* for π-calculus-based formalisms [4, 5, 13, 15, 17, 25], for CORBA [26], for a λ-calculus with side-effects [14], for a multi-threaded functional language [27], and recently, for a W3C standard description language for web services called Choreography Description Language (CDL) [29]. To our knowledge, the integration of session types into an object-oriented language (even a small, core calculus, as in [3, 10, 18]) has not been attempted so far.

In the present paper we argue that a seamless integration of class-based object-oriented programming and session types is possible, and that the resulting combination offers a powerful framework for writing safe, structured distributed applications with a formal foundation. We substantiate our proposal through the language $\mathcal{L}_{doos}$, a *Distributed Object-Oriented language with Session types*.

By extending class and method signatures to include the types of sessions, we achieved a clean integration of session types into the class based, object-oriented paradigm. Through a combination of remote method invocation (RMI), a standard distributed primitive in objects, session-based distributed primitives [17, 25] and linear interactions [16, 19], we obtained a flexible high-level programming style for remote communication. We also found that the functionality of branching and selection constructs in session types [4, 5, 13–15, 17, 25–27] can be compensated by methods, a natural notion of branching in objects. Subtyping on the branching types [13, 26] is, then, formalised through a standard inheritance mechanism.

Although we did not include branching and selection constructs in $\mathcal{L}_{doos}$, we did include a more specialized construction: conditional and iterative session types. For example, the conditional session type $!\mathbf{int}.!\langle ?\mathbf{char}, !\mathbf{float}\rangle.!\mathbf{int}.\mathbf{end}$ expresses that an integer will be sent followed by a boolean. If this boolean is `true`, then a character will be received, otherwise a float will be sent. Finally, an integer will be sent and the session will complete. Similarly, the iterative session type $!\mathbf{int}.!\langle ?\mathbf{char}.!\mathbf{float}\rangle^*.!\mathbf{int}.\mathbf{end}$ expresses that an integer will be sent followed by a boolean. If this boolean is `true`, then a character will be received, and then a float will be sent, and the process will iterate until a `false` is sent. An integer will then be sent, and the session will complete. Such types allow us to express protocols that require conditionals or repetition on *the same channel*.

To focus on the introduction of session types, $\mathcal{L}_{doos}$ does not include language features such as exceptions [2], synchronisation, serialisation [1], class (down)loading [1, 11], code or agent mobility [1, 7, 28], polymorphism [6, 18, 27], recursive types [26] or correspondence assertions [4, 5]. We believe that the inclusion of such features into $\mathcal{L}_{doos}$ is possible, albeit not necessarily trivial.

A key point for the safety of session communication is channel linearity. To check linearity by typing in an imperative object-oriented setting where object fields can contain channels requires sophisticated types, see for example [12]. In $\mathcal{L}_{doos}$ channel linearity as in [4, 5, 13–15, 17, 25, 27] comes from creating a private fresh channel name every time a session starts. Typing then ensures that all communication in the current session uses this new channel, and that after the session is completed there are no further occurrences of this channel. In this way we also avoid the need to deal with opening and closing operations on channels [27].

Apart from guaranteeing that all communications have the expected types (soundness), our type system guarantees that in a a well-typed $\mathcal{L}_{doos}$ program, there will never be a connection error (*i.e.* request and accept on same channel will have the same type), nor a communication error (*i.e.* never two simultaneous send or receive on same channel), nor an incorrect completion between server-client interactions (*i.e.* after a session started, it will complete on each of the participants, unless there is an exception, or divergence, or an unsuccessful attempt to start a further session). Thus, the type system can statically check the consistent composition of communication protocols.

The soundness of our system is weaker than that of all systems of session types for π-calculus processes [4, 5, 13, 15, 17, 25]. In fact all these systems assure a perfect pairing between processes willing to communicate. This is obtained simply by checking the compatibility of type environments before putting processes in parallel. Our system instead, following the approach of [14, 27], only ensures that a communication will safely evolve *after starting*: there is no guarantee that processes ready to start a session will ever find a companion. It is not difficult to add to our system a compatibility check between environments to ensure the stronger soundness discussed above, but we chose to avoid it since our aim is to model an open distributed system where new processes can appear at run time, and so no global assumption on safety liveness can be guaranteed.

In the remainder, Section 2 illustrates the basic ideas of $\mathcal{L}_{doos}$ through an example. Section 3 defines the syntax of the language. Section 4 presents the operational semantics. Section 5 illustrates the typing system. Section 6 is devoted to basic theorems on type safety and communication safety. Section 7 concludes.

A preliminary version of this paper is [8].

2 Example

The following example demonstrates some of the features of $\mathcal{L}_{doos}$.[3] It describes a situation where a seller employs an agent to sell some item to some buyer for the best price possible:

[3] Note that in order to write our example more naturally we use several constructs which are not part of our minimum language $\mathcal{L}_{doos}$, *i.e.* types float and void, methods without parameters, local variables, and conditionals, which can easily be added to $\mathcal{L}_{doos}$.

The agent begins negotiations by asking the seller both the price and by the minimum price. Then the agent sends the price to the buyer. The buyer, upon receipt of this price, makes an offer which he sends to the agent. The agent calculates whether the offer exceeds the minimum price and notifies the seller and the buyer accordingly. If the offer does not exceed the minimum price then the agent invites the seller to lower his minimum price and the negotiation iterates. Note however that the agent may now communicate with a different buyer, but will continue communicating with the same seller.

The example consists of classes Agent, Buyer, and Seller, each of which we shall now discuss separately:

```
1  class Agent extends Object {
2    float price, minPrice;      // seller's asking and minimum price
3    float offer;                // the offer made by the buyer
4
5     bool tryToSell () c1: !float.?float.!bool.end {
6           // connect with a buyer
7           request c1 : !float.?float.!bool.end {
8                c1.send(price); offer := c1.receive; c1.send(offer<minPrice);
9                return( offer <minPrice ); } }
10
11   void mediate() c2: ?float.?float.!⟨?float⟩*.end {
12         // connect with a seller
13         request c2 ?float.?float.!⟨?float⟩*.end {
14              price := c2.receive; minPrice := c2.receive;
15              c2.sendWhile ( tryToSell() )
16                   // if the value of tryToSell () is true
17                   { minPrice := c2.receive; }    }}
18 }
```

The class Agent represents the agent, with fields price to store the asking price, and minPrice to store the minimum price. The signature of the method tryToSell contains the type of the channel c1, *i.e.* !**float**.?**float**.!**bool**.**end**, thus indicating that c1 will send one **float** value, will then receive a **float** value, and then send a **bool** value.

Indeed, in the body of this method, the agent asks for a connection with a buyer through a channel c1 by the statement **request** c1 ..., which must be matched by a statement **accept** c1 ... at another node in the network.

In general, **accept** u s{e} represents the creation of a new server-side socket as in the java.net.ServerSocket class. Here u can be either a public channel name c (as in line 6 of class Buyer) or a variable x whose value is a public channel name c. In both cases the name c is analogous to the port used to instantiate the ServerSocket, which is the port on which the server will listen for connections. Execution proceeds when another node in the network contains a statement **request** u′ s{e′} where u′ is either the name c or a variable whose value is c. The statement **request** is similar to the creation of a new client-side socket from the java.net.Socket class. Here the name c can be thought of as corresponding to the hostname and port number of the server socket. When these match, execution continues and a new private channel is created to connect the two

nodes. Execution of e and e′ proceeds concurrently, with all occurrences of u in e and all occurrences of u′ in e′ replaced by the name of the just created channel. So both public channel names and channel variables play the role of placeholders in session bodies, since they are replaced by restricted and fresh channel names.

In the method tryToSell, after the connection has been established, *i.e.* in the body of the **request** c1 ..., the agent sends the asking price (c1.**send**(price)), then receives the buyer offer along the same channel (offer := c1.**receive**). Lastly he compares the offer with the minimum price and then decides on behalf of the buyer whether the offer was successful, and tells the buyer through c1 (c1.**send**(offer < minPrice)).

The signature of the method mediate contains the type of channel c2, *i.e.* c2 : ?**float**.?**float**.!⟨?**float**⟩*.**end**, which is an *iterative* session type, and which indicates that c2 will receive two **float**s and then send a **bool**; it that boolean is true, it will iterate, otherwise it will be the end of the session. The body of method mediate asks for a connection through channel c2, receives the asking and the minimum price along that channel, and then attempts a sale using method tryToSell (which returns a boolean). It sends the value of tryToSell along channel c2 to the seller; if the value is true, then it iterates, by receiving a new asking price along channel c2.

```
1  class Buyer extends Object {
2      float price ;   //   seller 's asking price
3      float offer ;   //   offer made by the buyer
4
5      void buy() c1 : !float.?float .!bool.end {
6          accept c1 !float.?float .!bool.end {
7          // connect with an agent
8              price := c1.receive; offer :=....;    c1.send(offer);
9              if c1.receive then .... else ... } }
10 }
```

The class Buyer represents the buyer, with fields price, offer, with the obvious meaning. In the method buy, the buyer connects with some agent, receives the asking price, calculates his offer and send it. He then receives a boolean indicating whether the seller's agent accepted the bid, and proceeds with appropriate actions. The signature of the method buy contains the type of the channel c1, *i.e.* !**float**.?**float**.!**bool**.**end**. Notice that this type describes the session from the viewpoint of the Agent, which is dual to that of the Buyer.

The class Seller represents the seller, with fields price and minPrice for the asking and the minimum price. The type of the channel c2 in method sell is the same that in mediate in Agent.

The method sell starts by calculating the asking and minimum prices. After the connection on channel c2 is established, the seller sends the asking and minimum prices along the newly created channel. It then receives a boolean value indicating whether the negotiations need to continue. If so, then the seller will proceed with the body of the **receiveWhile** ... statement, and will calculate a new minimum price and send it on the same channel to the agent. This process is repeated until the seller receives false, *i.e.* until no more negotiations are required.

Our example demonstrates session types and in particular the use of branching and iterative session types to express repetition and conditional execution over the same channel. [4]

```
1  class Seller {
2    float price,minPrice;      // asking price and minimum price
3
4    void sell ( ) c2: ?float.?float .! ⟨?float⟩*.end {
5      price:= ... ; minPrice:= ... ;
6      // connect to an agent
7      accept c2 ?float.?float .! ⟨?float⟩*.end {
8        c2.send(price); c2.send(minPrice);
9        c2.receiveWhile
10         // if the value received is true, then
11         { minPrice:= ... ; c2.send(minPrice); } }   }
12 }
```

The present example can be seen as a simplified object-oriented version of the Auctioneer example in [4]; the main difference is that the type system of [4] using correspondence assertions can detect bad behaviours which are type correct in our system.

Our type system guarantees the consistent composition of communication protocols of the various participants. Thus, it guarantees that:

- All communications have the expected types, *e.g.* in the method buy, in line 8 the expression c1.**receive** will return a **float**, while in line 9 the same expression will return a **bool**.
- There will never be a connection error, *e.g.* when line 13 of method mediate establishes a connection, it will only be with a channel of the appropriate type.
- There will never be a a communication error, *e.g.* when line 14 of method mediate performs c2.receive, there will not be a simultaneous **receive** on channel c2.
- There will never be an incorrect completion between server-client interactions, *e.g.* once the session in line 13 of mediate started, it will complete in each of the participants, unless there is an exception, or divergence. In particular notice that all iterations in line 15 will be successful.

3 A Distributed Object Oriented Language with Sessions

User syntax We distinguish *user syntax*, for programs at a local node, and *runtime syntax*, which occurs only at runtime as intermediate forms. We introduce the user syntax in Fig. 1. It is an extension of FJ [18], MJ [3] and DJ [1] (while omitting the new distributed primitives introduced in [1]), augmented with primitives for session communication [5, 15, 17, 27].

[4] In earlier work [8] we had shown sessions as first class values, (*e.g.* objects containing session channels), assigning session values to session variables, session types carrying session types, and nesting of sessions. However these constructs are not sufficient to enforce repeated execution on the *same* channel.

$$
\begin{array}{lrl}
\text{(type)} & \mathtt{t} & ::= C \mid \mathbf{bool} \mid \mathtt{s} \\
\text{(direction)} & \dagger & ::= \ \ ! \mid ? \\
\text{(part of session)} & \pi & ::= \ \ \varepsilon \mid \dagger \mathtt{t} \mid \dagger\langle \pi, \pi \rangle \mid \dagger\langle \pi \rangle^* \mid \pi.\pi \\
\text{(session)} & \mathtt{s} & ::= \ \ \pi.\mathbf{end} \\
 & & \\
\text{(meth sig)} & \mathit{methSig} & ::= \mathtt{t\ m\ (t)\ \Sigma} \\
\text{(class sig)} & \mathtt{CSig} & ::= \emptyset \mid \mathtt{CSig}, \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\mathit{field}^*\ \mathit{methSig}^*\} \\
\text{(session env)} & \Sigma & ::= \emptyset \mid \Sigma, \mathtt{c} : \mathtt{s} \\
\text{(class table)} & \mathtt{CT} & ::= \emptyset \mid \mathtt{CT}, \mathit{class} \\
 & & \\
\text{(class)} & \mathit{class} & ::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
\text{(field)} & \mathit{field} & ::= \mathtt{f\,t} \\
\text{(method)} & \mathit{meth} & ::= \mathtt{t\ m\ (t\ x)\ \Sigma\ \{e\}} \\
 & & \\
\text{(expression)} & \mathtt{e} & ::= \ \mathtt{x} \mid \mathtt{v} \mid \mathtt{this} \mid \mathtt{true} \mid \mathtt{false} \\
 & & \quad \mid \mathtt{e;\ e} \mid \ \mathbf{new}\ C \mid \mathtt{x := e} \mid \mathtt{e.f := e} \mid \mathtt{e.f} \mid \mathtt{e.m(e)} \\
 & & \quad \mid \mathtt{u}.\mathbf{receive} \mid \mathtt{u}.\mathbf{send}(\mathtt{e}) \\
 & & \quad \mid \mathtt{u}.\mathbf{receiveIf}\{\mathtt{e}\}\{\mathtt{e}\} \mid \mathtt{u}.\mathbf{sendIf}(\mathtt{e})\{\mathtt{e}\}\{\mathtt{e}\} \\
 & & \quad \mid \mathtt{u}.\mathbf{receiveWhile}\{\mathtt{e}\} \mid \mathtt{u}.\mathbf{sendWhile}(\mathtt{e})\{\mathtt{e}\} \\
 & & \quad \mid \mathbf{request}\ \mathtt{u\ s}\{\mathtt{e}\} \mid \mathbf{accept}\ \mathtt{u\ s}\{\mathtt{e}\} \\
\text{(identifier)} & \mathtt{u} & ::= \mathtt{c} \mid \mathtt{x} \\
\text{(value)} & \mathtt{v} & ::= \mathtt{null} \mid \mathtt{c}
\end{array}
$$

Fig. 1. User Syntax

The metavariable $\mathtt{t}$ ranges over types for channels and expressions, C ranges over class names, $\mathtt{s}$ ranges over session types. ? means *input*, while ! means *output*, and $\dagger$ ranges over $\{!, ?\}$, while **end** indicates the end of the session.

The metavariable π describes *parts* of a session. The *conditional* session part $!\langle \pi_1, \pi_2 \rangle$ sends a boolean value and proceeds with π_1 if the value is true, or π_2 if the value is false. Similarly $?\langle \pi_1, \pi_2 \rangle$ receives a boolean value and proceeds with π_1 if the value is true, π_2 if it is false. The *iterative* session part $!\langle \pi_1 \rangle^*$ sends a boolean value and if that value is true, continues with π_1, *iterating*. If the value sent is false, this session part finishes. The meaning of $?\langle \pi_1 \rangle^*$ is similar. Note that, the closing of a session, **end**, cannot appear within a conditional or iterative session part. This supports the design principle that sessions have to be closed at the level where they were opened; in other words, the responsibility of closing a session stays with the party that opened it.

To prescribe the channel usage in a method, we introduce *session environments*, Σ, which map channels to session types. Method declarations have the shape

$$\mathtt{t\,m\ (t\,x)\ \Sigma\ \{e\}}$$

which is standard, except for the addition of Σ.

A *Class signature*, CSig, denotes a class's interface [1]; it contains the types of fields, its superclass name and method signatures. This provides a lightweight mech-

(type)	$\mathtt{t} ::= \ldots \mid \mathbf{chan}(\mathtt{t})$
(identifier)	$\mathtt{u} ::= \ldots \mid \mathtt{o}$
(value)	$\mathtt{v} ::= \ldots \mid \mathtt{o}$
(expression)	$\mathtt{e} ::= \ldots \mid \mathsf{NullExc}$
(thread)	$P ::= \mathtt{e} \mid P \mid P$
(store)	$\sigma ::= \emptyset \mid \sigma{\cdot}[x \mapsto \mathtt{v}] \mid \sigma{\cdot}[\mathtt{o} \mapsto (C, \vec{\mathtt{f}} : \vec{\mathtt{v}})]$
(network)	$N ::= \mathbf{0} \mid l[P, \sigma, \mathtt{CT}] \mid N \| N \mid (\nu \mathtt{u} : \mathtt{t})N$

Fig. 2. Runtime Syntax

anism for determining the type of remote methods. We assume that CSig is available globally (this does not restrict generality, since in standard implementations uniqueness of each class is maintained through its digital signature). In contrast, class tables (containing method bodies) are maintained on a per-location basis.

The syntax of expressions, $\mathtt{e}$, $\mathtt{e}'$, is standard except for the four pairs of communication primitives. The first two lines express standard syntax, *i.e.* parameter, value, the receiver this, the literals true and false, sequence of expressions, object creation, assignment to parameters or fields, field access and method call. The next four lines describe the four communication pairs.

The first pair is for exchange of values or channels: $\mathtt{u}.$**receive** receives a value or a channel via $\mathtt{u}$, while $\mathtt{u}.$**send**$(\mathtt{e})$ first evaluates the expression $\mathtt{e}$, then sends its result via $\mathtt{u}$.

The second pair is for *conditional* communication: $\mathtt{u}.$**receiveIf**$\{\mathtt{e}\}\{\mathtt{e}'\}$ receives a value via $\mathtt{u}$, and if it is true continues with $\mathtt{e}$, otherwise with $\mathtt{e}'$. The expression $\mathtt{u}.$**sendIf**$(\mathtt{e})\{\mathtt{e}'\}\{\mathtt{e}''\}$ first evaluates the boolean expression $\mathtt{e}$, then sends its result via $\mathtt{u}$ and if the result was true continues with $\mathtt{e}'$, otherwise with $\mathtt{e}''$.

The third pair is for *iterative* communication: $\mathtt{u}.$**receiveWhile**$\{\mathtt{e}\}$ receives a value via $\mathtt{u}$, and if it is true continues with $\mathtt{e}$ and iterates, otherwise ends. The expression $\mathtt{u}.$**sendWhile**$(\mathtt{e})\{\mathtt{e}'\}$ first evaluates the boolean expression $\mathtt{e}$, then sends its result via $\mathtt{u}$ and if the result was true continues with $\mathtt{e}'$ and iterates, otherwise ends.

The last pair is for establishing connections: **request** $\mathtt{u}\,\mathtt{s}\{\mathtt{e}\}$ is for use by clients, and **accept** $\mathtt{u}\,\mathtt{s}\{\mathtt{e}\}$ for use by servers. The channel $\mathtt{u}$ denotes a shared interaction point which is used for creating new channels. In both **request** $\ldots\mathtt{s}\{\mathtt{e}\}$, and **accept** $\ldots\mathtt{s}\{\mathtt{e}\}$, the term $\{\mathtt{e}\}$ (called *session body*) denotes the block of (a sequence of) expressions in which the new channel is created at the beginning, and discarded at the end; the session $\mathtt{s}$ prescribes the communication protocol, which is opened by **request** or **accept**.

Runtime Syntax The runtime syntax in Fig. 2 extends the user syntax and represents a distributed state of multiple sites communicating with each other. The syntax uses *location names* $l, m, \ldots$ which can be thought of as IP addresses in a network.

Metavariable $\mathtt{t}$ is extended with *runtime channel types*, denoting the channel types used only for method invocations. Identifiers, $\mathtt{u}$, and values, $\mathtt{v}$, are extended to allow for object identifiers $\mathtt{o}, \mathtt{o}', \ldots$, which denote references to instances of classes. We shall

frequently write "o-id" for brevity, and we shall call o and c *names*. We extend expressions with $\mathsf{NullExc}$, denoting a null-pointer error. *Threads* are ranged over by P, P', where $P \mid P'$ says that P and P' are running in parallel.

A store σ contains local variables and objects, and $\vec{\mathsf{f}} : \vec{\mathsf{v}}$ is short-hand for a sequence $\mathsf{f}_1 : \mathsf{v}_1 ; \ldots ; \mathsf{f}_n : \mathsf{v}_n$. We apply similar abbreviations to other sequences [1, 18]. Sequences contain no duplicate names.

Networks, written N, comprise zero or more located configurations executing in parallel. We use $\mathbf{0}$ to denote the empty network, $l[P, \sigma, \mathtt{CT}]$ to denote the thread P executing at location l with store σ and class table $\mathtt{CT}$, $N_1 \parallel N_2$ is the parallel composition of two networks, and $(\nu \mathsf{u} : \mathsf{t})N$ makes the identifier u local to N.

The binding is standard and we use $\mathsf{fn}(\mathsf{e})/\mathsf{fv}(\mathsf{e})$ to denote a set of free names/variables. We say that a class name C occurs *free* in a expression e if e contains **new** C: the function $\mathsf{fcl}(\mathsf{e})$ returns the set of free class names of e.

4 Operational Semantics

This section presents the operational semantics of $\mathcal{L}_{doos}$, which extends the standard small step call-by-value reduction of [1, 3, 23]. The reduction relation is given modulo the standard structural equivalence rules of the π-calculus [22], written $\equiv$. We define *multi-step* reduction as: $\longrightarrow\!\!\!\!\rightarrow \stackrel{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$. We only discuss the more interesting rules. We start by listing the evaluation contexts.

$$\begin{array}{rl} E ::= & [\,] \mid E.\mathsf{f} \mid E;\mathsf{e} \mid \mathsf{x} := E \mid E.\mathsf{f} := \mathsf{e} \mid \mathsf{o}.\mathsf{f} := E \mid E.\mathsf{m}(\mathsf{e}) \mid \mathsf{o}.\mathsf{m}(E) \mid \mathsf{c}.\mathbf{send}(E) \\ & \mid \mathsf{u}.\mathbf{sendIf}(E)\{\mathsf{e}\}\{\mathsf{e}\} \mid \mathsf{u}.\mathbf{sendWhile}(E)\{\mathsf{e}\} \end{array}$$

Notice that $\mathbf{request}\, E\, \mathsf{s}\{\mathsf{e}\}$, and $\mathbf{accept}\, E\, \mathsf{s}\{\mathsf{e}\}$, are not evaluation contexts.[5] Neither are $\mathbf{request}\, \mathsf{u}\, \mathsf{s}\{E\}$, $\mathbf{accept}\, \mathsf{u}\, \mathsf{s}\{E\}$, $\mathsf{u}.\mathbf{sendIf}(\mathsf{e})\{E\}\{\mathsf{e}\}$, $\mathsf{u}.\mathbf{sendIf}(\mathsf{e})\{\mathsf{e}\}\{E\}$, $\mathsf{u}.\mathbf{sendWhile}(\mathsf{e})\{E\}$, $\mathsf{u}.\mathbf{receiveIf}\{E\}\{\mathsf{e}\}$, $\mathsf{u}.\mathbf{receiveIf}\{\mathsf{e}\}\{E\}$, or $\mathsf{u}.\mathbf{receiveWhile}\{E\}$ evaluation contexts, because they would allow session bodies to run before the start of the session, or parts of a conditional or iterative session to run before determining which conditional branch should be selected, or whether the iteration should continue.

Local Expressions The rules for execution of expressions which correspond to the sequential part of the language are standard [3, 10, 18]. Only the local store is modified, and the rules involve only the local store and the local class table. In Fig. 3 we give the rules for object creation and method invocation.

[5] Namely, if $\mathbf{request}\, E\, \mathsf{s}\{\mathsf{e}\}$ were an evaluation context, it would replace the name of a channel in E without replacing it in e. For example, then, for some session type s, and some state σ_1, where $\sigma_1(\mathsf{x}) = \mathsf{c}$, and applying also rule **RN-ReqAcc**, we would have:
$\ldots\mathbf{request}\, \mathsf{x}\, \mathsf{s}\{\mathsf{x}.\mathbf{receive}\}\ldots\sigma_1\ldots \parallel \ldots\mathbf{accept}\, \mathsf{c}\, \mathsf{s}\{\mathsf{c}.\mathbf{send}(3)\}\ldots\sigma_2\ldots \longrightarrow$
$\ldots\mathbf{request}\, \mathsf{c}\, \mathsf{s}\{\mathsf{x}.\mathbf{receive}\}\ldots\sigma_1\ldots \parallel \ldots\mathbf{accept}\, \mathsf{c}\, \mathsf{s}\{\mathsf{c}.\mathbf{send}(3)\}\ldots\sigma_2\ldots \longrightarrow$
$(\nu \mathsf{c}' : \mathsf{s})(\ldots x.\mathbf{receive}\ldots\sigma_1\ldots \parallel \ldots \mathsf{c}'.\mathbf{send}(3)\ldots\sigma_2\ldots)$, and execution would be stuck.
For similar reasons, $\mathbf{accept}\, E\, \mathsf{s}\{\mathsf{e}\}$ is not a context.

RC-New

$$\frac{\mathsf{fields}(C) = \vec{\mathtt{f}}\,\vec{\mathtt{t}}}{\textbf{new}\ C,\sigma,\mathtt{CT} \longrightarrow (\nu \mathsf{o}:C)(\mathsf{o},\sigma\cdot[\mathsf{o}\mapsto(C,\vec{\mathsf{f}}:\vec{\mathtt{null}})],\mathtt{CT})}\ C\in\mathsf{dom}(\mathtt{CT})$$

RC-LocMeth

$$\frac{\sigma(\mathsf{o}) = (C,\ldots)\quad \mathsf{mbody}(\mathsf{m},C,\mathtt{CT}) = (\mathsf{x},\mathsf{e})\quad \mathsf{mtype}(\mathsf{m},C) = \mathtt{t}\to\mathtt{t}'}{\mathsf{o}.\mathsf{m}(\mathsf{v}),\sigma,\mathtt{CT} \longrightarrow (\nu \mathsf{x}:\mathtt{t})(\mathsf{e}[\mathsf{o}/\mathtt{this}],\sigma\cdot[\mathsf{x}\mapsto\mathsf{v}],\mathtt{CT})}$$

Fig. 3. Expression Reduction

Allocation of new objects, described by **RC-New**, explicitly restricts identifiers, thus representing "freshness" or "uniqueness" of the address in the store. The function $\mathsf{fields}(C)$ examines the class signature and returns the field declarations for C.

The method invocation rule is **RC-LocMeth**; the function $\mathsf{mbody}(\mathsf{m},C,\mathtt{CT})$ looks up m in the local class table, and returns a pair consisting of the method code and the formal parameter name. The function $\mathsf{mtype}(\mathsf{m},C)$ looks up m in the global class signature and returns the type of the method [18]. The receiver o replaces `this` in the method body and a new store entry x is allocated for the formal parameter v.

Communication $\mathcal{L}_{doos}$ has two kinds of communication rules: those for *remote method and field invocation*, and those for *session communication*, which are inspired by π-calculus rules [22]. Fig. 4 defines reduction for remote method and field invocation; the first three rules are for congruence, the fourth rule is structural.

Rule **RN-Fld** allows reading at location l_1 a field of an object stored at a *different location*, l_2. Similarly, **RN-FldAss** allows the code in location l_1 to assign a value to a field stored in a different location, l_2.

Rule **RN-RemMeth** describes remote method call; location l_1 executes a method call where the receiver is an object stored in a different location l_2: a new runtime private channel c, shared between l_1 and l_2, is created; after that, at l_2 the method call is executed by rule **RC-LocMeth**; the result v is then safely sent back from l_2 to l_1 via this new private channel c by **RN-CommMeth**; since c is only used once (*i.e.* it is a linear channel in the sense of [1, 16, 19]), it is finally discarded.

Session Communication The main session communication rules are formalised in Fig. 5. Rule **RN-ReqAcc** describes opening of sessions: if location l_1 requires a session on u_1 and location l_2 accepts a session on u_2 and the values of u_1 and u_2 are the same channel name, then, a new private channel c is created and u_1 and u_2 are replaced by c in the session bodies in the standard way noting that

$$\textbf{request}\ \mathsf{u}'\mathsf{s}\{\mathsf{e}\}[\mathsf{c}/\mathsf{u}] = \textbf{request}\ \mathsf{u}'\mathsf{s}\{\mathsf{e}[\mathsf{c}/\mathsf{u}]\}$$
$$\textbf{accept}\ \mathsf{u}'\mathsf{s}\{\mathsf{e}\}[\mathsf{c}/\mathsf{u}] = \textbf{accept}\ \mathsf{u}'\mathsf{s}\{\mathsf{e}[\mathsf{c}/\mathsf{u}]\}$$

$$\textbf{RN-Conf}\quad \frac{P,\sigma,\mathtt{CT} \longrightarrow (\nu\vec{u}:\vec{t})(P',\sigma',\mathtt{CT})}{l[P,\sigma,\mathtt{CT}] \longrightarrow (\nu\vec{u}:\vec{t})(l[P',\sigma',\mathtt{CT}])} \qquad \textbf{RN-Par}\quad \frac{N \longrightarrow N'}{N \,|\!|\, N_0 \longrightarrow N' \,|\!|\, N_0}$$

$$\textbf{RN-Res}\quad \frac{N \longrightarrow N'}{(\nu u:t)N \longrightarrow (\nu u:t)N'} \qquad \textbf{RN-Str}\quad \frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

RN-Fld

$$l_1[E[\mathsf{o}.\mathsf{f_i}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[Q,\sigma_2,\mathtt{CT}_2] \longrightarrow l_1[E[\mathsf{v_i}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[Q,\sigma_2,\mathtt{CT}_2]$$
$$\sigma_2(\mathsf{o}) = (C,\vec{\mathsf{f}}:\vec{\mathsf{v}})$$

RN-FldAss

$$l_1[E[\mathsf{o}.\mathsf{f}:=\mathsf{v}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[Q,\sigma_2,\mathtt{CT}_2] \longrightarrow l_1[E[\mathsf{v}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[Q,\sigma_2',\mathtt{CT}_2]$$
$$\mathsf{o} \in \mathsf{dom_o}(\sigma_2) \quad \sigma_2{}' = \sigma_2[\mathsf{o} \mapsto \sigma_2(\mathsf{o})[\mathsf{f} \mapsto \mathsf{v}]]$$

RN-RemMeth

$$l_1[E[\mathsf{o}.\mathsf{m}(\mathsf{v})] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[Q,\sigma_2,\mathtt{CT}_2]$$
$$\longrightarrow$$
$$(\nu\mathsf{c}:\mathbf{chan}(\mathsf{t}))(l_1[E[\mathsf{c}.\mathbf{receive}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[\mathsf{c}.\mathbf{send}(\mathsf{o}.\mathsf{m}(\mathsf{v})) \,|\, Q,\sigma_2,\mathtt{CT}_2])$$
$$\sigma_2(\mathsf{o}) = (C,\ldots) \quad \mathsf{mtype}(\mathsf{m},C) = \mathsf{t}' \to \mathsf{t} \quad \mathsf{c} \text{ fresh}$$

RN-CommMeth

$$(\nu\mathsf{c}:\mathbf{chan}(\mathsf{t}))(l_1[E_1[\mathsf{c}.\mathbf{send}(\mathsf{v})] \,|\, Q_1,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[E_2[\mathsf{c}.\mathbf{receive}] \,|\, Q_2,\sigma_2,\mathtt{CT}_2])$$
$$\longrightarrow$$
$$l_1[E_1[\mathsf{null}] \,|\, Q_1,\sigma_1,\mathtt{CT}_1] \,|\!|\, l_2[E_2[\mathsf{v}] \,|\, Q_2,\sigma_2,\mathtt{CT}_2]$$

Fig. 4. Network Communication

but importantly

$$\mathbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}[\mathsf{c}/\mathsf{u}] = \mathbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$$
$$\mathbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}[\mathsf{c}/\mathsf{u}] = \mathbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$$

i.e. substitutions of synchronisation channel names cannot move inside nested sessions synchronising on the same name. The freshness of c guarantees privacy and linearity of the session communication between l_1 and l_2. Notice that stores associate values with variables, so if u_1 is a variable of type s then $\sigma_1(\mathsf{u}_1)$ will be a channel name, and similarly for u_2.

Rule **RN-CommSess** formalises the session communication where sent value v has the type $\mathtt{t}$; after a series of applications of this rule, the session completes and the channel c has type **end**.

In rules **RN-CommSessIf-true** and **RN-CommSessIf-false** first a boolean is exchanged, and then according to the value of this boolean the execution proceeds with the first or the second branches.

Rule **RN-CommSessWhile** simply expresses the iteration by means of the conditional.

RN-ReqAcc

$$l_1[E_1[\mathbf{request}\ \mathsf{u}_1\ \mathsf{s}\{\mathsf{e}_1\}]] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathbf{accept}\ \mathsf{u}_2\ \mathsf{s}\{\mathsf{e}_2\}] \mid Q_2, \sigma_2, \mathtt{CT}_2] \longrightarrow (\nu \mathsf{c} : \mathsf{s})(l_1[E_1[\mathsf{e}_1[\mathsf{c}/\mathsf{u}_1] \mid Q_1], \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{e}_2[\mathsf{c}/\mathsf{u}_2]] \mid Q_2, \sigma_2, \mathtt{CT}_2]) \quad \mathsf{c} \text{ fresh}$$

u_1 and u_2 are the same channel name or $\sigma_1(\mathsf{u}_1) = \mathsf{u}_2$ or $\mathsf{u}_1 = \sigma_2(\mathsf{u}_2)$ or $\sigma_1(\mathsf{u}_1) = \sigma_2(\mathsf{u}_2)$

RN-CommSess

$$(\nu \mathsf{c} : \dagger \mathtt{t}.\mathsf{s})(l_1[E_1[\mathsf{c}.\mathbf{send}(\mathsf{v})] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{c}.\mathbf{receive}] \mid Q_2, \sigma_2, \mathtt{CT}_2]) \longrightarrow (\nu \mathsf{c} : \mathsf{s})(l_1[E_1[\mathsf{null}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{v}] \mid Q_2, \sigma_2, \mathtt{CT}_2])$$

RN-CommSessIf-true

$$(\nu \mathsf{c} : \dagger\langle \pi_1, \pi_2 \rangle.\mathsf{s})(l_1[E_1[\mathsf{c}.\mathbf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{c}.\mathbf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \mid Q_2, \sigma_2, \mathtt{CT}_2]) \longrightarrow (\nu \mathsf{c} : \pi_1.\mathsf{s})(l_1[E_1[\mathsf{e}_1] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{e}_3] \mid Q_2, \sigma_2, \mathtt{CT}_2])$$

RN-CommSessIf-false

$$(\nu \mathsf{c} : \dagger\langle \pi_1, \pi_2 \rangle.\mathsf{s})(l_1[E_1[\mathsf{c}.\mathbf{sendIf}(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{c}.\mathbf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \mid Q_2, \sigma_2, \mathtt{CT}_2]) \longrightarrow (\nu \mathsf{c} : \pi_2.\mathsf{s})(l_1[E_1[\mathsf{e}_2] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{e}_4] \mid Q_2, \sigma_2, \mathtt{CT}_2])$$

RN-CommSessWhile

$$(\nu \mathsf{c} : \dagger\langle \pi \rangle^*.\mathsf{s})(l_1[E_1[\mathsf{c}.\mathbf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{c}.\mathbf{receiveWhile}\{\mathsf{e}_2\}] \mid Q_2, \sigma_2, \mathtt{CT}_2]) \longrightarrow (\nu \mathsf{c} : \dagger\langle \pi.\dagger\langle \pi \rangle^*, \varepsilon \rangle.\mathsf{s})(l_1[E_1[\mathsf{c}.\mathbf{sendIf}(\mathsf{e})\{\mathsf{e}_1; \mathsf{c}.\mathbf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{c}.\mathbf{receiveIf}\{\mathsf{e}_2; \mathsf{c}.\mathbf{receiveWhile}\{\mathsf{e}_2\}\}\{\mathsf{null}\}] \mid Q_2, \sigma_2, \mathtt{CT}_2])$$

Fig. 5. Session Communication

5 Session Types and Typing System

The type system of $\mathcal{L}_{doos}$ has three kinds of typing judgments. The judgments for threads and nets are standard, they just tell us that under certain assumptions on the types of variables, o-ids, `this` and channels, the thread and respectively the net is well-formed. So the judgments have the shape:

$$\Gamma \vdash P : \textbf{thread} \quad \text{and} \quad \Gamma \vdash N : \textbf{net}$$

where the environment Γ is defined by:

$$\Gamma := \emptyset \mid \Gamma, \mathtt{x} : \mathtt{t} \mid \Gamma, \mathtt{o} : C \mid \Gamma, \mathtt{this} : C \mid \Gamma, \mathtt{c} : \mathtt{s} \mid \Gamma, \mathtt{c} : \textbf{chan}(\mathtt{t})$$

When typing expressions we need to take into account how session types are "consumed", *i.e.* when an input or an output communication prescribed by a session type takes place through **receive** or **send** instruction. For this reason we add session environments to both sides of typing judgments, giving them the shape

$$\Gamma; \Sigma \vdash \mathtt{e} : \mathtt{t}; \Sigma'$$

where Γ is the environment, $\mathtt{t}$ is the type of $\mathtt{e}$, Σ and Σ' give the session types of channels before and after the evaluation of $\mathtt{e}$. We call them the *pre* and *post session environment* respectively.

Notice that since **request** and **accept** instructions contain the session types of the connecting channels and method declarations contain the session environment (i.e. the session types of the used channels), we could avoid global assumptions on session types of channel names. The cost would be a run time check that the session types in request and accept coincide before starting sessions.

In the following subsections we will discuss the more interesting rules. We only mention here that there is a standard subtyping (denoted by $<:$), which we assume causes no cycle as in [3, 18], and which is judged on the class signature.

Well-formed class tables Methods, classes and class tables are well-formed with respect to an environment which must contain all session environments of methods. This is prescribed by the rule checking that a method is ok:

M-ok

$$\frac{\Sigma, \mathtt{this} : C, \mathtt{x} : \mathtt{t}_1; \emptyset \vdash \mathtt{e} : \mathtt{t}; \emptyset}{\Gamma, \mathtt{this} : C \vdash \mathtt{t}_2\, \mathtt{m}(\mathtt{t}_1\, \mathtt{x}) \Sigma\{\mathtt{e}\} : \mathtt{ok\ in}\ C} \qquad \begin{array}{l} \Sigma \subseteq \Gamma \\ \mathtt{mtype}(\mathtt{m}, C) = \mathtt{t}_1 \to \mathtt{t}_2 \\ \mathtt{t} <: \mathtt{t}_2 \end{array}$$

The environment Γ is propagated in the rules for checking well-formedness of classes and class tables.

Notice that both the pre and the post session environments for typing the method body are empty. This ensures that all send and receive instructions are inside sessions as we will see in discussing thread and network typing.

Expression typing The rule for typing expression composition illustrates a first use of session environments:

$$\textbf{TE-Seq}\quad \frac{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t};\Sigma' \qquad \Gamma;\Sigma' \vdash \mathsf{e}' : \mathsf{t}';\Sigma''}{\Gamma;\Sigma \vdash \mathsf{e};\mathsf{e}' : \mathsf{t}';\Sigma''}$$

The post session environment Σ' of e typing is used as pre session environment for typing e'. The typing rule for method calls:

$$\textbf{TE-Meth}\quad \frac{\Gamma;\Sigma \vdash \mathsf{e} : C;\Sigma' \qquad \Gamma;\Sigma' \vdash \mathsf{e}' : \mathsf{t}';\Sigma''}{\Gamma;\Sigma \vdash \mathsf{e}.\mathsf{m}(\mathsf{e}') : \mathsf{t};\Sigma''} \quad \begin{array}{l} \mathsf{msignature}(\mathsf{m},C) \subseteq \Gamma \\ \mathsf{mtype}(\mathsf{m},C) = \mathsf{t}'' \to \mathsf{t} \\ \mathsf{t}' <: \mathsf{t}'' \end{array}$$

demands that the method signature of m in C (determined by the method signature look-up function $\mathsf{msignature}(\mathsf{m},C)$) is contained in the environment Γ. Further, the session environments of e and e' must agree as in rule **TE-Seq**. Finally the type of e' should conform to the method type returned by the look-up function $\mathsf{mtype}(\mathsf{m},C)$.

Session typing The importance of the session environments in expression typing is made clear by the rules for typing **send** and **receive**:

$$\textbf{TE-SessSend}\quad \frac{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t};\Sigma', \mathsf{c} : !\mathsf{t}.\mathsf{s}}{\Gamma;\Sigma \vdash \mathsf{c}.\textbf{send}(\mathsf{e}) : \mathit{Object};\Sigma', \mathsf{c} : \mathsf{s}} \qquad \textbf{TE-SessReceive}\quad \frac{}{\Gamma;\Sigma, \mathsf{c} : ?\mathsf{t}.\mathsf{s} \vdash \mathsf{c}.\textbf{receive} : \mathsf{t};\Sigma, \mathsf{c} : \mathsf{s}}$$

The key observation is that in both cases the typing consumes exactly the output or the input type that heads the session type of the current channel c. The typing of **send** also takes into account that the typing of e can modify the session environment.

The typing rules for opening sessions are:

$$\textbf{TE-Req}\quad \frac{\Gamma, \mathsf{u} : \mathsf{s};\Sigma, \mathsf{c} : \mathsf{s} \vdash \mathsf{e}[\mathsf{c}/\mathsf{u}] : \mathsf{t};\Sigma', \mathsf{c} : \textbf{end} \quad \mathsf{c} \notin \mathsf{fn}(\mathsf{e}) \quad \mathsf{c} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{u} : \mathsf{s};\Sigma \vdash \textbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\} : \mathsf{t};\Sigma'}$$

$$\textbf{TE-Acc}\quad \frac{\Gamma, \mathsf{u} : \mathsf{s};\Sigma, \mathsf{c} : \overline{\mathsf{s}} \vdash \mathsf{e}[\mathsf{c}/\mathsf{u}] : \mathsf{t};\Sigma', \mathsf{c} : \textbf{end} \quad \mathsf{c} \notin \mathsf{fn}(\mathsf{e}) \quad \mathsf{c} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{u} : \mathsf{s};\Sigma \vdash \textbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\} : \mathsf{t};\Sigma'}$$

where $\overline{\mathsf{s}}$ denotes the *dual* session type of s defined inductively by $\overline{\textbf{end}} = \textbf{end}$, $\overline{!\mathsf{t}.\mathsf{s}} = ?\mathsf{t}.\overline{\mathsf{s}}$, $\overline{?\mathsf{t}.\mathsf{s}} = !\mathsf{t}.\overline{\mathsf{s}}$, and the substitution $[\mathsf{c}/\mathsf{u}]$ obeys the same conditions as given in Section 4.[6]

[6] Notice that the name of the channel, u, is replaced by a fresh channel name, c. This is so, because, a) u may be a variable, but Σ contains only constant channels, and b) it allows us to type nested session openings of the same name, *e.g.* **request** $\mathsf{c}\ \mathsf{s}\{...$**request** $\mathsf{c}\ \mathsf{s}\{...\}...\}$.

The key point is that these rules ensure *linear* use of runtime session channels; for every new session, there should be exactly one receiver waiting to receive from $\mathtt{c}$, and one sender waiting to send on $\mathtt{c}$. This is guaranteed by replacing the opening channel $\mathtt{u}$ in $\mathtt{e}$ by a fresh channel $\mathtt{c}$. The type **end** of $\mathtt{c}$ in the post session environment of typing $\mathtt{e}$ ensures that the session is completed after evaluation of $\mathtt{e}$. Notice that $\mathtt{c}$ does not appear in the conclusion.

The remaining rules give types for conditional and iterative session types. Note that within iterations depending on the value received/sent on a channel $\mathtt{c}$, rules **TE-SessRecWhile** and **TE-SessSendWhile** forbid communication on any other open channel except for $\mathtt{c}$; *e.g.* for $\mathtt{c}.\mathbf{sendWhile}(\mathtt{e}')\{\mathtt{e}\}$ and $\mathtt{c}.\mathbf{receiveWhile}\{\mathtt{e}\}$, the typing rules require for any communication $\mathtt{c}'.\mathbf{receive}$ or $\mathtt{c}'.\mathbf{send}(...)$ within $\mathtt{e}$ that $\mathtt{c}=\mathtt{c}'$, or that the communication is enclosed within an inner $\mathbf{accept}\ \mathtt{c}'\mathtt{s}\{...\}$ or $\mathbf{request}\ \mathtt{c}'\mathtt{s}\{...\}$. This constraint is clearly necessary in order to get soundness of communications (Theorem 6.3).

TE-SessRecIf

$$\frac{\Gamma;\Sigma,\mathtt{c}:\pi_1.\mathtt{s}\vdash \mathtt{e}_1:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s} \qquad \Gamma;\Sigma,\mathtt{c}:\pi_2.\mathtt{s}\vdash \mathtt{e}_2:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s}}{\Gamma;\Sigma,\mathtt{c}:?\langle\pi_1,\pi_2\rangle.\mathtt{s}\vdash \mathtt{c}.\mathbf{receiveIf}\{\mathtt{e}_1\}\{\mathtt{e}_2\}:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s}}$$

TE-SessSendIf

$$\frac{\Gamma;\Sigma\vdash \mathtt{e}:\mathbf{bool};\Sigma \qquad \Gamma;\Sigma,\mathtt{c}:\pi_1.\mathtt{s}\vdash \mathtt{e}_1:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s} \qquad \Gamma;\Sigma,\mathtt{c}:\pi_2.\mathtt{s}\vdash \mathtt{e}_2:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s}}{\Gamma;\Sigma,\mathtt{c}:!\langle\pi_1,\pi_2\rangle.\mathtt{s}\vdash \mathtt{c}.\mathbf{sendIf}(\mathtt{e})\{\mathtt{e}_1\}\{\mathtt{e}_2\}:\mathtt{t};\Sigma',\mathtt{c}:\mathtt{s}}$$

TE-SessRecWhile

$$\frac{\Gamma;\Sigma,\mathtt{c}:\pi.\mathtt{s}\vdash \mathtt{e}:\mathtt{t};\Sigma,\mathtt{c}:\mathtt{s}}{\Gamma;\Sigma,\mathtt{c}:?\langle\pi\rangle^*.\mathtt{s}\vdash \mathtt{c}.\mathbf{receiveWhile}\{\mathtt{e}\}:\mathtt{t};\Sigma,\mathtt{c}:\mathtt{s}}$$

TE-SessSendWhile

$$\frac{\Gamma;\Sigma\vdash \mathtt{e}:\mathbf{bool};\Sigma \qquad \Gamma;\Sigma,\mathtt{c}:\pi.\mathtt{s}\vdash \mathtt{e}':\mathtt{t};\Sigma,\mathtt{c}:\mathtt{s}}{\Gamma;\Sigma,\mathtt{c}:!\langle\pi\rangle^*.\mathtt{s}\vdash \mathtt{c}.\mathbf{sendWhile}(\mathtt{e})\{\mathtt{e}'\}:\mathtt{t};\Sigma,\mathtt{c}:\mathtt{s}}$$

Thread and Network typing Rule **TT-Start** promotes expressions to threads; all channels of the post session environment should be completed (*i.e.* be typed by **end**) and all sessions in the pre session environment should conform to the environment.

TT-Start

$$\frac{\Gamma;\{\mathtt{c}_i:\mathtt{s}_i \mid i\in I\}\vdash \mathtt{e}:\mathtt{t};\{\mathtt{c}_i:\mathbf{end} \mid i\in I\} \quad \forall i\in I.\mathtt{c}_i:\mathtt{s}_i\in\Gamma\vee\mathtt{c}_i:\overline{\mathtt{s}_i}\in\Gamma}{\Gamma\vdash \mathtt{e}:\mathbf{thread}}$$

Notice that when all send and receive operations are inside sessions, both the pre and the post session environments for typing $\mathtt{e}$ can be empty.

Rule **TN-Conf** states that a location is a well-typed network in an environment if its thread P is well-typed, its store σ and class table $\mathtt{CT}$ are $\mathtt{ok}$ in the same environment, and if all free classes in P as well as their superclasses (we denote this set by $\mathsf{fcl}(P)$) are locally available – the latter is guaranteed through the requirement $\mathsf{fcl}(P)\subseteq \mathsf{dom}(\mathtt{CT})$

and the last condition.

$$\textbf{TN-Conf}\quad \frac{\Gamma \vdash P : \textbf{thread} \quad \Gamma \vdash \sigma : \mathtt{ok} \quad \Gamma \vdash \mathtt{CT} : \mathtt{ok} \quad \mathsf{fcl}(P) \subseteq \mathsf{dom}(\mathtt{CT}) \quad \forall C \in \mathsf{dom}(\mathtt{CT}) \,.\, C <: D \vee D \in \mathsf{fcl}(C, \mathtt{CT}) \implies D \in \mathsf{dom}(\mathtt{CT})}{\Gamma \vdash l[P, \sigma, \mathtt{CT}] : \textbf{net}}$$

6 Type Safety and Communication Safety

As expected, the type system of Section 5 satisfies the subject reduction property.This is formulated as follows.

Theorem 6.1 (Subject Reduction).

- *If* $\Gamma; \Sigma \vdash \mathtt{e} : \mathtt{t}; \Sigma'$, *and* $\Gamma \vdash \sigma : \mathtt{ok}$, *and* $\Gamma \vdash \mathtt{CT} : \mathtt{ok}$ *and* $\mathtt{e}, \sigma, \mathtt{CT} \longrightarrow (\nu \vec{u} : \vec{t}')(\mathtt{e}', \sigma', \mathtt{CT})$ *then* $\Gamma, \vec{u} : \vec{t}'; \Sigma \vdash \mathtt{e}' : \mathtt{t}'; \Sigma'$ *with* $\mathtt{t}' <: \mathtt{t}$ *and* $\Gamma, \vec{u} : \vec{t}' \vdash \sigma' : \mathtt{ok}$.
- *If* $\Gamma \vdash P : \textbf{thread}$, *and* $\Gamma \vdash \sigma : \mathtt{ok}$, *and* $\Gamma \vdash \mathtt{CT} : \mathtt{ok}$ *and* $P, \sigma, \mathtt{CT} \longrightarrow (\nu \vec{u} : \vec{t}')(P', \sigma', \mathtt{CT})$ *then* $\Gamma, \vec{u} : \vec{t}' \vdash P' : \textbf{thread}$ *and* $\Gamma, \vec{u} : \vec{t}' \vdash \sigma' : \mathtt{ok}$.
- *If* $\Gamma \vdash N : \textbf{net}$, *and* $N \longrightarrow N'$ *then* $\Gamma \vdash N' : \textbf{net}$.

The proof is based on generation lemmas, substitution lemmas and a detailed analysis of channel use.

Even more interesting than subject reduction, are the following properties of $\mathcal{L}_{doos}$:

P1 no *connection error* can occur, *i.e.* request and accept on the same channel must have the same session type;
P2 no *communication error* can occur, *i.e.* in the same net there cannot be two sends or two receives on the same channel;
P3 after a session has begun *the required communications are always executed in the expected order*;
P4 after a session has begun *all the required communications are executed* unless one of the following situations occurs:
- a null pointer exception is thrown;
- the computation diverges; or
- there is a request or accept instruction waiting for the dual instruction.

These properties hold for a network obtained by reduction from an initial network. We say that a network N is *initial* if (writing $\prod_{0 \le i < n} N_i$ for $N_0 \,\|\, N_1 \,\|\, \ldots \,\|\, N_{n-1}$):

- $\vdash N : \textbf{net}$ is derivable using rule **TT-Start** with empty session environments in the premises;
- $N \equiv (\nu \vec{c} : \vec{s})(\prod_{0 \le i < n} l_i[\mathtt{e}_i, \emptyset, \mathtt{CT}_i])$, where each $\mathtt{e}_i$ is a user expression; and
- N is closed.

Notice that the condition on the use of rule **TT-Start** is satisfied whenever all send and receive instructions are inside method bodies, a natural choice in the object-oriented paradigm.

In order to formulate properties **P1** and **P2**, we add a new constant ConmErr (*connection or communication error*) to the network and the following rule:

$$l_1[E_1[\mathsf{e}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathsf{e}'] \mid Q_2, \sigma_2, \mathtt{CT}_2] \longrightarrow \mathsf{ConmErr}$$

if e *clashes* with e', where e clashes with e' when

$$\mathsf{e}, \mathsf{e}' \in \{\mathsf{c}.\mathbf{receive}, \mathsf{c}.\mathbf{send}(\ldots), \mathsf{c}.\mathbf{receiveIf}\{\ldots\}\{\ldots\}, \mathsf{c}.\mathbf{sendIf}(\ldots)\{\ldots\}\{\ldots\},\\ \mathsf{c}.\mathbf{receiveWhile}\{\ldots\}, \mathsf{c}.\mathbf{sendWhile}(\ldots)\{\ldots\}, \mathbf{request}\ \mathsf{c}\ \mathsf{s}\{\ldots\}, \mathbf{accept}\ \mathsf{c}\ \mathsf{s}\{\ldots\}\}$$

and they do not occur both in the premise of one of the rules in Fig. 5. In other words when e and e' belong both to the above set they do not clash if $\mathsf{e} = \mathsf{c}.\mathbf{receive}$ and $\mathsf{e}' = \mathsf{c}.\mathbf{send}(\mathsf{e}'_0)$, or $\mathsf{e} = \mathsf{c}.\mathbf{receiveIf}\{\mathsf{e}_0\}\{\mathsf{e}_1\}$ and $\mathsf{e}' = \mathsf{c}.\mathbf{sendIf}(\mathsf{e}'_0)\{\mathsf{e}'_1\}\{\mathsf{e}'_2\}$, or $\mathsf{e} = \mathsf{c}.\mathbf{receiveWhile}\{\mathsf{e}_0\}$ and $\mathsf{e}' = \mathsf{c}.\mathbf{sendWhile}(\mathsf{e}'_0)\{\mathsf{e}'_1\}$, or $\mathsf{e} = \mathbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_0\}$ and $\mathsf{e}' = \mathbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}'_0\}$ or vice versa.

We can now prove that from initial nets, we never reach a configuration containing clashing expressions.

Theorem 6.2 (ConmErr **Freedom).** *Suppose that N_0 is an initial net and $N_0 \longrightarrow\!\!\!\!\!\rightarrow N$. Then N does not contain* ConmErr*, i.e. there does not exist N' such that $N \equiv N' \parallel \mathsf{ConmErr}$.*

The proof of the above theorem is straightforward from the subject reduction theorem.

For properties **P3** and **P4** we formulate the following soundness theorem:

Theorem 6.3 (Soundness). *Let N_0 be an initial net, $N_0 \longrightarrow\!\!\!\!\!\rightarrow (\nu\vec{u}:\vec{t})N$, and $(\nu\vec{u}:\vec{t})N \longrightarrow (\nu c:s)(\nu\vec{u}:\vec{t})N' \stackrel{\mathsf{def}}{=} (\nu c:s)N_1$ by rule* **RN-ReqAcc** *with $s = \pi_1.\pi.\pi_2.\mathbf{end}$. If $(\nu c:s)N_1$ does not*

- *produce* NullExc *or*
- *diverge or*
- *stop on a request or accept instruction waiting for the dual instruction*

then

$$(\nu c:s)N_1 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\pi.\pi_2.\mathbf{end})N_2 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\pi_2.\mathbf{end})N_3 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\mathbf{end})N_4$$

with $c \notin \mathsf{fn}(N_4)$, where:

- *if $\pi = \dagger t$ then $(\nu c:\pi.\pi_2.\mathbf{end})N_2 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\pi_2.\mathbf{end})N_3$ with exactly one application of rule* **RN-CommSess** *on channel c;*
- *if $\pi = \dagger\langle\pi',\pi''\rangle$ then the first rule involving channel c is*
  - *either* **RN-CommSessIf-true** *and the application of this rules gives $(\nu c:\pi'.\pi_2.\mathbf{end})N'_2$ and $(\nu c:\pi'.\pi_2.\mathbf{end})N'_2 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\pi_2.\mathbf{end})N_3$;*
  - *or* **RN-CommSessIf-false** *and the application of this rules gives $(\nu c:\pi''.\pi_2.\mathbf{end})N'_2$ and $(\nu c:\pi''.\pi_2.\mathbf{end})N'_2 \longrightarrow\!\!\!\!\!\rightarrow (\nu c:\pi_2.\mathbf{end})N_3$;*

– *if* $\pi = \dagger\langle\pi'\rangle^*$ *then the first rule involving channel* c *is* **RN-CommSessWhile** *and the application of this rules gives* $(\nu \mathsf{c} : \pi_3.\pi_2\,.\mathbf{end})N_2'$ *with* $\pi_3 \in \{\dagger\langle\pi'\rangle^*, \varepsilon\}$ *and* $(\nu \mathsf{c} : \pi_3.\pi_2\,.\mathbf{end})N_2' \longrightarrow\!\!\!\!\!\rightarrow (\nu \mathsf{c} : \pi_2\,.\mathbf{end})N_3$.

The soundness proof requires careful analysis of the evaluation order and invariant properties of networks.

Finally we get:

Theorem 6.4 (Completion of Sessions). *Suppose* N_0 *is an initial net,* $N_0 \longrightarrow\!\!\!\!\!\rightarrow N \equiv (\nu \vec{\mathsf{u}} : \vec{\mathsf{t}})\prod_{0\leq \mathsf{i}<\mathsf{n}} l_{\mathsf{i}}[\mathsf{e}_{\mathsf{i}}, \sigma_{\mathsf{i}}, \mathsf{CT}_{\mathsf{i}}]$ *and* N *is irreducible. Then either all* e_{i} *are values* $(0 \leq \mathsf{i} < \mathsf{n})$ *or there is* j $(0 \leq \mathsf{j} < \mathsf{n})$ *such that* $\mathsf{e}_{\mathsf{j}} \in \{\mathsf{NullExc}, E[\mathbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}'\}], E[\mathbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}'\}]\}$.

7 Conclusions and Further Work

Session types have been successfully applied to theoretical settings such as the π-calculus [4, 5, 13, 15, 17, 25], a multi-threaded functional language [27], to practical settings such as CORBA [26] and a web-services description language [29]. With $\mathcal{L}_{doos}$ we aimed to link language development to engineering and standardisation practice.

To our knowledge $\mathcal{L}_{doos}$ is the first application of session types to a distributed, object-oriented class-based programming language. Our design aims were to restrict the number of novel features introduced into the object-oriented language (we added only four pairs of primitives for standard session communication in the user syntax), and to obtain a simple typing system by extending class and method signatures to contain the usage of channels assigned by session types. We have written several example programs, demonstrating that $\mathcal{L}_{doos}$ can express communication in a style that is natural for programmers from the object-oriented community.

It is worthwhile to notice that our session types are regular expressions of a limited shape, which can also be denoted by sum and recursion. Branching types instead are variant types, and therefore the recursive session types of [13, 14, 17, 26] are richer than ours.

The subtyping relation on session types considered in [13, 26] is covariant for input, contravariant for output as in [24] and moreover allow to change the number of branches in branching types. As our session types are regular expressions, the inclusion of regular languages induces a natural notion of subtyping which is simple but not interesting, because it lacks covariance and contravariance of inputs and outputs.

We plan to investigate extensions that would allow channels to carry channels, and channels to be passed as parameters to methods. In particular, we want to allow the passing of linear channels, through the use of π-types as parameter types; on the other hand, in order to ensure linearity, we will forbid π-types as the types of local variables or fields.

Furthermore, we will re-evaluate our design decision of omitting selection primitives from the $\mathcal{L}_{doos}$-session types. While in traditional session types, function names are included in types (*e.g.* sell:?**float**.?**float**.$\langle$!**float**$\rangle^*$.**end** would be the session type of the seller), in $\mathcal{L}_{doos}$ they are not included (*e.g.* ?**float**.?**float**.$\langle$!**float**$\rangle^*$.**end** is the type of a *channel* used by sell). With this design decision the structure of the program is primarily

reflected in the classes and their methods, and therefore method names were not a part of the sessions types.

Finally, we wish to evaluate the various designs through a sequence of case studies and to develop type checking algorithms following [4].

Acknowledgements We are grateful to the anonymous referees, to Kohei Honda, and to Simon Gay for pointing out some errors in the examples, and some weaknesses in the presentation. We are indebted to Dimitris Mostrous for many insightful comments. During and after the presentation at TGC-05 we had many interesting questions and suggestions from the participants, in particular from Rocco De Nicola, Mark Miller, Eugenio Moggi, and Davide Sangiorgi. Surely our future work on this subject will be strongly influenced by this useful interaction.

References

1. Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with Explicit Code Mobility. In *OOPSLA '05 (to appear), the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM Press, 2005.
2. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Simplifying Types in a Calculus for Java Exceptions. Technical report, DISI - Università di Genova, 2002.
3. Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
4. Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Typechecking Safe Process Synchronization. In *FGUC 2004*, ENTCS, 2004.
5. Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *To appear in JFP*, 2005.
6. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA'98*, pages 183–200. ACM Press, 1998.
7. Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, D. Le Métayer Editor.
8. Mariangiola Dezani, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A Distributed Object Oriented Language with Session Types. In *Preliminary Proceedings of TGC'05*, 2005. http://www.cs.unibo.it/ sangio/TGC05/.
9. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
10. Sophia Drossopoulou. Advanced Issues in Object Oriented Languages Course Notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.
11. Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In *ESOP'03*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, 2003.
12. Manuel Fahndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI '02*, pages 13–24. ACM Press, 2002.
13. Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.
14. Simon Gay, Vasco T. Vasconcelos, and António Ravara. Session Types for Inter-process Communication. TR 2003–133, Department of Computing, University of Glasgow, March 2003.

15. Kohei Honda. Types for Dyadic Interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
16. Kohei Honda. Composing Processes. In *POPL'96*, pages 344–357. ACM Press, 1996.
17. Kohei Honda, Vasco T. 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-Verlag, 1998.
18. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
19. Naoki Kobayashi, Benjamin Pierce, and David Turner. Linear Types and π-calculus. In *POPL'96*, pages 358–371. ACM Press, 1996.
20. Sun Microsystems Inc. Java home page. http://www.javasoft.com/.
21. Sun Microsystems Inc. The Java Tutorial: All About Sockets. http://java.sun.com/docs/books/tutorial/networking/sockets/.
22. Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.
23. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
24. Benjamin C. Pierce and Davide Sangiorgi. Typing and Subtyping for Mobile Processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.
25. 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-Verlag, 1994.
26. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In *Foclasa 2002*, volume 68(3) of *ENTCS*. Elsevier, 2002.
27. Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session Types for Functional Multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
28. Jan Vitek and Giuseppe Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, volume 1686 of *LNCS*, pages 47–77, 1999.
29. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.