A Distributed Object-Oriented Language with Session Types*

January 27, 2005

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

[1] Dipartimento di Informatica, Universitá 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 offers a compositional method to statically check safety of communication protocols.

1 Introduction

In distributed systems, physically separated (and potentially mobile) computational entities cooperate or complete by passing code and data to one another. Existing theoretical foundations, which have been successful in sequential programming (as structured programming [8] and type disciplines for programming languages [20]) 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 [17], naturally extending the sequential message-passing-oriented paradigm of objects. In current practice, however, code is often written in terms of bare socket-based communications

* Work partially supported by the Royal Academy, the EU project DART, and the EPSRC.

[18]; 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 [12], can specify protocols of communication by describing the sequence and types of entities read on a channel. For example, the session type `add.!int.?bool.end` expresses that the identifier `add` will be sent, followed by an `int`-value, 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, 11, 12, 14, 21], for CORBA [22], for a multi-threaded functional language [23], and recently, for a W3C standard description language for web services called Choreography Description Language (CDL) [25]. To our knowledge, the integration of session types into an object-oriented language (even a small, core calculus, as in [3, 9, 15]) has not been attempted so far.

The present paper argues 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 [14, 21] and linear interactions [13, 16], 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 [14, 21] can be compensated by methods, a natural notion of branching in objects. Subtyping on the branching types [11] is, then, formalised through a standard inheritance mechanism. 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, 10], code or agent mobility [1, 7, 24], polymorphism [6, 15, 23], recursive types [22] or correspondence assertions [4, 5]. We believe that the inclusion of such features into $\mathcal{L}_{doos}$ is possible, albeit not necessarily trivial.

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. For the referees' convenience only, the Appendix contains complete definitions and proofs.

2 Example

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

The seller sends to the agent the price followed by the minimum price he is prepared to accept. The agent begins negotiations by sending 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 accordingly the seller and the buyer. If the offer does not exceed the minimum price, then the agent invites the buyer to lower his minimum price, and the negotiation iterates. Note however, that the agent may now communicate with a different buyer, but he will continue communicating with the same seller.

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

The class `Outcome` represents the outcome of the first round of negotiations; it has a boolean field `success` to indicate whether the negotiation was successful, and the field `ch` which stores the channel on which the negotiation can continue; the type of `ch` is `?float.!boolean.end` to indicate that a float will be received, a boolean will be sent, and then the session will terminate.

```
class Outcome{  boolean success; ?float.!boolean.end ch; }
```

The class `Seller` represents the seller, with fields `price`, and `minPrice` for the asking and the minimum price, `o` for the outcome of the first negotiation, `result` for the outcomes of the successive negotiations and a method for selling. The signature of the method contains the types of two channels, *i.e.* `c1:?float.?float.!Outcome.end`, `c4:?float.!boolean.end`, thus indicating that `c1` will receive two `float`s and then send an `Outcome`, while `c4` will receive a `float` and then send a `boolean`. Notice that the types describe the session from the viewpoint of the `Agent`, which is dual to that of the `Seller`. Channel `c1` will be used to communicate with the agent, and then channel `c4` may be used to continue the communication if the negotiation was not successful. Note that although both channels connect the *same* seller and agent, we need *different* channels because the number and the types of the communications are different the first time and the successive ones.

The method `sell` starts by calculating the asking and minimum prices. It then asks for a connection through a channel `c1` by the `accept c1 ...` statement, which must be matched by a statement `request c1 ...` at another node in the network.

In general `accept c t { e }` represents the creation of a new server-side socket as in the `java.net.ServerSocket` class. Here 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 c t { e' }`. The statement `request` is similar to the creation of a new

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

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 channel is created to connect the two nodes. Execution of `e` and `e'` proceeds concurrently, with all occurrences of `c` replaced by the name of the new channel.

In this example, after the connection on channel `c1` is established, the seller sends the asking and minimum prices along the newly created channel, receives an `Outcome` and stores it in `o`. If `o` reports that the negotiation was unsuccessful, then a new minimum price is calculated and a new connection attempted on the channel stored in `o`. Along this channel, the seller sends the new minimum price, and receives the result of the deal (success or failure). This process is repeated until negotiations succeed.

```
class Seller {

 float price,minPrice; // asking price and minimum price
 Outcome o;            // outcome of current negotiation
 boolean result;       // whether the deal was successful

 void sell( ) c1: ?float.?float.!Outcome.end, c4: ?float.!boolean.end {
    // c1, c4 are the initial, and subsequent channels to agent
    price:= ... ; minPrice:= ... ;
    accept c1 ?float.?float.!Outcome.end {
    // connect with agent
       c1.send(price); c1.send(minPrice); o:=c1.receive;
       result := o.success
       if !(result) then c4:=o.ch;
       while !(result) {
              minPrice:= ... ;
              accept c4 ?float.!boolean.end {
              // connect with same agent
                   c4.send(minPrice);
                   result := c4.receive; } }    } }
}
```

The class `Agent` represents the agent, with fields `price`, `minPrice`, `o`, `result` with the obvious meaning, and offer for the offer made by the buyer.

The signature of the method `mediate` contains the types of two channels, *i.e.* `c1: ?float.?float.!Outcome.end`, `c3: ?float.!boolean.end`, where `c1` will be used on the first round, and `c3` will be used on all subsequent negotiation rounds with the seller. The method `mediate` asks for a connection through channel `c1`, receives the asking and the minimum price along that channel, and then attempts a sale using method `tryToSell` (which returns a boolean). It initialises the `o` object with that value and the channel `c3`, and sends `o` along `c1` back to the seller. Thus, it ensures that all further communications by the buyer using his channel `c4` will be with the current agent. If the negotiation was not successful, then the agent reads a new minimum price on `c3`, tries to sell again, and sends the result along `c3`; the process is repeated until the negotiation is successful.

The method `tryToSell` connects with some buyer (possibly a different one for each call), sends the price and receives an offer. The agent then decides on behalf of the buyer whether the offer was successful, and tells the buyer through `c2`.

```
class Agent extends Object {

 float price, minPrice; // seller' asking and minimum price
 float offer;           // the offer made by the buyer
 Outcome o;             // outcome of negotiation
 boolean result;        // whether the deal was successful

 void mediate() c1: ?float.?float.!Outcome.end, c3: ?float.!boolean.end {
   // c1 for first round the seller; c2 for subsequent rounds
   request c1 ?float.?float.!Outcome.end {
   // connect with a seller
      price := c1.receive; minPrice := c1.receive;
      result := tryToSell();
      o := new Outcome; o.success := result; o.ch := c3;
      c1.send(o);
      while !(result) {
          // the new session is with the seller from before
          request c3 ?float.!boolean.end{
              minPrice := c3.receive;
              result:= tryToSell();
              c3.send( result ); } } }}

  boolean tryToSell() c2: !float.?float.!boolean.end {
      request c2 : !float.?float.!boolean.end {
      // connect with a buyer
        c2.send(price); offer := c2.receive; c2.send( offer > minPrice);
       return (offer>minPrice); } }
}
```

The class `Buyer` represents the buyer. In method `buy` he creates a connection with an agent, along which it reads the asking price. Along this channel, he sends his offer and receives a boolean indicating whether the seller's agent accepted the bid.

```
class Buyer extends Object {
   float price; // seller's asking price
   float offer; // offer made by the buyer

   void buy() c2: !float.?float.!boolean.end {
      accept c2 !float.?float.!boolean.end {
      // connect with an agent
         price := c2.receive; offer:=....; c2.send(offer)
         if c2.receive then .... else ... } }
}
```

Our example demonstrates session types and sessions as first class values, *e.g.* sending of objects containing session channels (in method `mediate`), and assigning session values to session variables (in method `sell`). It also makes use of higher order session types, as *e.g.* in `?float.?float.!Outcome.end`. It uses nesting of sessions (*e.g.* the session `c2` is "nested" within session `c1`).

Notice also, that although the type system does not allow for recursive types, through the use of iteration, we are able to type iterative sessions—here what would amount to a session like `?float.?float.!Outcome.(?float.?boolean)*.end`, where $*$ would mean repetition—therefore we do not need such a session type.

3 A Distributed Object Oriented Language with Sessions

3.1 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 [15], MJ [3] and DJ [1] (while omitting the new distributed primitives introduced in [1]), augmented with primitives for session communication [14, 23].

(type)	t ::=	$C \mid \mathsf{s}$
(session)	s ::=	$\mathbf{end} \mid ?\mathsf{t}.\mathsf{s} \mid !\mathsf{t}.\mathsf{s}$
(meth sig)	$\mathit{methSig}$ ::=	$\mathsf{t}\ \mathsf{m}\ (\mathsf{t})\ \Sigma$
(class sig)	CSig ::=	$\emptyset \mid \mathsf{CSig}, \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\mathit{field}^*\ \mathit{methSig}^*\}$
(signature)	Σ ::=	$\emptyset \mid \Sigma, \mathsf{c} : \mathsf{s}$
(class table)	$\mathtt{CT}$ ::=	$\emptyset \mid \mathtt{CT}, \mathit{class}$
(class)	class ::=	$\mathbf{class}\ C\ \mathbf{extends}\ C\ \{\mathit{field}^*\ \mathit{meth}^*\}$
(field)	field ::=	$\mathsf{f}\ \mathsf{t}$
(method)	meth ::=	$\mathsf{t}\ \mathsf{m}\ (\mathsf{t}\ \mathsf{x})\ \Sigma\ \{\mathsf{e}\}$
(expression)	e ::=	$\mathsf{e};\ \mathsf{e} \mid \mathbf{new}\ C \mid \mathsf{v} \mid \mathsf{this} \mid \mathsf{x} := \mathsf{e} \mid \mathsf{e}.\mathsf{f} := \mathsf{e} \mid \mathsf{x} \mid \mathsf{e}.\mathsf{f} \mid \mathsf{e}.\mathsf{m}(\mathsf{e})$
		$\mid \mathsf{u}.\mathbf{receive} \mid \mathsf{u}.\mathbf{send}(\mathsf{e}) \mid \mathbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\} \mid \mathbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$
(identifier)	u ::=	$\mathsf{c} \mid \mathsf{x}$
(value)	v ::=	$\mathsf{null} \mid \mathsf{c}$

Fig. 1. User Syntax

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

To prescribe the channel usage in a method, we introduce *effects*, Σ, which map channels to session types. The method declarations are then of the shape

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

where all is standard, except for the addition of the effect.

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 mechanism 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, e, e', is standard except for the two pairs of communication primitives. The first pair is for exchanging values or channels: u.**receive** is for receiving values or channels via u, while u.**send**(e) first evaluates the expression e, then sends its result via u. The other pair is for establishing the connection: **request** $\mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ is for use by clients, and **accept** $\mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ for use by servers. The channel u denotes a shared interaction point which is used for creating new channels. In both **request** ...$\mathsf{s}\{\mathsf{e}\}$, and **accept** ...$\mathsf{s}\{\mathsf{e}\}$, the term $\{\mathsf{e}\}$ 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 s prescribes the communication protocol, which is opened by **request** or **accept**.

3.2 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* $l, m, \ldots$

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

Fig. 2. Runtime Syntax

which can be thought of as IP addresses in a network.

The first line of the grammar extends types with *runtime channel types*, which denote the channel types which are only used for the method invocations. The second and the third lines extend identifiers and values to allow for object identifiers $o, \ldots$, which denote references to instances of classes. We shall frequently write "o-id" for brevity, and we shall call o and c *names*. In the forth line, Error denotes the null-pointer error. The fifth line describes *threads*, 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, 15]. Sequences contain no duplicate names.

In the last line, 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 $\mathtt{CT}$, $N_1 \,||\, N_2$ is the parallel of two networks, and $(\nu u : \mathsf{t})N$ 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, 20]. The reduction relation is given modulo the standard structural equivalence rules of the π-calculus [19], written $\equiv$. We define *multi-step* reductions as: $\twoheadrightarrow \stackrel{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$. We only discuss the more interesting rules. We start by listing the reduction contexts.

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

4.1 Local Expressions

The rules for execution of expressions which correspond to the sequential part of the language are standard [3, 9, 15]. Only the local store is modified, and the rules involve only the local store and the local class table. Below we give the rules for object creation and method invocation.

RC-New

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

RC-LocMeth

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

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 $\mathsf{mbody}(\mathtt{m},C,\mathtt{CT})$ looks up m in the local class table, and returns a pair consisting of the method code and the formal parameter name. The receiver o replaces `this` in the method body and a new store entry x is allocated for the formal parameter v.

4.2 Networks

$\mathcal{L}_{doos}$ 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 [19].

RN-Conf
$$\frac{P, \sigma, \mathtt{CT} \longrightarrow (\nu \vec{\mathsf{u}} : \vec{\mathsf{t}})(P', \sigma', \mathtt{CT})}{l[P, \sigma, \mathtt{CT}] \longrightarrow (\nu \vec{\mathsf{u}} : \vec{\mathsf{t}})(l[P', \sigma', \mathtt{CT}])}$$

RN-Par
$$\frac{N \longrightarrow N'}{N \,\|\, N_0 \longrightarrow N' \,\|\, N_0}$$

RN-Res
$$\frac{N \longrightarrow N'}{(\nu \mathsf{u} : \mathsf{t})N \longrightarrow (\nu \mathsf{u} : \mathsf{t})N'}$$

RN-Str
$$\frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

RN-Fld
$$l_1[E[\mathsf{o}.\mathsf{f}_\mathsf{i}] \,|\, P, \sigma_1, \mathtt{CT}_1] \,\|\, l_2[Q, \sigma_2, \mathtt{CT}_2] \;\longrightarrow\; l_1[E[\mathsf{v}_\mathsf{i}] \,|\, P, \sigma_1, \mathtt{CT}_1] \,\|\, l_2[Q, \sigma_2, \mathtt{CT}_2]$$
$$\sigma_2(\mathsf{o}) = (C, \vec{\mathsf{f}} : \vec{\mathsf{v}})$$

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

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

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

RN-ReqAcc
$$l_1[E_1[\textbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}] \,|\, Q_1, \sigma_1, \mathtt{CT}_1] \,\|\, l_2[E_2[\textbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_2\}] \,|\, Q_2, \sigma_2, \mathtt{CT}_2]$$
$$\longrightarrow$$
$$(\nu \mathsf{c}' : \mathsf{s})(l_1[E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}] \,|\, Q_1], \sigma_1, \mathtt{CT}_1] \,\|\, l_2[E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] \,|\, Q_2, \sigma_2, \mathtt{CT}_2]) \quad \mathsf{c}'\ \text{fresh}$$

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

Fig. 3. Network Reduction

Fig. 3 defines reduction for networks; the first three rules are congruence rules, and the fourth rule is the structural one.

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 $\mathtt{c}$, shared between l_1 and l_2, is created; after that, at l_2 the method call is executed by **RC-LocMeth** (the local method call in § 4.1); the result $\mathtt{v}$ is then safely sent back from l_2 to l_1 via this new private channel $\mathtt{c}$ by **RN-RemCommMeth**; since $\mathtt{c}$ is only used once (*i.e.* it is a linear channel in the sense of [1, 13, 16]), it is finally discarded.

Session communication is formalised by **RN-ReqAcc** and **Rn-RemCommSess**. The rule **RN-ReqAcc** describes opening of sessions: if location l_1 requires and location l_2 accepts an opening channel $\mathtt{c}$, then, a new private channel $\mathtt{c}'$ is created and the opening channel $\mathtt{c}$ is replaced by $\mathtt{c}'$. The freshness of $\mathtt{c}'$ guarantees a private and safe session communication between l_1 and l_2. The rule **RN-RemCommSess** formalises the session communication where sent value $\mathtt{v}$ has the type $\mathtt{t}$; after a series of applications of this rule, the session completes when channel $\mathtt{c}$ has type **end**.

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, instead, 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 effects to both sides of typing judgments, and thus have judgments of 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, respectively, the session types of channels before and after the evaluation of $\mathtt{e}$. We call them the *initial* and *final effect* respectively.

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, 15], and which is judged on the class signature.

5.1 Well-formed class tables

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

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

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

Notice that both the initial and the final effects for typing the method body are empty, and this assures that all send and receive instructions are inside sessions, as we will see in 5.4.

5.2 Expression typing

The rule for typing expression composition illustrates a first use of effects:

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

The final effect Σ' of e typing is used as initial effect for typing e'.

The typing rule for method calls:

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

prescribes the method signature of m in C (determined by the method signature look-up function $\mathsf{msignature}(\mathsf{m}, C)$) be contained in the environment Γ. Further, the effects of e and e' must agree as in rule **TE-Seq**. Finally the type of e' should conforms the method type look-up function $\mathsf{mtype}(\mathsf{m}, C)$ [15].

5.3 Session typing

A main use of effects in typing expressions 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 which is the top of the session type of the current channel c. The typing of send also takes into account the final effect of typing e.

The typing rules for opening sessions are:

TE-Req

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

TE-Acc

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

where $\overline{\mathsf{s}}$ denotes the *dual* session type of s defined inductively by $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{!\mathsf{t}.\mathsf{s}} = ?\mathsf{t}.\overline{\mathsf{s}}$, $\overline{?\mathsf{t}.\mathsf{s}} = !\mathsf{t}.\overline{\mathsf{s}}$.

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 c and one sender waiting to send on c. This is guaranteed by replacing the opening channel u in e by a fresh channel c. The type **end** of c in the final effect of typing e ensures that the session is completed after evaluation of e. Notice that c does not appear in the conclusion.

5.4 Thread and Network typing

Rule **TT-Start** promotes expressions to threads; all channels of the final effect should be completed (i.e. be typed by **end**) and all sessions in the initial effect should conform with the environment.

TT-Start

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

Notice that when all send and receive are inside sessions both the initial and the final effects for typing e can be empty.

Rule **TN-Conf** states that a location is a net 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 are locally available – the latter is guaranteed through the requirement $\mathsf{fcl}(P) \subseteq \mathsf{dom}(\mathtt{CT})$ and the last condition.

TN-Conf

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

6 Type Safety and Communication Safety

As expected, the type system of Section 5 enjoys subject reduction, which can be formalized 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* $\mathtt{e},\sigma,\mathtt{CT} \longrightarrow (\nu\tilde{\mathtt{u}}:\vec{\mathtt{t}}')(\mathtt{e}',\sigma',\mathtt{CT})$ *then* $\Gamma,\tilde{\mathtt{u}}:\tilde{\mathtt{t}'};\Sigma \vdash \mathtt{e}':\mathtt{t}';\Sigma'$ *with* $\mathtt{t}' <: \mathtt{t}$ *and* $\Gamma,\tilde{\mathtt{u}}:\tilde{\mathtt{t}'} \vdash \sigma':\mathtt{ok}$.
- *If* $\Gamma \vdash P:\mathbf{thread}$, *and* $\Gamma \vdash \sigma:\mathtt{ok}$, *and* $\Gamma \vdash \mathtt{CT}:\mathtt{ok}$ *and* $P,\sigma,\mathtt{CT} \longrightarrow (\nu\tilde{\mathtt{u}}:\vec{\mathtt{t}}')(P',\sigma',\mathtt{CT})$ *then* $\Gamma,\tilde{\mathtt{u}}:\tilde{\mathtt{t}'} \vdash P':\mathbf{thread}$ *and* $\Gamma,\tilde{\mathtt{u}}:\tilde{\mathtt{t}'} \vdash \sigma':\mathtt{ok}$.
- *If* $\Gamma \vdash N:\mathbf{net}$, *and* $N \longrightarrow N'$ *then* $\Gamma \vdash N':\mathbf{net}$.

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

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

1. no *connection error* can occur, *i.e.* request and accept on the same channel must have the same session type;
2. no *communication error* can occur, *i.e.* in the same net there cannot be two sends or two receives on the same channel;
3. after a session started *the required communications are always executed in the expected order*;
4. after a session started *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 reducing a well-typed closed network in which all processes are user expressions typed with empty effects and all restrictions are restrictions of session channels. We write $\prod_{0\leq i<n} N_i$ for $N_0 \parallel N_1 \parallel \ldots \parallel N_{n-1}$. We define then an initial network as follows.

Definition 6.2 (Initial Nets). A net N is *initial* if $\vdash N:\mathbf{net}$ is derivable using rule **TT-Start** only with empty effects in the premises, and $N \equiv (\nu\tilde{\mathtt{c}}:\tilde{\mathtt{s}})(\prod_{0\leq i<n} l_i[\mathtt{e}_i,\emptyset,\mathtt{CT}_i])$, where each $\mathtt{e}_i$ is a user expression.

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 formalize points (1) and (2), we add a new constant CError (*connection or communication error*) to the network and the following rules:

$$
\begin{array}{rl}
l_l[E_1[\mathtt{c}.\mathbf{send}(\mathtt{v})] \mid Q_1,\sigma_1,\mathtt{CT}_1] \parallel l_2[E_2[\mathtt{c}.\mathbf{send}(\mathtt{v}')] \mid Q_2,\sigma_2,\mathtt{CT}_2] & \longrightarrow \mathtt{CError} \\
l_l[E_1[\mathtt{c}.\mathbf{receive}] \mid Q_1,\sigma_1,\mathtt{CT}_1] \parallel l_2[E_2[\mathtt{c}.\mathbf{receive}] \mid Q_2,\sigma_2,\mathtt{CT}_2] & \longrightarrow \mathtt{CError} \\
l_l[E_1[\mathbf{request}\,\mathtt{c}\,\mathtt{s}\{\mathtt{e}_1\}] \mid Q_1,\sigma_1,\mathtt{CT}_1] \parallel l_2[E_2[\mathbf{accept}\,\mathtt{c}\,\mathtt{s}'\{\mathtt{e}_2\}] \mid Q_2,\sigma_2,\mathtt{CT}_2] & \longrightarrow \mathtt{CError}
\end{array}
$$

where $\mathtt{s} \neq \mathtt{s}'$.

Theorem 6.3 (CError Freedom). *Suppose that N_0 is an initial net and $N_0 \twoheadrightarrow N$. Then N does not contain* $\mathtt{CError}$*, i.e. there does not exist N' such that $N \equiv N' \,\|\, \mathtt{CError}$.*

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

For points (3) and (4) we have:

Theorem 6.4 (Soundness). *Let N_0 be an initial net, $N_0 \twoheadrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})N$, and $(\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})N \longrightarrow (\nu \mathsf{c} : \mathsf{s})(\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})N' \stackrel{\mathsf{def}}{=} (\nu \mathsf{c} : \mathsf{s})N_1$ by rule* **RN-ReqAcc** *with $\mathsf{s} = \dagger_1 \mathtt{t}_1.\cdots.\dagger_n \mathtt{t}_n.\mathbf{end}$. Then either $(\nu \mathsf{c} : \mathsf{s})N_1$ diverges or*

$$(\nu \mathsf{c} : \dagger_1 \mathtt{t}_1 \cdots .\dagger_n \mathtt{t}_n.\mathbf{end})N_1 \twoheadrightarrow (\nu \mathsf{c} : \dagger_2 \mathtt{t}_2 \cdots .\dagger_n \mathtt{t}_n.\mathbf{end})N_2 \twoheadrightarrow \cdots \twoheadrightarrow (\nu \mathsf{c} : \mathbf{end})N_{n+1}$$

with $\mathsf{c} \notin \mathsf{fn}(N_{n+1})$.

The soundness proof needs a careful analysis of the evaluation order and of the invariants properties of the networks.

The above theorem can be rephrased as follows: if the typable net N reduces to N' by rule **RN-ReqAcc** creating the new channel c with type $\dagger_1 \mathtt{t}_1.\cdots.\dagger_n \mathtt{t}_n.\mathbf{end}$, then either (a) N' diverges; or (b) $N' \twoheadrightarrow (\nu \mathsf{c} : \mathbf{end})N''$ where

- there are exactly n applications of rule **RN-RemCommSess** on channel c;
- in the i-th application, the request process sends a value of type $\mathtt{t}_i$ if $\dagger_i = !$ or receives a value of type $\mathtt{t}_i$ if $\dagger_i = ?$; and
- c does not occur in N''

Finally we get:

Theorem 6.5 (Completion of Sessions). *Suppose N_0 is an initial net, $N_0 \twoheadrightarrow N \equiv (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}}) \prod_{0 \le i < n} l_i[\mathsf{e}_i, \sigma_i, \mathtt{CT}_i]$ and N is irreducible. Then either all e_i are values $(0 \le i < n)$ or there is j $(0 \le j < n)$ such that $e_j \in \{\mathtt{Error}, E[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\,\{\mathsf{e}'\}], E[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\,\{\mathsf{e}'\}]\}$.*

7 Conclusions, Discussions, and Further Work

Session types have been successfully applied to theoretical settings such as the π-calculus [4, 5, 11, 12, 14, 21], a multi-threaded functional language [23], and also to practical settings such as CORBA [22] and a web-services description language [25]. With $\mathcal{L}_{doos}$ we aimed to link language development to the 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 aims when designing $\mathcal{L}_{doos}$ were to restrict the number of novel features to be introduced into the object-oriented language (we added only four primitives for standard session communication in the user syntax), and to design 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 provide a disciplined programming style for communication that is natural for programmers from the object-oriented community.

The session types of $\mathcal{L}_{doos}$ are limited, and in particular do not support branching or recursive types [4, 5, 11, 12, 14, 21]. However, the richness of the object-oriented language constructs allows $\mathcal{L}_{doos}$ to express examples that would require more advanced session types in other settings. For instance, the iteration of the communication between buyer and agent in our example in Section 2 could be written without branching or recursive types, because choice and iteration are available in the underlying control and data structures of objects.

However, this approach is only partially satisfying: although the programs *could* be written in $\mathcal{L}_{doos}$, the information expressed through the current type structures is more limited. Taking the seller/agent communication as an example, in the current version of $\mathcal{L}_{doos}$ we consider that `mediate` uses *two* channels, `c1` and `c3`, while in actual fact it only uses *one*. With branching types, we would have been able to describe that `c1` and `c3` coincide. Furthermore, the agent and the seller iterate the request-accept pair on channel `c3`, thus requiring more synchronisation than necessary. It would have been more natural, and more efficient, if the channel request-accept took place *before* the iteration; without recursion in the type system this cannot be expressed.[4]

In further work we plan to consider extensions to our type system, as in [4, 5, 23], which would give more information and control about session behaviours and would allow a more liberal use of channels.

Furthermore, we plan to 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.?Outcome.end` would be the session type of the seller), in $\mathcal{L}_{doos}$ they are not included (*e.g.* `!float.!float.?Outcome.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. When writing more advanced examples, however, we discovered the need for branching types and for including names of functions into the session types. Thus we plan to investigate the alternative design as well, and compare it with our current design through a sequence of case studies.

[4] In more detail, with branching types, and some weak form of dependent and recursive types, we would be able to express a type like the following:

```
type AgentToSeller == ?float.?float.( !true.end OR !false.?float)*
```

which accurately expresses the communication between seller and agent, *i.e.*that two float s are received, and then either the value true is sent and the communication closes, or, the value false is sent, a float is received, and the process iterates until the value true is sent. Then, using this type, the body of the method mediate would be

```
void mediate() c1 : AgentToSeller  // c1 for first and all subsequent rounds with seller
   request c1 AgentToSeller { // connect with a seller
   price := c1.receive; minPrice := c1.receive;
   result := tryToSell();
   while!(c1.send(result)){
      minPrice := c1.request;
      result := tryToSell(); }} }
```

and thus we would be able to expresses that the first round and all subsequent rounds take place on the *same* channel. The body for the method sell from class Seller would be modified in a similar way.

References

1. Alexander Ahern and Nobuko Yoshida. Formal Analysis of a Distributed Object-Oriented Language and Runtime. http://www.doc.ic.ac.uk/~aja/dcbl.html, December 2004.
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 *Proc. 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 *Proceedings of OOPSLA 98*, October 1998.
7. Luca Cardelli and Andy Gordon. Mobile ambients. *TCS*, 240, 2000.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. Sophia Drossopoulou. Advanced issues in object oriented languages course notes. http://www.doc.ic.ac.uk/~scd/Teaching/AdvOO.html.
10. Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *12th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, April 2003.
11. Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Proc. of ESOP'99*, number 1576 in LNCS, pages 74–90. Springer-Verlag, 1999.
12. Kohei Honda. Types for dyadic interaction. In *Proc. CONCUR '93*, number 715 in LNCS, pages 509–523. Springer, 1993.
13. Kohei Honda. Composing processes. In *Proceedings of POPL'96*, pages 344–357, 1996.
14. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138. Springer-Verlag, 1998.
15. 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.
16. Naoki Kobayashi, Benjamin Pierce, and David Turner. Linear types and π-calculus. In *Proceedings of POPL'96*, pages 358–371, 1996.
17. Sun Microsystems Inc. Java home page. http://www.javasoft.com/.
18. Sun Microsystems Inc. The Java Tutorial: All About Sockets. http://java.sun.com/docs/books/tutorial/networking/sockets/.
19. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
20. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
21. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proc. of PARLE'94*, number 817 in LNCS, pages 398–413. Springer-Verlag, 1994.
22. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures (Foclasa 2002)*, volume 68(3) of *ENTCS*. Elsevier, August 2002.

23. Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511. Springer-Verlag, 2004.
24. Jan Vitek and Giuseppe Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, 1999.
25. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.

Appendix

The Appendix is structured as follows:

- Appendix A contains the definitions of set of location names, set of free class names and lookup functions.
- Appendix B lists the structural rules and the reduction rules.
- Appendix C lists the typing rules.
- Appendix D proves the subject reduction theorem, Theorem 6.1 in Section 6.
- Appendix E proves Theorems 6.4 (soundness) and 6.5 (session completeness) in Section 6.

Many definitions and all the rules are given inside figures.

A Appendix: Auxiliary Definitions

$$\mathsf{loc}(\mathbf{0}) = \emptyset, \quad \mathsf{loc}(l[P,\sigma,\mathtt{CT}]) = \{l\}, \quad \mathsf{loc}(N_1 \,||\, N_2) = \mathsf{loc}(N_1) \cup \mathsf{loc}(N_2), \mathsf{loc}((\nu \mathsf{u} : \mathsf{t})N) = \mathsf{loc}(N)$$

$$\mathsf{fcl}(\mathsf{v}) = \mathsf{fcl}(\mathsf{this}) = \mathsf{fcl}(\mathsf{x}) = \mathsf{fcl}(\mathsf{u}\,.\mathbf{receive}) = \emptyset \qquad \mathsf{fcl}(\ \mathbf{new}\ C) = \{C\}$$

$$\mathsf{fcl}(\mathsf{x}{:=}\mathsf{e}) = \mathsf{fcl}(\mathsf{e}.\mathsf{f}) = \mathsf{fcl}(\mathsf{u}\,.\mathbf{send}\,(\mathsf{e})) = \mathsf{fcl}(\mathbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}) = \mathsf{fcl}(\mathbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}) = \mathsf{fcl}(\mathsf{e})$$

$$\mathsf{fcl}(\mathsf{e};\mathsf{e}') = \mathsf{fcl}(\mathsf{e}.\mathsf{f}{:=}\mathsf{e}') = \mathsf{fcl}(\mathsf{e}\,.\,\mathsf{m}\ (\ \mathsf{e}')) = \mathsf{fcl}(\mathsf{e}) \cup \mathsf{fcl}(\mathsf{e}')$$

$$\mathsf{fcl}(P \,|\, P') = \mathsf{fcl}(P) \cup \mathsf{fcl}(P') \qquad \mathsf{fcl}((\nu \mathsf{u} : \mathsf{t})P) = \mathsf{fcl}(P)$$

$$\mathsf{fcl}(C, \mathtt{CT}) = \{\mathsf{fcl}(\mathsf{e}) \mid \mathsf{mbody}(\mathsf{m}, C, \mathtt{CT}) = (\mathsf{x}, \mathsf{e})\}$$

Fig. 4. Location Names and Free Class Names

Field lookup

$$\mathsf{fields}(\mathit{Object}) = \bullet \qquad \frac{\mathsf{fields}(D) = \tilde{\mathsf{f}}'\,\tilde{\mathsf{t}}' \quad \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig}}{\mathsf{fields}(C) = \tilde{\mathsf{f}}'\,\tilde{\mathsf{t}}', \tilde{\mathsf{f}}\,\tilde{\mathsf{t}}}$$

Method lookup

$$\mathsf{methods}(\mathit{Object}) = \bullet \qquad \frac{\mathsf{methods}(D) = \tilde{MS}' \quad \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig}}{\mathsf{methods}(C) = \tilde{MS}', \tilde{MS}}$$

Method signature lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig} \quad \mathsf{t}_2\,\mathsf{m}(\mathsf{t}_1)\Sigma \in \tilde{MS}}{\mathsf{msignature}(\mathsf{m}, C) = \Sigma}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig} \quad \mathsf{m} \notin \tilde{MS}}{\mathsf{msignature}(\mathsf{m}, C) = \mathsf{msignature}(\mathsf{m}, D)}$$

Method type lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig} \quad \mathsf{t}_2\,\mathsf{m}(\mathsf{t}_1)\Sigma \in \tilde{MS}}{\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1 \to \mathsf{t}_2}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{MS}\} \in \mathsf{CSig} \quad \mathsf{m} \notin \tilde{MS}}{\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)}$$

Method body lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{M}\} \in \mathtt{CT} \quad \mathsf{t}_2\,\mathsf{m}(\mathsf{t}_1\,\mathsf{x})\Sigma\{\mathsf{e}\} \in \tilde{M}}{\mathsf{mbody}(\mathsf{m}, C, \mathtt{CT}) = (\mathsf{x}, \mathsf{e})} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\,\vec{M}\} \in \mathtt{CT} \quad \mathsf{m} \notin \tilde{M}}{\mathsf{mbody}(\mathsf{m}, C, \mathtt{CT}) = \mathsf{mbody}(\mathtt{m}, D, \mathtt{CT})}$$

Fig. 5. Lookup Functions

B Appendix: Operational Semantics

Threads

$$P \,|\, \mathsf{null} \equiv P \qquad P \,|\, P_0 \equiv P_0 \,|\, P \qquad P \,|\, (P_0 \,|\, P_1) \equiv (P \,|\, P_0) \,|\, P_1$$

Networks

$$N \,\|\, \mathbf{0} \equiv N \qquad N \,\|\, N_0 \equiv N_0 \,\|\, N \qquad N \,\|\, (N_0 \,\|\, N_1) \equiv (N \,\|\, N_0) \,\|\, N_1$$

$$(\nu \mathsf{u} : \mathsf{t})\mathbf{0} \equiv \mathbf{0} \qquad (\nu \mathsf{u} : \mathsf{t})(N \,\|\, N_0) \equiv (\nu \mathsf{u} : \mathsf{t})N \,\|\, N_0 \quad \mathsf{u} \notin \mathsf{fnv}(N_0)$$

$$(\nu \mathsf{u} : \mathsf{t})(\nu \mathsf{u}' : \mathsf{t}')N \equiv (\nu \mathsf{u}' : \mathsf{t}')(\nu \mathsf{u} : \mathsf{t})N \qquad l[(\nu \mathsf{u} : \mathsf{t})(P, \sigma, \mathtt{CT})] \equiv (\nu \mathsf{u} : \mathsf{t})l[P, \sigma, \mathtt{CT}]$$

Fig. 6. Structural Equivalence

RC-NullFld
$\mathtt{null}.\mathsf{f}, \sigma, \mathtt{CT} \longrightarrow \mathtt{Error}, \sigma, \mathtt{CT}$

RC-NullFldAss
$\mathtt{null}.\mathsf{f} := \mathsf{v}, \sigma, \mathtt{CT} \longrightarrow \mathtt{Error}, \sigma, \mathtt{CT}$

RC-NullMeth
$\mathtt{null}.\mathsf{m}(\mathsf{v}), \sigma, \mathtt{CT} \longrightarrow \mathtt{Error}, \sigma, \mathtt{CT}$

RC-NullProp
$E[\mathtt{Error}], \sigma, \mathtt{CT} \longrightarrow \mathtt{Error}, \sigma, \mathtt{CT}$

Fig. 7. Null Pointer Exceptions

RC-Var
$$\mathsf{x}, \sigma, \mathtt{CT} \longrightarrow \sigma(\mathsf{x}), \sigma, \mathtt{CT}$$

RC-Fld
$$\frac{\sigma(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})}{\mathsf{o}.\mathsf{f}_i, \sigma, \mathtt{CT} \longrightarrow \mathsf{v}_i, \sigma, \mathtt{CT}}$$

RC-Seq
$$\frac{\mathsf{e}_1, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(\mathsf{v}, \sigma', \mathtt{CT})}{\mathsf{e}_1; \mathsf{e}_2, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(\mathsf{e}_2, \sigma', \mathtt{CT})} \quad \tilde{\mathsf{u}} \notin \mathsf{fnv}(\mathsf{e}_2)$$

RC-Ass
$$\mathsf{x} := \mathsf{v}, \sigma, \mathtt{CT} \longrightarrow \mathsf{v}, \sigma[\mathsf{x} \mapsto \mathsf{v}], \mathtt{CT}$$

RC-FldAss
$$\frac{\sigma' = \sigma[\mathsf{o} \mapsto \sigma(\mathsf{o})[\mathsf{f} \mapsto \mathsf{v}]]}{\mathsf{o}.\mathsf{f} := \mathsf{v}, \sigma, \mathtt{CT} \longrightarrow \mathsf{v}, \sigma', \mathtt{CT}} \quad \mathsf{o} \in \mathsf{dom}(\sigma)$$

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

RC-Cong
$$\frac{\mathsf{e}, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(\mathsf{e}', \sigma', \mathtt{CT})}{E[\mathsf{e}], \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(E[\mathsf{e}'], \sigma', \mathtt{CT})} \quad \tilde{\mathsf{u}} \notin \mathsf{fnv}(E)$$

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

Fig. 8. Expression Reduction

RC-Res
$$\frac{(\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(P, \sigma, \mathtt{CT}) \longrightarrow (\nu \tilde{\mathsf{u}}' : \vec{\mathsf{t}}')(P', \sigma', \mathtt{CT})}{(\nu \mathsf{u}\,\tilde{\mathsf{u}} : \mathsf{t}\,\tilde{\mathsf{t}})(P, \sigma, \mathtt{CT}) \longrightarrow (\nu \mathsf{u}\,\tilde{\mathsf{u}}' : \mathsf{t}\,\tilde{\mathsf{t}}')(P', \sigma', \mathtt{CT})}$$

RC-Str
$$\frac{P \equiv P_0 \quad P_0' \equiv P' \quad P_0, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(P_0', \sigma', \mathtt{CT})}{P, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(P', \sigma', \mathtt{CT})}$$

RC-Par
$$\frac{P_1, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(P_1', \sigma', \mathtt{CT})}{P_1 \,|\, P_2, \sigma, \mathtt{CT} \longrightarrow (\nu \tilde{\mathsf{u}} : \vec{\mathsf{t}})(P_1' \,|\, P_2, \sigma', \mathtt{CT})} \quad \tilde{\mathsf{u}} \notin \mathsf{fnv}(P_2)$$

Fig. 9. Thread Reduction

RN-Conf
$$\frac{P,\sigma,\mathtt{CT} \longrightarrow (\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})(P',\sigma',\mathtt{CT})}{l[P,\sigma,\mathtt{CT}] \longrightarrow (\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})(l[P',\sigma',\mathtt{CT}])}$$

RN-Par
$$\frac{N \longrightarrow N'}{N \,\|\, N_0 \longrightarrow N' \,\|\, N_0}$$

RN-Res
$$\frac{N \longrightarrow N'}{(\nu\mathsf{u}:\mathsf{t})N \longrightarrow (\nu\mathsf{u}:\mathsf{t})N'}$$

RN-Str
$$\frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

RN-Fld
$$l_I[E[\mathsf{o}.\mathsf{f}_\mathsf{i}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,\|\, l_2[Q,\sigma_2,\mathtt{CT}_2] \;\longrightarrow\; l_I[E[\mathsf{v}_\mathsf{i}] \,|\, P,\sigma_1,\mathtt{CT}_1] \,\|\, l_2[Q,\sigma_2,\mathtt{CT}_2]$$
$$\sigma_2(\mathsf{o}) = (C,\tilde{\mathsf{f}}:\tilde{\mathsf{v}})$$

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

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

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

RN-ReqAcc
$$l_I[E_1[\mathbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}] \,|\, Q_1,\sigma_1,\mathtt{CT}_1] \,\|\, l_2[E_2[\mathbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_2\}] \,|\, Q_2,\sigma_2,\mathtt{CT}_2]$$
$$\longrightarrow$$
$$(\nu\mathsf{c}':\mathsf{s})(l_I[E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}] \,|\, Q_1],\sigma_1,\mathtt{CT}_1] \,\|\, l_2[E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] \,|\, Q_2,\sigma_2,\mathtt{CT}_2]) \quad \mathsf{c}'\text{ fresh}$$

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

Fig. 10. Network Reduction

C Appendix: Typing Rules

$$\frac{C \in dom(\mathsf{CSig})}{C <: C} \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\,\vec{\mathsf{t}}\ \vec{MS}\} \in \mathsf{CSig}}{C <: D}$$

Fig. 11. Subtyping

TE-Var

$$\Gamma, \mathsf{x} : \mathsf{t}; \Sigma \vdash \mathsf{x} : \mathsf{t}; \Sigma$$

TE-This

$$\Gamma, \mathsf{this} : C; \Sigma \vdash \mathsf{this} : C; \Sigma$$

TE-Fld

$$\frac{\Gamma; \Sigma \vdash \mathsf{e} : C; \Sigma'}{\Gamma; \Sigma \vdash \mathsf{e.f} : \mathsf{t}; \Sigma'} \ \mathsf{f\,t} \in \mathsf{fields}(C)$$

TE-Seq

$$\frac{\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma' \qquad \Gamma; \Sigma' \vdash \mathsf{e}' : \mathsf{t}'; \Sigma''}{\Gamma; \Sigma \vdash \mathsf{e}; \mathsf{e}' : \mathsf{t}'; \Sigma''}$$

TE-Ass

$$\frac{\Gamma, \mathsf{x} : \mathsf{t}'; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma'}{\Gamma, \mathsf{x} : \mathsf{t}'; \Sigma \vdash \mathsf{x} := \mathsf{e} : \mathsf{t}; \Sigma'} \ \mathsf{t} <: \mathsf{t}'$$

TE-FldAss

$$\frac{\Gamma; \Sigma \vdash \mathsf{e.f} : \mathsf{t}; \Sigma' \qquad \Gamma; \Sigma' \vdash \mathsf{e}' : \mathsf{t}'; \Sigma''}{\Gamma; \Sigma \vdash \mathsf{e.f} := \mathsf{e}' : \mathsf{t}'; \Sigma''} \ \mathsf{t}' <: \mathsf{t}$$

TE-New

$$\frac{\vdash C : \mathtt{tp}}{\Gamma; \Sigma \vdash \ \mathbf{new}\ C : C; \Sigma}$$

TE-Meth

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

Fig. 12. Rules for Expressions

TV-Null
$$\frac{\vdash \mathtt{t} : \mathtt{tp}}{\Gamma;\Sigma \vdash \mathtt{null} : \mathsf{t};\Sigma}$$

TV-NullPE
$$\frac{\vdash \mathtt{t} : \mathtt{tp}}{\Gamma;\Sigma \vdash \mathtt{Error} : \mathsf{t};\Sigma}$$

TV-Oid
$$\frac{}{\Gamma, o : C;\Sigma \vdash o : C;\Sigma}$$

Fig. 13. Rules for Values

TE-SessSend
$$\frac{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t};\Sigma', \mathsf{c} : !\mathsf{t}.\mathsf{s}}{\Gamma;\Sigma \vdash \mathsf{c}.\mathbf{send}(\mathsf{e}) : \mathit{Object};\Sigma', \mathsf{c} : \mathsf{s}}$$

TE-SessReceive
$$\frac{}{\Gamma;\Sigma, \mathsf{c} : ?\mathsf{t}.\mathsf{s} \vdash \mathsf{c}.\mathbf{receive} : \mathsf{t};\Sigma, \mathsf{c} : \mathsf{s}}$$

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

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

Fig. 14. Rules for Sessions

TER-ChannelSend
$$\frac{\Gamma, \mathsf{c} : \mathbf{chan}(\mathsf{t});\Sigma \vdash \mathsf{e} : \mathsf{t};\Sigma'}{\Gamma, \mathsf{c} : \mathbf{chan}(\mathsf{t});\Sigma \vdash \mathsf{c}.\mathbf{send}(\mathsf{e}) : \mathit{Object};\Sigma'}$$

TER-ChannelReceive
$$\frac{}{\Gamma, \mathsf{c} : \mathbf{chan}(\mathsf{t});\Sigma \vdash \mathsf{c}.\mathbf{receive} : \mathsf{t};\Sigma}$$

Fig. 15. Rules for Runtime Expressions

TT-Start
$$\frac{\begin{array}{l}\Gamma;\{\mathsf{c}_i : \mathsf{s}_i \mid i \in I\} \vdash \mathsf{e} : \mathsf{t};\{\mathsf{c}_i : \mathbf{end} \mid i \in I\} \\ \forall i \in I.\mathsf{c}_i : \mathsf{s}_i \in \Gamma \vee \mathsf{c}_i : \overline{\mathsf{s}_i} \in \Gamma\end{array}}{\Gamma \vdash \mathsf{e} : \mathbf{thread}}$$

TT-Par
$$\frac{\Gamma \vdash P : \mathbf{thread} \quad \Gamma \vdash P' : \mathbf{thread}}{\Gamma \vdash P \mid P' : \mathbf{thread}}$$

Fig. 16. Rules for Threads

TN-Conf

$$\frac{\Gamma \vdash P : \mathbf{thread} \quad \Gamma \vdash \sigma : \mathtt{ok} \quad \Gamma \vdash \mathtt{CT} : \mathtt{ok} \quad \mathsf{fcl}(P) \subseteq \mathsf{dom}(\mathtt{CT}) \quad \mathit{Comp}(\mathtt{CT})}{\Gamma \vdash l[P, \sigma, \mathtt{CT}] : \mathbf{net}}$$

TN-Nil

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathbf{net}}$$

TN-Par

$$\frac{\mathsf{loc}(N) \cap \mathsf{loc}(N') = \emptyset \quad \Gamma \vdash N : \mathbf{net} \quad \Gamma \vdash N' : \mathbf{net}}{\Gamma \vdash N \| N' : \mathbf{net}}$$

TN-Res

$$\frac{\Gamma, \mathsf{u} : \mathsf{t} \vdash N : \mathbf{net}}{\Gamma \vdash (\nu \mathsf{u} : \mathsf{t}) N : \mathbf{net}}$$

Fig. 17. Rules for Networks

Wf-Class

$$\frac{C \in \mathsf{dom}(\mathsf{CSig})}{\vdash C : \mathtt{tp}}$$

Wf-SessType

$$\frac{}{\vdash \mathsf{s} : \mathtt{tp}}$$

Wf-Channel

$$\frac{\vdash \mathsf{t} : \mathtt{tp}}{\vdash \mathbf{chan}(\mathsf{t}) : \mathtt{tp}}$$

Fig. 18. Well-formed Types

S-Nil

$$\Gamma \vdash \emptyset : \mathtt{ok}$$

S-Var

$$\frac{\Gamma \vdash \sigma : \mathtt{ok} \quad \Gamma \vdash \mathsf{x} : \mathsf{t} \quad \Gamma \vdash \mathsf{v} : \mathsf{t}'}{\Gamma \vdash \sigma \cdot [\mathsf{x} \mapsto \mathsf{v}] : \mathtt{ok}} \quad \begin{array}{l} \mathsf{x} \notin \mathsf{dom}_{\mathsf{v}}(\sigma) \\ \mathsf{t}' <: \mathsf{t} \end{array}$$

S-Oid

$$\frac{\Gamma \vdash \sigma : \mathtt{ok} \quad \Gamma \vdash \mathsf{o} : C \quad \Gamma \vdash \mathsf{v}_i : \mathsf{t}'_i}{\Gamma \vdash \sigma \cdot [\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})] : \mathtt{ok}} \quad \begin{array}{l} \mathsf{o} \notin \mathsf{dom}_{\mathsf{o}}(\sigma) \\ \mathsf{fields}(C) = \tilde{\mathsf{f}}\,\tilde{\mathsf{t}} \\ \mathsf{t}'_i <: \mathsf{t}_i \end{array}$$

Fig. 19. Well-formed Stores

M-ok

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

CS-ok

$$\frac{}{\vdash \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{MS}\} : \mathtt{ok}} \quad \begin{array}{l} \tilde{\mathsf{f}} \notin \mathsf{fields}(D) \\ \mathsf{m} \in \tilde{MS} \implies \mathsf{m} \notin \mathsf{methods}(D) \end{array}$$

CSig-ok

$$\frac{\vdash \mathsf{CSig} : \mathtt{ok} \qquad \vdash \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{MS}\} : \mathtt{ok}}{\vdash \mathsf{CSig}, \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{MS}\} : \mathtt{ok}}$$

C-ok

$$\frac{\Gamma, \mathtt{this} : C \vdash \tilde{M} : \mathtt{ok}\ \mathtt{in}\ C}{\Gamma \vdash \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{M}\} : \mathtt{ok}} \quad \begin{array}{l} \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{MS}\} \in \mathsf{CSig} \\ \mathsf{t}_2\, \mathsf{m}(\mathsf{t}_1\, \mathsf{x})\Sigma\{...\} \in \tilde{M} \quad \text{iff} \quad \mathsf{t}_2\, \mathsf{m}(\mathsf{t}_1)\Sigma \in \tilde{MS} \end{array}$$

CT-Nil

$$\Gamma \vdash \emptyset : \mathtt{ok}$$

CT-ok

$$\frac{\Gamma \vdash \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{M}\} : \mathtt{ok} \qquad \Gamma \vdash \mathtt{CT} : \mathtt{ok}}{\Gamma \vdash \mathtt{CT}, \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\vec{\mathsf{f}}\, \vec{\mathsf{t}}\, \vec{M}\} : \mathtt{ok}}$$

CT-comp

$$\frac{\forall C \in \mathsf{dom}(\mathtt{CT})\ .\ C <: D\ \vee\ D \in \mathsf{fcl}(C, \mathtt{CT}) \implies D \in \mathsf{dom}(\mathtt{CT})}{\mathit{Comp}(\mathtt{CT})}$$

Fig. 20. Well-formed Class Signatures and Tables

D Appendix: Proof of Subject Reduction Theorem

In this appendix, we prove the subject reduction theorem, Theorem 6.1 in Section 6. We first list the basic lemmas together with the definitions used for formalizing these lemmas. We use Δ as short for $\Gamma;\Sigma$.

Definition D.1. 1. A channel name or variable $\mathtt{u}$ is a *synchroniser* in an expression $\mathtt{e}$ if $\mathtt{e}$ contains **request** $\mathtt{u}\ \mathtt{s}\{\mathtt{e}'\}$ or **accept** $\mathtt{u}\ \mathtt{s}\{\mathtt{e}'\}$ for some $\mathtt{s}$ and $\mathtt{e}'$.

2. A channel name $\mathtt{c}$ is a *pseudo-synchroniser* in a derivation $\mathcal{D}$ if $\mathcal{D}$ contains an application of rule **(TE-Req)** or **(TE-Acc)** the subject of whose premise is $\mathtt{e}[\mathtt{c}/\mathtt{u}]$ for some $\mathtt{e}$ and $\mathtt{u}$.
3. The *channel set* of an expression $\mathtt{e}$ is the set of channels names which occur in $\mathtt{e}$ and are not synchronisers in $\mathtt{e}$.
4. A channel name $\mathtt{c}$ is a *proper channel* in a derivation $\mathcal{D}$ if the type of $\mathtt{c}$ in $\mathcal{D}$ is a channel type.
5. A channel name $\mathtt{c}$ is a *session channel* in a derivation $\mathcal{D}$ if the type of $\mathtt{c}$ in $\mathcal{D}$ is a session type and $\mathtt{c}$ is a neither synchroniser nor a pseudo-synchroniser in $\mathcal{D}$.

Definition D.2. 1. A session type $\mathtt{s}$ is a *suffix* of a session type $\mathtt{s}'$ (notation $\mathtt{s} \preceq \mathtt{s}'$) if $\mathtt{s}' = \dagger_1\mathtt{t}_1.\cdots\dagger_n\mathtt{t}_n.\mathtt{s}$ for some $\dagger_i \in \{!,?\}$ $(1 \leq i \leq n)$ and types $\mathtt{t}_1,\ldots,\mathtt{t}_n$ $(n \geq 0)$.

2. A session environment Σ is a *sup-suffix* of a session environment Σ' (notation $\Sigma \preceq^{\supseteq} \Sigma'$) if $\forall \mathtt{c}:\mathtt{s}' \in \Sigma' \exists \mathtt{c}:\mathtt{s} \in \Sigma$ such that $\mathtt{s} \preceq \mathtt{s}'$.
3. A session environment Σ is a *sub-suffix* of a session environment Σ' (notation $\Sigma \preceq^{\subseteq} \Sigma'$) if $\forall \mathtt{c}:\mathtt{s} \in \Sigma \exists \mathtt{c}:\mathtt{s}' \in \Sigma'$ such that $\mathtt{s} \preceq \mathtt{s}'$.
4. A session environment Σ is a *complete suffix* of a session environment Σ' (notation $\Sigma \preceq^{=} \Sigma'$) if $\Sigma \preceq^{\supseteq} \Sigma'$ and $\Sigma \preceq^{\subseteq} \Sigma'$.

Lemma D.3. *1. The channel set of a user expression is empty.*

2. *If $\mathcal{D}$ is a derivation the subject of whose conclusion is $\mathtt{e}$, then the channel set of $\mathtt{e}$ is the union of the sets of proper and session channels of $\mathcal{D}$.*
3. *If $\mathcal{D}$ and $\mathcal{D}'$ are two derivations of the same judgement $\Gamma \vdash \mathtt{e} : \mathbf{thread}$ then the sets of proper and session channels of $\mathcal{D}$ and $\mathcal{D}'$ coincide. Therefore we call them the sets of proper and session channels of $\Gamma \vdash \mathtt{e} : \mathbf{thread}$.*
4. *The set of proper channels of $\Gamma \vdash \mathtt{e} : \mathbf{thread}$ is $\{\mathtt{c} \mid \mathtt{c} : \mathbf{chan}(\mathtt{t}) \in \Gamma\}$. The sets of session channels of $\Gamma \vdash \mathtt{e} : \mathbf{thread}$ is a subset of $\{\mathtt{c} \mid \mathtt{c} : \mathtt{s} \in \Gamma\}$.*

Lemma D.4 (Unicity of Sub-derivations). *If $\mathcal{D}$ is a derivation of $\Delta \vdash E[\mathtt{e}] : \mathtt{t};\Sigma$, then there is a unique sub-derivation of $\mathcal{D}$ the subject of whose conclusion is $\mathtt{e}[\tilde{\mathtt{c}}/\tilde{\mathtt{u}}]$ for some $\tilde{\mathtt{c}},\tilde{\mathtt{u}}$ such that all $\tilde{\mathtt{c}}$ are pseudo-synchronisers in $\mathcal{D}$ and all $\tilde{\mathtt{u}}$ are synchronisers in $E[\mathtt{e}]$, and it corresponds to the showed occurrence of $\mathtt{e}$.*

Lemma D.5 (Environments).

1. *If $\Delta \vdash \mathtt{e} : \mathtt{t};\Sigma$ and $\Delta \subseteq \Delta'$ then $\Delta' \vdash \mathtt{e} : \mathtt{t};\Sigma$.*
2. *If $\Gamma \vdash \mathtt{e} : \mathbf{thread}$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash \mathtt{e} : \mathbf{thread}$.*
3. *If $\Gamma;\Sigma \vdash \mathtt{e} : \mathtt{t};\Sigma'$ then $\Sigma' \preceq^{=} \Sigma$.*
4. *If $\mathcal{D}$ is a derivation of $\Gamma;\Sigma \vdash \mathtt{e} : \mathtt{t};\Sigma'$ and $\mathtt{u}$ occurs free in $\mathtt{e}$ then there is a type $\mathtt{t}$ such that $\mathtt{u} : \mathtt{t} \in \Gamma$.*

5. *If* $\Gamma;\Sigma \vdash \mathsf{e}[\tilde{\mathsf{c}}/\tilde{\mathsf{u}}]:\mathsf{t};\Sigma'$ *is a sub-derivation of* $\Gamma';\Sigma'' \vdash E[\mathsf{e}]:\mathsf{t}';\Sigma'''$ *then* $\Gamma' \subseteq \Gamma$, $\Sigma'' \subseteq \Sigma$ *and* $\Sigma''' \preceq^{\subseteq} \Sigma'$.

Lemma D.6 (Substitution Lemma for Expressions).

1. *If* $\Delta \vdash \mathsf{e}:\mathsf{t};\Sigma$ *and* c *is fresh, then* $\Delta[\mathsf{c}/\mathsf{c}'] \vdash \mathsf{e}[\mathsf{c}/\mathsf{c}']:\mathsf{t};\Sigma[\mathsf{c}/\mathsf{c}']$.
2. *If* $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{t};\Sigma'$, $\varphi:\mathsf{t}' \in \Delta$ *with* $\varphi \in \{\mathsf{this},\mathsf{u}\}$, *and* $\Gamma;\emptyset \vdash \mathsf{v}:\mathsf{t}';\emptyset$ *then* $\Delta \vdash \mathsf{e}[\mathsf{v}/\varphi]:\mathsf{t};\Sigma$.
3. *If* $\Gamma \vdash \mathsf{e}:\textbf{thread}$, $\varphi:\mathsf{t}' \in \Gamma$ *with* $\varphi \in \{\mathsf{this},\mathsf{u}\}$, *and* $\Gamma;\emptyset \vdash \mathsf{v}:\mathsf{t}';\emptyset$ *then* $\Gamma \vdash \mathsf{e}[\mathsf{v}/\varphi]:\textbf{thread}$.
4. *If* $\Gamma;\Sigma \vdash \mathsf{e}[\tilde{\mathsf{c}}/\tilde{\mathsf{u}}]:\mathsf{t};\Sigma_1$ *is the sub-derivation of* $\Gamma';\Sigma' \vdash E[\mathsf{e}]:\mathsf{t}';\Sigma'_1$ *typing the showed occurrence of* e, *and* $\Gamma;\Sigma,\tilde{\mathsf{c}}':\tilde{\mathsf{s}} \vdash \mathsf{e}'[\tilde{\mathsf{c}}/\tilde{\mathsf{u}}]:\mathsf{t};\Sigma_1,\tilde{\mathsf{c}}':\tilde{\mathsf{s}}'$ *then* $\Gamma';\Sigma',\tilde{\mathsf{c}}':\tilde{\mathsf{s}} \vdash E[\mathsf{e}']:\mathsf{t}';\Sigma',\tilde{\mathsf{c}}':\tilde{\mathsf{s}'}$.
5. *If* $\Gamma,\mathsf{c}:\dagger\mathsf{t}.\mathsf{s};\Sigma,\mathsf{c}:\dagger'\mathsf{t}.\mathsf{s}' \vdash \mathsf{e}[\tilde{\mathsf{c}}'/\tilde{\mathsf{u}}]:\mathsf{t}_1;\Sigma_1$ *is the sub-derivation of* $\Gamma',\mathsf{c}:\dagger\mathsf{t}.\mathsf{s};\Sigma',\mathsf{c}:\dagger'\mathsf{t}.\mathsf{s}' \vdash E[\mathsf{e}]:\mathsf{t}';\Sigma'_1$ *typing the showed occurrence of* e, $\mathsf{s}' = \begin{cases} \mathsf{s} & \textit{if } \dagger' = \dagger, \\ \overline{\mathsf{s}} & \textit{otherwise}, \end{cases}$ *and* $\Gamma,\mathsf{c}:\mathsf{s};\Sigma,\mathsf{c}:\mathsf{s}' \vdash \mathsf{e}'[\tilde{\mathsf{c}}'/\tilde{\mathsf{u}}]:\mathsf{t}_1;\Sigma_1$ *then* $\Gamma',\mathsf{c}:\mathsf{s};\Sigma',\mathsf{c}:\mathsf{s}' \vdash E[\mathsf{e}']:\mathsf{t}';\Sigma'_1$.

Lemma D.7 (Generation Lemma for Non-Session User Expressions).

1. *If* $\Gamma;\Sigma \vdash \varphi:\mathsf{t};\Sigma'$ *with* $\varphi \in \{\mathsf{this},\mathsf{u}\}$ *then* $\varphi:\mathsf{t} \in \Gamma$.
2. *If* $\Gamma;\Sigma \vdash \mathsf{e}';\mathsf{e}:\mathsf{t};\Sigma'$ *then* $\Gamma;\Sigma \vdash \mathsf{e}':\mathsf{t}';\Sigma''$ *and* $\Gamma;\Sigma'' \vdash \mathsf{e}:\mathsf{t};\Sigma'$ *for some type* t *and some environment* Σ''.
3. *If* $\Gamma;\Sigma \vdash \mathsf{x}:=\mathsf{e}:\mathsf{t};\Sigma'$ *then* $\Gamma;\Sigma \vdash \mathsf{x}:\mathsf{t};\Sigma$ *and* $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{t}';\Sigma'$ *for some type* t' *such that* $\mathsf{t} <: \mathsf{t}'$.
4. *If* $\Delta \vdash \mathsf{e}.\mathsf{f}:\mathsf{t};\Sigma$ *then* $\Delta \vdash \mathsf{e}:C;\Sigma$ *and* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some class* C.
5. *If* $\Gamma;\Sigma \vdash \mathsf{e}'.\mathsf{f}:=\mathsf{e}:\mathsf{t};\Sigma'$ *then* $\Gamma;\Sigma \vdash \mathsf{e}'.\mathsf{f}:\mathsf{t}';\Sigma''$ *and* $\Gamma;\Sigma'' \vdash \mathsf{e}:\mathsf{t};\Sigma'$ *for some type* t' *such that* $\mathsf{t} <: \mathsf{t}'$ *and some environment* Σ''.
6. *If* $\Gamma;\Sigma \vdash \mathsf{e}.\mathsf{m}(\mathsf{e}'):\mathsf{t};\Sigma'$ *then* $\Gamma;\Sigma \vdash \mathsf{e}':\mathsf{t}';\Sigma''$ *and* $\Gamma;\Sigma'' \vdash \mathsf{e}:C;\Sigma'$ *for some class* C *such that* $\mathsf{msignature}(\mathsf{m},C) \subseteq \Gamma$, $\mathsf{mtype}(\mathtt{m},C) = \mathsf{t}'' \to \mathsf{t}$ *and* $\mathsf{t}' <: \mathsf{t}''$ *and for some environment* Σ''.

Lemma D.8 (Generation Lemma for Session User Expressions).

1. *If* $\Gamma;\Sigma \vdash \mathsf{c}.\textbf{send}(\mathsf{e}):\mathsf{t};\Sigma'$ *then* $\mathsf{t} = \mathit{Object}$, $\Sigma' = \Sigma'',\mathsf{c}:\mathsf{s}$, *and* $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{t}';\Sigma'',\mathsf{c}:!\mathsf{t}'.\mathsf{s}$ *for some type* t' *and environment* Σ''.
2. *If* $\Gamma;\Sigma \vdash \mathsf{c}.\textbf{receive}:\mathsf{t};\Sigma'$ *then* $\Sigma = \Sigma'',\mathsf{c}:?\mathsf{t}.\mathsf{s}$, *and* $\Sigma' = \Sigma'',\mathsf{c}:\mathsf{s}$ *for some environment* Σ''.
3. *If* $\Gamma;\Sigma \vdash \textbf{request}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}:\mathsf{t};\Sigma'$ *then* $\mathsf{u}:\mathsf{s} \in \Gamma$ *and* $\Gamma;\Sigma,\mathsf{c}:\mathsf{s} \vdash \mathsf{e}[\mathsf{c}/\mathsf{u}]:\mathsf{t};\Sigma',\mathsf{c}:\textbf{end}$ *for some session type* s *and channel name* c *such that* $\mathsf{c} \notin \mathsf{fn}(\mathsf{e}) \cup \mathsf{dom}(\Gamma)$.
4. *If* $\Gamma;\Sigma \vdash \textbf{accept}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}:\mathsf{t};\Sigma'$ *then* $\mathsf{u}:\mathsf{s} \in \Gamma$ *and* $\Gamma;\Sigma,\mathsf{c}:\overline{\mathsf{s}} \vdash \mathsf{e}[\mathsf{c}/\mathsf{u}]:\mathsf{t};\Sigma',\mathsf{c}:\textbf{end}$ *for some session type* s *and channel name* c *such that* $\mathsf{c} \notin \mathsf{fn}(\mathsf{e}) \cup \mathsf{dom}(\Gamma)$.

Lemma D.9 (Generation Lemma for Run-Time Expressions).

1. *If* $\Gamma;\Sigma \vdash \mathsf{c}.\textbf{send}(\mathsf{e}):\mathsf{t};\Sigma'$ *then* $\mathsf{t} = \mathit{Object}$, *and one of the following cases hold:*
   (a) $\Sigma' = \Sigma'',\mathsf{c}:\mathsf{s}$, *and* $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{t}';\Sigma'',\mathsf{c}:!\mathsf{t}'.\mathsf{s}$ *for some type* t' *and environment* Σ'';

(b) $\mathsf{c}:\mathbf{chan}(\mathsf{t}) \in \Gamma$, *and* $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{t}';\Sigma'$ *for some type* t'.

2. *If* $\Gamma;\Sigma \vdash \mathsf{c}.\mathbf{receive}:\mathsf{t};\Sigma'$ *then one of the following cases hold:*
   (a) $\Sigma = \Sigma'', \mathsf{c}:?\mathsf{t}.\mathsf{s}$, *and* $\Sigma' = \Sigma'', \mathsf{c}:\mathsf{s}$ *for some environment* Σ''*;*
   (b) $\mathsf{c}:\mathbf{chan}(\mathsf{t}) \in \Gamma$.

Lemma D.10 (Generation Lemma for Threads).

1. *If* $\{\mathsf{c}_i \mid i \in I\}$ *is set of the session channels of* $\Gamma \vdash \mathsf{e}:\mathbf{thread}$*, then* $\Gamma;\{\mathsf{c}_i:\mathsf{s}_i \mid i \in I\} \vdash \mathsf{e}:\mathsf{t};\{\mathsf{c}_i:\mathbf{end} \mid i \in I\}$ *for some session types* s_i *(*$i \in I$*), and type* t *such that* $\forall i \in I.\mathsf{c}_i:\mathsf{s}_i \in \Gamma \vee \mathsf{c}_i:\overline{\mathsf{s}}_i \in \Gamma$.
2. *If* $\Gamma \vdash P \mid P':\mathbf{thread}$ *then* $\Gamma \vdash P:\mathbf{thread}$ *and* $\Gamma \vdash P':\mathbf{thread}$.
3. *If* $\Gamma \vdash (\nu \mathsf{u}:\mathsf{t})(P):\mathbf{thread}$ *then* $\Gamma, \mathsf{u}:\mathsf{t} \vdash P:\mathbf{thread}$.

Lemma D.11 (Generation Lemma for Nets).

1. *If* $\Gamma \vdash l[P,\sigma,\mathtt{CT}]:\mathbf{net}$ *then* $\Gamma \vdash P:\mathbf{thread}$, $\Gamma \vdash \sigma:\mathtt{ok}$, $\Gamma \vdash \mathtt{CT}:\mathtt{ok}$, $\mathsf{fcl}(P) \subseteq \mathsf{dom}(\mathtt{CT})$, *and* $Comp(\mathtt{CT})$.
2. *If* $\Gamma \vdash N \| N':\mathbf{net}$ *then* $\Gamma \vdash N:\mathbf{net}$ *and* $\Gamma \vdash N':\mathbf{net}$.
3. *If* $\Gamma \vdash (\nu \mathsf{u}:\mathsf{t})N:\mathbf{net}$ *then* $\Gamma, \mathsf{u}:\mathsf{t} \vdash N:\mathbf{net}$.

The subject reduction for expressions and threads are straightforward from the above generation lemmas. Hence we only consider the networks.

Theorem D.12 (Subject Reduction for Nets). *If* $\Gamma \vdash N:\mathbf{net}$*, and* $N \longrightarrow N'$ *then* $\Gamma \vdash N':\mathbf{net}$.

Proof. By induction on $\longrightarrow$.
Rule **RN-ReqAcc**. In this case

$$N \equiv l_l[E_1[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}] \mid Q_1, \sigma_1, \mathtt{CT}_1] \,\|\, l_2[E_2[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}] \mid Q_2, \sigma_2, \mathtt{CT}_2],$$

$$N' \equiv (\nu \mathsf{c}':\mathsf{s})(l_l[E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}] \mid Q_1], \sigma_1, \mathtt{CT}_1] \,\|\, l_2[E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] \mid Q_2, \sigma_2, \mathtt{CT}_2])$$

where c' fresh.

By Lemma D.11(1) and (2) $\Gamma \vdash N:\mathbf{net}$ implies $\Gamma \vdash E_1[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}] \mid Q_1:\mathbf{thread}$, $\Gamma \vdash E_2[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}] \mid Q_2:\mathbf{thread}$, $\Gamma \vdash \sigma_1:\mathtt{ok}$, $\Gamma \vdash \sigma_2:\mathtt{ok}$, $\Gamma \vdash \mathtt{CT}_1:\mathtt{ok}$, $\Gamma \vdash \mathtt{CT}_2:\mathtt{ok}$.

If $\{\mathsf{a}_i \mid i \in I\}$ is the set of session channels of $\Gamma \vdash E_1[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}]:\mathbf{thread}$, $\{\mathsf{b}_j \mid j \in J\}$ is the set of session channels of $\Gamma \vdash E_2[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}]:\mathbf{thread}$, by Lemma D.10(1) and (2) $\Gamma \vdash E_1[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}] \mid Q_1:\mathbf{thread}$ and $\Gamma \vdash E_2[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}] \mid Q_2:\mathbf{thread}$ imply

$$\Gamma;\{\mathsf{a}_i:\mathsf{p}_i \mid i \in I\} \vdash E_1[\mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}]:\mathsf{t}_1;\{\mathsf{a}_i:\mathbf{end} \mid i \in I\}$$

$$\Gamma;\{\mathsf{b}_j:\mathsf{q}_j \mid j \in J\} \vdash E_2[\mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}]:\mathsf{t}_2;\{\mathsf{b}_j:\mathbf{end} \mid j \in J\}$$

for suitable types such that $\forall i \in I.\mathsf{a}_i:\mathsf{p}_i \in \Gamma \vee \mathsf{a}_i:\overline{\mathsf{p}}_i \in \Gamma$ and $\forall j \in J.\mathsf{b}_j:\mathsf{q}_j \in \Gamma \vee \mathsf{b}_j:\overline{\mathsf{q}}_j \in \Gamma$.

By Lemma D.4 there are Γ_1, Σ_1, $\tilde{\mathsf{c}}_1$, $\tilde{\mathsf{u}}_1$, $\mathsf{t}_1{}'$, Σ'_1 and Γ_2, Σ_2, $\tilde{\mathsf{c}}_2$, $\tilde{\mathsf{u}}_2$, $\mathsf{t}_2{}'$, Σ'_2 such that:

$$\Gamma_1;\Sigma_1 \vdash \mathbf{request}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_1\}[\tilde{\mathsf{c}}_1/\tilde{\mathsf{u}}_1]:\mathsf{t}_1{}';\Sigma'_1$$

$$\Gamma_2;\Sigma_2 \vdash \mathbf{accept}\,\mathsf{c}\,\mathsf{s}\{\mathsf{e}_2\}[\tilde{\mathsf{c}}_2/\tilde{\mathsf{u}}_2]:\mathsf{t}_2{}';\Sigma'_2.$$

We get $\mathsf{c}:\mathsf{s}\in\Gamma_1$, $\mathsf{c}:\mathsf{s}\in\Gamma_2$,

$$\Gamma_1;\Sigma_1,\mathsf{c}_1':\mathsf{s}\vdash \mathsf{e}_1[\tilde{\mathsf{c}}_1/\tilde{\mathsf{u}}_1][\mathsf{c}_1'/\mathsf{c}]:\mathsf{t}_1';\Sigma_1',\mathsf{c}_1':\mathbf{end}$$

$$\Gamma_2;\Sigma_2,\mathsf{c}_2':\overline{\mathsf{s}}\vdash \mathsf{e}_2[\tilde{\mathsf{c}}_2/\tilde{\mathsf{u}}_2][\mathsf{c}_2'/\mathsf{c}]:\mathsf{t}_2';\Sigma_2',\mathsf{c}_2':\mathbf{end}$$

by Lemma D.8(3) and (4) where $\mathsf{c}_1'\notin\mathsf{fn}(\mathsf{e}_1[\tilde{\mathsf{c}}_1/\tilde{\mathsf{u}}_1])$, $\mathsf{c}_1'\notin\Gamma_1$, $\mathsf{c}_2'\notin\mathsf{fn}(\mathsf{e}_2[\tilde{\mathsf{c}}_2/\tilde{\mathsf{u}}_2])$, $\mathsf{c}_2'\notin\Gamma_2$.

Being c' fresh Lemma D.6(1) implies

$$\Gamma_1;\Sigma_1,\mathsf{c}':\mathsf{s}\vdash \mathsf{e}_1[\tilde{\mathsf{c}}_1/\tilde{\mathsf{u}}_1][\mathsf{c}'/\mathsf{c}]:\mathsf{t}_1';\Sigma_1',\mathsf{c}':\mathbf{end}$$

$$\Gamma_2;\Sigma_2,\mathsf{c}':\overline{\mathsf{s}}\vdash \mathsf{e}_2[\tilde{\mathsf{c}}_2/\tilde{\mathsf{u}}_2][\mathsf{c}'/\mathsf{c}]:\mathsf{t}_2';\Sigma_2',\mathsf{c}':\mathbf{end}$$

By Lemmas D.5(1) and D.6(4) we get

$$\Gamma,\mathsf{c}':\mathsf{s};\{\mathsf{a}_i:\mathsf{p}_i\mid i\in I\},\mathsf{c}':\mathsf{s}\vdash E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]]:\mathsf{t}_1;\{\mathsf{a}_i:\mathbf{end}\mid i\in I\},\mathsf{c}':\mathbf{end},$$

$$\Gamma,\mathsf{c}':\mathsf{s};\{\mathsf{b}_j:\mathsf{q}_j\mid j\in J\},\mathsf{c}':\overline{\mathsf{s}}\vdash E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]]:\mathsf{t}_2;\{\mathsf{b}_j:\mathbf{end}\mid j\in J\},\mathsf{c}':\mathbf{end}$$

and $\forall i\in I.\mathsf{a}_i:\mathsf{p}_i\in\Gamma\vee\mathsf{a}_i:\overline{\mathsf{p}}_i\in\Gamma$ and $\forall j\in J.\mathsf{b}_j:\mathsf{q}_j\in\Gamma\vee\mathsf{b}_j:\overline{\mathsf{q}}_j\in\Gamma$.

By rule **(TT-Start)** we derive

$$\Gamma,\mathsf{c}':\mathsf{s}\vdash E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]]:\mathbf{thread}$$

and

$$\Gamma,\mathsf{c}':\mathsf{s}\vdash E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]]:\mathbf{thread}$$

so we can easily conclude $\Gamma\vdash N':\mathbf{net}$.

Rule **RN-RemCommSess**. In this case

$$N\equiv(\nu\mathsf{c}:\dagger\mathsf{t}.\mathsf{s})(l_I[E_1[\mathsf{c}.\mathbf{send}(\mathsf{v})]\,|\,Q_1,\sigma_1,\mathtt{CT}_1]\,\|\,l_2[E_2[\mathsf{c}.\mathbf{receive}]\,|\,Q_2,\sigma_2,\mathtt{CT}_2]),$$

$$N'\equiv(\nu\mathsf{c}:\mathsf{s})(l_I[E_1[\mathsf{null}]\,|\,Q_1,\sigma_1,\mathtt{CT}_1]\,\|\,l_2[E_2[\mathsf{v}]\,|\,Q_2,\sigma_2,\mathtt{CT}_2])$$

where $\dagger\in\{?,!\}$.

As in previous case by Lemmas D.11(1), (2) and D.10(1), (2) $\Gamma\vdash N:\mathbf{net}$ implies for suitable channels names and types:

$$\Gamma,\mathsf{c}:\dagger\mathsf{t}.\mathsf{s};\{\mathsf{a}_i:\mathsf{p}_i\mid i\in I\},\mathsf{c}:\dagger_1\mathsf{t}.\mathsf{s}_1\vdash E_1[\mathsf{c}.\mathbf{send}(\mathsf{v})]:\mathsf{t}_1;\{\mathsf{a}_i:\mathbf{end}\mid i\in I\},\mathsf{c}:\mathbf{end}$$

$$\Gamma,\mathsf{c}:\dagger\mathsf{t}.\mathsf{s};\{\mathsf{b}_j:\mathsf{q}_j\mid j\in J\},\mathsf{c}:\dagger_2\mathsf{t}.\mathsf{s}_2\vdash E_2[\mathsf{c}.\mathbf{receive}]:\mathsf{t}_2;\{\mathsf{b}_j:\mathbf{end}\mid j\in J\},\mathsf{c}:\mathbf{end}$$

where $\dagger_1,\dagger_2\in\{?,!\}$, $\mathsf{s}_1=\begin{cases}\mathsf{s} & \text{if } \dagger_1=\dagger,\\ \overline{\mathsf{s}} & \text{otherwise,}\end{cases}$ $\mathsf{s}_2=\begin{cases}\mathsf{s} & \text{if } \dagger_2=\dagger,\\ \overline{\mathsf{s}} & \text{otherwise,}\end{cases}$ $\{\mathsf{a}_i\mid i\in I\},\mathsf{c}$ is the set of session channels of $\Gamma,\mathsf{c}:\dagger\mathsf{t}.\mathsf{s}\vdash E_1[\mathsf{c}.\mathbf{send}(\mathsf{v})]:\mathbf{thread}$, $\{\mathsf{b}_j\mid j\in J\},\mathsf{c}$ is the set of session channels of $\Gamma,\mathsf{c}:\dagger\mathsf{t}.\mathsf{s}\vdash E_2[\mathsf{c}.\mathbf{receive}]:\mathbf{thread}$, and $\forall i\in I.\mathsf{a}_i:\mathsf{p}_i\in\Gamma\vee\mathsf{a}_i:\overline{\mathsf{p}}_i\in\Gamma$ and $\forall j\in J.\mathsf{b}_j:\mathsf{q}_j\in\Gamma\vee\mathsf{b}_j:\overline{\mathsf{q}}_j\in\Gamma$.

By Lemmas D.4 and D.5(5) (remarking that in the current expressions there are no names or variables to replace) there are Γ_1, Σ_1, $\mathsf{t}_1{}'$, Σ_1' and Γ_2, Σ_2, $\mathsf{t}_2{}'$, Σ_2' such that:

$$\Gamma_1, \mathsf{c} : \dagger\mathsf{t}.\mathsf{s}; \Sigma_1, \mathsf{c} : \dagger_1\mathsf{t}.\mathsf{s}_1 \vdash \mathsf{c}.\textbf{send}(\mathsf{v}) : \mathsf{t}_1{}'; \Sigma_1'$$

$$\Gamma_2, \mathsf{c} : \dagger\mathsf{t}.\mathsf{s}; \Sigma_2, \mathsf{c} : \dagger_2\mathsf{t}.\mathsf{s}_2 \vdash \mathsf{c}.\textbf{receive} : \mathsf{t}_2{}'; \Sigma_2'.$$

We get $\mathsf{t}_1{}' = \mathit{Object}$, $\dagger_1 = !$, $\Sigma_1' = \Sigma_1, \mathsf{c} : \mathsf{s}_1$, $\Gamma_1; \Sigma_1 \vdash \mathsf{v} : \mathsf{t}; \Sigma_1$, $\mathsf{t}_2{}' = \mathsf{t}$, $\dagger_2 = ?$, $\Sigma_2' = \Sigma_2, \mathsf{c} : \mathsf{s}_2$, by Lemma D.9(1) and (2) taking into account that the typing of values does not modify the environments. Notice that if $\mathsf{v} = \mathsf{o}$ then $\mathsf{o} : \mathsf{t} \in \Gamma$ by Lemma D.5(4) and then $\mathsf{o} : \mathsf{t} \in \Gamma_2$ by Lemma D.5(5).

By the typing rules of values:

$$\Gamma_1, \mathsf{c} : \mathsf{s}; \Sigma_1' \vdash \mathsf{null} : \mathit{Object}; \Sigma_1'$$
$$\Gamma_2, \mathsf{c} : \mathsf{s}; \Sigma_2' \vdash \mathsf{v} : \mathsf{t}; \Sigma_2'.$$

By Lemma D.6(5) we get

$$\Gamma, \mathsf{c} : \mathsf{s}; \{\mathsf{a}_i : \mathsf{p}_i \mid i \in I\}, \mathsf{c} : \mathsf{s}_1 \vdash E_1[\mathsf{null}] : \mathsf{t}_1; \{\mathsf{a}_i : \textbf{end} \mid i \in I\}, \mathsf{c} : \textbf{end},$$

$$\Gamma, \mathsf{c} : \mathsf{s}; \{\mathsf{b}_j : \mathsf{q}_j \mid j \in J\}, \mathsf{c} : \mathsf{s}_2 \vdash E_2[\mathsf{v}] : \mathsf{t}_2; \{\mathsf{b}_j : \textbf{end} \mid j \in J\}, \mathsf{c} : \textbf{end}$$

and $\forall i \in I.\mathsf{a}_i : \mathsf{p}_i \in \Gamma \vee \mathsf{a}_i : \overline{\mathsf{p}}_i \in \Gamma$ and $\forall j \in J.\mathsf{b}_j : \mathsf{q}_j \in \Gamma \vee \mathsf{b}_j : \overline{\mathsf{q}}_j \in \Gamma$.

By rule **(TT-Start)** we derive

$$\Gamma, \mathsf{c} : \mathsf{s} \vdash E_1[\mathsf{null}] : \textbf{thread}$$

and

$$\Gamma, \mathsf{c} : \mathsf{s} \vdash E_2[\mathsf{v}] : \textbf{thread}$$

so we can easily conclude $\Gamma \vdash N' : \textbf{net}$.

E Appendix: Proofs of Soundness and Completion of Sessions

This section gives the proofs of Theorems 6.4 and 6.5 after introducing auxiliary definitions, lemmas and theorems.

Definition E.1 (Net Values and Stuck Nets).

1. The set $\mathbf{N}_v$ of *net values* is the smallest set such that:
   - $\mathbf{0} \in \mathbf{N}_v$;
   - $l[\mathsf{v}, \sigma, \mathtt{CT}] \in \mathbf{N}_v$;
   - if $N_1 \in \mathbf{N}_v$ and $N_2 \in \mathbf{N}_v$ then $N_1 \,||\, N_2 \in \mathbf{N}_v$;
   - if $N \in \mathbf{N}_v$ then $(\nu \mathsf{u} : \mathsf{t})N \in \mathbf{N}_v$.
   - if $N \in \mathbf{N}_v$ and $N \equiv N' \in \mathbf{N}_v$ then $N' \in \mathbf{N}_v$.
2. A net N is a *stuck net* if $N \not\longrightarrow$, and $N \in \mathbf{N}_v$.

Definition E.2 (Convergence and Divergence).

1. A net N *converges to* N' (notation $N \Downarrow N'$) if $N \longrightarrow N'$ and $N' \not\longrightarrow$.

2. A net N *properly converges* (notation $N \Downarrow$) if $N \Downarrow N'$ for some value N'.
3. A net N *diverges* (notation $N \Uparrow$) if not $N \Downarrow$.

Definition E.3 (Canonical forms). A net is in *canonical form* if it is of the shape $(\nu\tilde{\mathsf{u}} : \vec{\mathsf{t}})N$ where N does not contain restrictions.

Definition E.4 (Session Part). An expression e is a u*-session part* of an expression e' if $\mathsf{e}' = E[\textbf{request}\ \mathsf{u}\ \mathsf{s}\{E'[\mathsf{e}]\}]$ or $\mathsf{e}' = E[\textbf{accept}\ \mathsf{u}\ \mathsf{s}\{E'[\mathsf{e}]\}]$. An expression e is a *session part* of an expression e' if e is a u*-session part* of e' for some u.

Definition E.5 (Evaluation Order). The pre-order relation $\mathsf{e}_1 \preceq_{\mathsf{e}} \mathsf{e}_2$ (to be read: the expression e_1 *is evaluated before* the expression e_2 *inside* the expression e) is the smallest pre-order relation which satisfies:

– $\mathsf{e}_1 \preceq_{\mathsf{e}_2} \mathsf{e}_2$ if e_1 is a sub-expression of e_2 but it is not a session part of e_2;
– $\textbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\} \preceq_{\textbf{request}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\}} \mathsf{e}$;
– $\textbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\} \preceq_{\textbf{accept}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\}} \mathsf{e}$;
– $\mathsf{e}_1 \preceq_{\mathsf{e}_1;\mathsf{e}_2} \mathsf{e}_2$;
– $\mathsf{e}_1 \preceq_{\mathsf{e}_1.\mathsf{f}:=\mathsf{e}_2} \mathsf{e}_2$;
– $\mathsf{e}_1 \preceq_{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)} \mathsf{e}_2$;
– $\mathsf{e}_1 \preceq_{\mathsf{e}} \mathsf{e}_2 \implies \mathsf{e}_1 \preceq_{C[\mathsf{e}]} \mathsf{e}_2$.[5]

Lemma E.6 (Canonical forms). *Each net is structurally equivalent to a canonical form.*

Theorem E.7 (Invariants).
If N is an initial net, and $N \longrightarrow\!\!\!\!\!\rightarrow (\nu\tilde{\mathsf{u}} : \vec{\mathsf{t}})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ then for all i ($0 \le i < n$):

1. $P_i \equiv E[\textbf{ new } C] \mid P'_i$ *implies* $C \in \mathsf{dom}(\mathtt{CT}_i)$*;*
2. $\mathsf{fv}(P_i) \subseteq \mathsf{fv}(\sigma_i) \subseteq \{\tilde{\mathsf{u}}\}$*;*
3. $0 \le j \ne i < n$ *implies* $\mathsf{dom}(\sigma_i) \cap \mathsf{dom}(\sigma_j) = \emptyset$*;*
4. $\mathsf{o} \in \mathsf{fn}(P_i) \cup \mathsf{fn}(\sigma_i)$ *implies* $\mathsf{o} \in \mathsf{dom}(\sigma_j)$ *for some j ($0 \le j < n$);*
5. $\sigma_i(\mathsf{o}) = (C, \ldots)$ *implies* $C \in \mathsf{dom}(\mathtt{CT}_i)$*;*
6. $P_i \equiv \mathsf{c}.\textbf{send}(\mathsf{o}.\mathsf{m}(\mathsf{v})) \mid P'_i$ *and* $\mathsf{c} : \textbf{chan}(\mathsf{t}) \in \tilde{\mathsf{u}} : \tilde{\mathsf{t}}$ *imply* $\mathsf{o} \in \mathsf{dom}(\sigma_i)$.

Proof. (1). $\vdash N : \textbf{net}$ implies by Theorem D.12 and Lemma D.11(3) and (2) $\tilde{\mathsf{u}} : \tilde{\mathsf{t}} \vdash l_i[P_i, \sigma_i, \mathtt{CT}_i] : \textbf{net}$ for all i ($0 \le i < n$). By Lemma D.11(1) we get $\mathsf{fcl}(P_i) \subseteq \mathsf{dom}(\mathtt{CT}_i)$.

(2), (3), (4), and (5). By induction on $\longrightarrow$. It is easy to check that these conditions hold for initial nets. Notice that the domains of stores only contain variables and object identifiers. The induction step follows from the observation that:

– the variables are only created in rule **RC-LocMeth**,
– the object identifiers are only created in rule **RC-New**,

and both these rules satisfy the conditions.

(6). The run time expression $\mathsf{c}.\textbf{send}(\mathsf{o}.\mathsf{m}(\mathsf{v}))$ with $\mathsf{c} : \textbf{chan}(\mathsf{t})$ is only created in rule **RN-RemMeth**, where the condition holds.

[5] $C[\,]$ is an arbitrary context.

Lemma E.8. *1. If* $\Gamma;\Sigma,\mathsf{c}:!\mathsf{t}.\mathsf{s}\vdash\mathsf{e}:\mathsf{t};\Sigma',\mathsf{c}:\mathsf{s}$ *then* e *contains exactly one occurrence of* c *inside a sub-expression of the shape* $\mathsf{c}.\mathbf{send}(\mathsf{e}')$ *for some* e' *which is not in a session part of* e.

2. *If* $\Gamma;\Sigma,\mathsf{c}:?\mathsf{t}.\mathsf{s}\vdash\mathsf{e}:\mathsf{t};\Sigma',\mathsf{c}:\mathsf{s}$ *then* e *contains exactly one occurrence of* c *inside a sub-expression of the shape* $\mathsf{c}.\mathbf{receive}$ *which is not in a session part of* e.
3. *If* $\mathcal{D}$ *is a derivation of* $\Gamma;\Sigma\vdash\mathsf{e}:\mathsf{t};\Sigma'$*, and there are sub-derivations of* $\mathcal{D}$ *whose conclusion are respectively* $\Gamma_1;\Sigma_1\vdash\mathsf{e}_1:\mathsf{t}_1;\Sigma_1'$ *and* $\Gamma_2;\Sigma_2\vdash\mathsf{e}_2:\mathsf{t}_2;\Sigma_2'$ *with* $\Sigma_1'\preceq^{\supseteq}\Sigma_2$*, then* $\mathsf{e}_1\preceq_{\mathsf{e}}\mathsf{e}_2$.

Proof. (1) and (2). Since only rules **TE-SessSend** and **TE-SessReceive** shorten the session types occurring in session environments.

(3). By induction on the deductions.

Theorem E.9 (Evaluation Order).
If N_0 *is an initial net,* $N_0\longrightarrow\!\!\!\!\rightarrow N\Downarrow$ *where* $N\equiv(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})(l[\mathsf{e}\,|\,P,\sigma,\mathtt{CT}]\,\|\,N')$ *and* $\mathsf{e}_1\preceq_{\mathsf{e}}\mathsf{e}_2$*, then* $N\longrightarrow\!\!\!\!\rightarrow(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})(l[E_1[\mathsf{e}_1]\,|\,P,\sigma,\mathtt{CT}]\,\|\,N_1)\longrightarrow\!\!\!\!\rightarrow(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})(l[E_2[\mathsf{e}_2]\,|\,P,\sigma,\mathtt{CT}]\,\|\,N_2)$ *for some* $E_1[\,],N_1,E_2[\,],N_2$.

Proof. By induction on the definition of $\mathsf{e}_1\preceq_{\mathsf{e}}\mathsf{e}_2$.

Theorem E.10 (Session Evaluation).

1. *If* $\Gamma;\Sigma\vdash\mathbf{request}\;\mathsf{u}\;\mathsf{s}\{\mathsf{e}\}:\mathsf{t};\Sigma'$ *with* $\mathsf{s}=\dagger_1\mathsf{t}_1.\cdots.\dagger_n\mathsf{t}_n.\mathbf{end}$ *then there are* $\mathsf{e}_1,\dots,\mathsf{e}_n$ *which are not* u*-session parts of* e *such that* $\mathsf{e}_i\preceq_{\mathsf{e}}\mathsf{e}_{i+1}$ $(1\le i<n)$ *and*
$$\mathsf{e}_i=\begin{cases}\mathsf{u}.\mathbf{receive} & \text{if } \dagger_i=?,\\ \mathsf{u}.\mathbf{send}(\mathsf{e}) & \text{if } \dagger_i=!,\end{cases}\quad(1\le i\le n).$$
2. *If* $\Gamma;\Sigma\vdash\mathbf{accept}\;\mathsf{u}\;\mathsf{s}\{\mathsf{e}\}:\mathsf{t};\Sigma'$ *with* $\mathsf{s}=\dagger_1\mathsf{t}_1.\cdots.\dagger_n\mathsf{t}_n.\mathbf{end}$ *then there are* $\mathsf{e}_1,\dots,\mathsf{e}_n$ *which are not* u*-session parts of* e *such that* $\mathsf{e}_i\preceq_{\mathsf{e}}\mathsf{e}_{i+1}$ $(1\le i<n)$ *and*
$$\mathsf{e}_i=\begin{cases}\mathsf{u}.\mathbf{receive} & \text{if } \dagger_i=!,\\ \mathsf{u}.\mathbf{send}(\mathsf{e}) & \text{if } \dagger_i=?,\end{cases}\quad(1\le i\le n).$$

Proof. 1. By Lemma D.8(3) $\Gamma;\Sigma\vdash\mathbf{request}\;\mathsf{u}\;\mathsf{s}\{\mathsf{e}\}:\mathsf{t};\Sigma'$ implies $\mathsf{u}:\mathsf{s}\in\Gamma$ and $\Gamma;\Sigma,\mathsf{c}:\mathsf{s}\vdash\mathsf{e}[\mathsf{c}/\mathsf{u}]:\mathsf{t};\Sigma',\mathsf{c}:\mathbf{end}$ for some session type s and channel name c such that $\mathsf{c}\notin\mathsf{fn}(\mathsf{e})\cup\mathsf{dom}(\Gamma)$. Lemma E.8(1) and (2) assure the existence of $\mathsf{e}_1,\dots,\mathsf{e}_n$ with the required shapes and (3) gives $\mathsf{e}_i\preceq_{\mathsf{e}}\mathsf{e}_{i+1}$ $(1\le i<n)$.

Theorem E.11 (Soundness). *If* N_0 *is an initial net,* $N_0\longrightarrow\!\!\!\!\rightarrow(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})N$ *and* $(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})N\longrightarrow(\nu\mathsf{c}:\mathsf{s})(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})N'\stackrel{\text{def}}{=}(\nu\mathsf{c}:\mathsf{s})N_1$ *by rule* (**RN-ReqAcc**) *with* $\mathsf{s}=\dagger_1\mathsf{t}_1.\cdots.\dagger_n\mathsf{t}_n.\mathbf{end}$. *Then either* $(\nu\mathsf{c}:\mathsf{s})N_1\Uparrow$ *or*

$$(\nu\mathsf{c}:\dagger_1\mathsf{t}_1\cdots.\dagger_n\mathsf{t}_n.\mathbf{end})N_1\longrightarrow\!\!\!\!\rightarrow(\nu\mathsf{c}:\dagger_2\mathsf{t}_2\cdots.\dagger_n\mathsf{t}_n.\mathbf{end})N_2\longrightarrow\!\!\!\!\rightarrow\cdots\longrightarrow\!\!\!\!\rightarrow(\nu\mathsf{c}:\mathbf{end})N_{n+1}$$

with $c\notin\mathsf{fn}(N_{n+1})$.

Proof. If $(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})N\longrightarrow(\nu\mathsf{c}:\mathsf{s})(\nu\tilde{\mathsf{u}}:\vec{\mathsf{t}})N'$ by rule (**RN-ReqAcc**) then

$$N\equiv l_l[E_1[\mathbf{request}\;\mathsf{c}'\,\mathsf{s}\{\mathsf{e}_1\}]\,|\,Q_1,\sigma_1,\mathtt{CT}_1]\,\|\,l_2[E_2[\mathbf{accept}\;\mathsf{c}'\,\mathsf{s}\{\mathsf{e}_2\}]\,|\,Q_2,\sigma_2,\mathtt{CT}_2]\,\|\,N''$$

$$N' \equiv l_I[E_1[\mathtt{e}_1[\mathtt{c}/\mathtt{c}'] \mid Q_1], \sigma_1, \mathtt{CT}_1] \parallel l_2[E_2[\mathtt{e}_2[\mathtt{c}/\mathtt{c}']] \mid Q_2, \sigma_2, \mathtt{CT}_2] \parallel N''$$

Being N_0 an initial net, by Theorem D.12 $\tilde{\mathtt{u}} : \tilde{\mathtt{t}} \vdash N : \textbf{net}$ and therefore by Lemmas D.11, D.10 and D.4 there are Γ_1, Σ_1, $\tilde{\mathtt{c}}_1$, $\tilde{\mathtt{u}}_1$, $\mathtt{t}_1{}'$, Σ_1' and Γ_2, Σ_2, $\tilde{\mathtt{c}}_2$, $\tilde{\mathtt{u}}_2$, $\mathtt{t}_2{}'$, Σ_2' such that:

$$\Gamma_1; \Sigma_1 \vdash \textbf{request}\ \mathtt{c}'\,\mathtt{s}\,\{\mathtt{e}_1\}[\tilde{\mathtt{c}}_1/\tilde{\mathtt{u}}_1] : \mathtt{t}_1{}'; \Sigma_1'$$

$$\Gamma_2; \Sigma_2 \vdash \textbf{accept}\ \mathtt{c}'\,\mathtt{s}\,\{\mathtt{e}_2\}[\tilde{\mathtt{c}}_2/\tilde{\mathtt{u}}_2] : \mathtt{t}_2{}'; \Sigma_2'.$$

By Lemma E.10 this implies that then there are $\mathtt{e}_1', \ldots, \mathtt{e}_n'$ which are not $\mathtt{c}'$-session parts of $\mathtt{e}_1[\tilde{\mathtt{c}}_1/\tilde{\mathtt{c}'}_1]$ such that $\mathtt{e}_i' \preceq_{\mathtt{e}_1[\tilde{\mathtt{c}}_1/\tilde{\mathtt{c}'}_1]} \mathtt{e}_{i+1}'$ $(1 \leq i < n)$ and

$$\mathtt{e}_i' = \begin{cases} \mathtt{c}'.\textbf{receive} & \text{if } \dagger_i = ?, \\ \mathtt{c}'.\textbf{send}(\mathtt{e}) & \text{if } \dagger_i = !, \end{cases} \quad (1 \leq i \leq n)$$

and that there are $\mathtt{e}_1'', \ldots, \mathtt{e}_n''$ which are not $\mathtt{c}'$-session parts of $\mathtt{e}_2[\tilde{\mathtt{c}}_2/\tilde{\mathtt{c}'}_2]$ such that $\mathtt{e}_i'' \preceq_{\mathtt{e}_2[\tilde{\mathtt{c}}_2/\tilde{\mathtt{c}'}_2]} \mathtt{e}_{i+1}''$ $(1 \leq i < n)$ and

$$\mathtt{e}_i'' = \begin{cases} \mathtt{c}'.\textbf{receive} & \text{if } \dagger_i = !, \\ \mathtt{c}'.\textbf{send}(\mathtt{e}) & \text{if } \dagger_i = ?, \end{cases} \quad (1 \leq i \leq n).$$

By Lemma E.5 $\mathtt{e}_1', \ldots, \mathtt{e}_n'$ and $\mathtt{e}_1'', \ldots, \mathtt{e}_n''$ are evaluated in the given order and this concludes the proof.

Theorem E.12 (Typeable Stucks). *Suppose N_0 is an initial net and we have $N_0 \longrightarrow\!\!\!\!\rightarrow N$ with N a stuck net not containing* `Error`*. Then there exists l_i, $\mathtt{e}_i$, σ_i and $\mathtt{CT}_i$ $(0 \leq i < n)$ such that $N \equiv (\nu \tilde{\mathtt{u}} : \vec{\mathtt{t}}) \prod_{0 \leq i < n} l_i[\mathtt{e}_i, \sigma_i, \mathtt{CT}_i]$ and e_i is either* (1) $E[\textbf{request}\ \mathtt{c}\,\mathtt{s}\,\{\mathtt{e}'\}]$ *or* (2) $E[\textbf{accept}\ \mathtt{c}\,\mathtt{s}\,\{\mathtt{e}'\}]$.

Proof. By cases on the shape of $\mathtt{e}$. By Lemma E.7(1) $\mathtt{e}$ cannot be **new** C. By Lemma E.7(2) $\mathtt{e}$ cannot be $\mathtt{x}$. By Lemma E.7(4) $\mathtt{e}$ cannot be $\mathtt{o.m(v)}$. By rule **RN-RemMeth** $\mathtt{e}$ cannot be $\mathtt{c}.\textbf{send}(\mathtt{w})$ or $\mathtt{c}.\textbf{receive}$ if $\mathtt{c}$ is a proper channel. By Lemma E.10 $\mathtt{e}$ cannot be $\mathtt{c}.\textbf{send}(\mathtt{w})$ or $\mathtt{c}.\textbf{receive}$ if $\mathtt{c}$ is a session channel. Since N is well-typed $\mathtt{e}$ cannot be a "message-not-understood" error.

Now Theorem 6.5 is a direct consequence of the above theorem.