Amalgamating the Session Types and the Object Oriented Programming Paradigms

Sophia Drossopoulou[1], Mariangiola Dezani-Ciancaglini[2], Mario Coppo[2]

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

Abstract. We suggest an amalgamation of the session type and the object oriented paradigm whereby sessions are amalgamated with methods; where threads consist of the execution of session bodies on objects and communicate with each other through asynchronously sending/receiving objects on channels; where the choice on how to respond to a session request is based on the name of the request and the class of the object receiving the request; where the choice on how to continue a session is made on the basis of the class of the object sent/received; and where sessions are not first class, but can be delegated to other sessions.
We demonstrate our ideas through a small language, STOOP, and an example. We formalize a smaller calculus, $\mathcal{F}eather^{\texttt{STOOP}}$, and give a formal definition, and prove subject reduction and progress. The latter property is notoriously difficult and sometimes impossible to achieve in sessions languages, however it holds in $\mathcal{F}eather^{\texttt{STOOP}}$.

1 Introduction

Session types [14, 18] or channel contracts [9] support lightweight descriptions of communication protocols, by giving types to communication channels, in terms of the types of values sent or received, *e.g.*, the type `?int.!bool` expresses that an integer will be received and then a boolean value will be sent. A session involves channels of dual session type, thus guaranteeing, that after a session has started, the values sent and received will be of the appropriate type.

Session types have been introduced into several different settings, *i.e.*, for variants of the π-calculus [1, 11, 13, 14, 18], for CORBA [19], for functional languages [12, 20], for boxed ambients [10], for the W3C standard description language for Web Services CDL [2, 3, 15, 17, 21], and for object oriented programming languages [4, 6–8, 19]. In the Singularity OS [9] C# has been extended with channel communication across threads which is governed by channel contracts, and verified against state machine descriptions.

Applications of session types span from web services [17] to operating systems [9]. Such applications are often written on object oriented languages, thus motiving research into the *combination* of session types with object oriented languages.

In the current paper we explore the *amalgamation* of the paradigm of session types with that of object oriented programming. This contrasts to earlier work [4, 6–8, 19] which *extended* the oo paradigm, or even existing languages, with features from session

	"traditional" session	"traditional" method	"amalgamated" session/method
request on	a thread	an object	an object
execution starts when	threads reach certain point	immediately	immediately
executed body is	rest of thread	determined by class of receiver	determined by class of receiver
execution	concurrent	same thread	concurrent
communication	any direction interleaved with computations	n-inputs then computation then one output	any direction interleaved with computations

Fig. 1. "traditional" sessions, "traditional" methods, and "amalgamated" sessions/methods .

types. The need for such an amalgamation stems from the observation, that sessions and methods have related, but different features.

We call our approach STOOP (for *S*ession *T*ypes and *O*bject *O*riented *P*rogramming). STOOP is only a "language kernel", since it is only concerned with the amalgamation of the oo and the sessions paradigm, but is agnostic about issues to do with synchronization, distribution, copying of values across local heaps *e.t.c.*. STOOP drives the amalgamation very deeply, by unifying sessions and methods, and by basing choices (*i.e.*, which session body to choose, and how to continue with the session) on the class objects (*i.e.*, the receiving object, or the object being sent or received), rather than, on labels, as in traditional session types. STOOP also supports delegation of the current active channel, but does not support higher order sessions, which can be found in most session types systems, *e.g.*, [1, 3, 4, 6, 7, 11, 13, 14, 16, 18]. Furthermore, STOOP supports recursive types, as in traditional session type systems. Because STOOP supports delegation, but not higher order sessions, it satisfies the progress property, a property which is *not* guaranteed in most sessions types systems, even though it is guaranteed in [6–8].

We believe that the amalgamation we suggest leads to a clearer language design, and that STOOP can elegantly express most useful protocols, in a simple way, while retaining the highly desirable progress property.

The current paper is organized as follows: In section 2 we discuss the philosophy of STOOP , in section 3 we describe STOOP in terms of an example, and in section 4 we discuss the approach, how it would be incorporated into a "real" language. We study the language STOOP , in terms of a featherweight, smaller language, $\mathcal{F}eather^{\texttt{STOOP}}$. In section 5 we give an overview of $\mathcal{F}eather^{\texttt{STOOP}}$, and discuss how it can encode the features used in the example, and how traditional methods can be encoded by STOOP sessions. In sections 6, 7, 8 we describe the syntax, operational semantics and static typing of $\mathcal{F}eather^{\texttt{STOOP}}$, and in section 9 we outline the proofs of subject reduction and of progress. In section 10 we draw conclusions and discuss further work.

2 The philosophy of STOOP

In figure 1 we compare "traditional" sessions [7, 14, 20] and "traditional" methods from object oriented languages, with the "amalgamated" session/methods as in STOOP. "Traditional" sessions are invoked on threads in a manner similar to the Ada rendez-vous, and execution starts when two threads reach a certain point in their execution, where they can "service" the session. The body is determined by the rest of the expression currently run by the thread, and is executed concurrently with the requesting thread. "Traditional" sessions allow communication of any number of values in any direction. On the other hand, "traditional" methods are invoked on an object, the body to run is determined by the class of the receiving object, execution is immediate, and takes place in the same thread as the request, and it supports any number of inputs, followed by computation, followed by one output.

In STOOP we have "amalgamated" sessions/methods, which, for brevity, we shall call sessions from now on. Invocation takes place on an object (*e.g.*, a customer asks to withdraw money from a particular ATM machine), and execution takes place immediately, and concurrently with the requesting thread. The body is determined by the class of the receiving object (*e.g.*, the ATM services the withdrawal request according to its class), and any number of communications interleaved with computation are possible.

We believe that the above amalgamated model of session naturally reflects our intuition of services. Furthermore, it can neatly encode "standard" methods.

We now list the features of STOOP:

- Classes have fields and session bodies. Session bodies are selected based on the object classes.
- STOOP is multithreaded, and communication is asynchronous.
- At every step, in each thread, there is one single active channel on which communications are performed.
- A thread may make a session request through $\mathtt{e.s}\{\mathtt{e}'\}$; then, e is evaluated to an object, and the expression from the session body in its class is executed concurrently with $\mathtt{e}'$, introducing a new pair of fresh channels k and $\tilde{\mathtt{k}}$ (one for each communication direction) to perform communications between the session body and $\mathtt{e}'$ [3].
- The expressions send(e) and receive send/receive objects on the active thread channel.
- The expression $\mathtt{sendCase(e)}\,\{C_1 \rhd \mathtt{e}_1 \,[]\, \ldots \,[]\, C_n \rhd \mathtt{e}_n\}$ evaluates object e and sends it on the active thread channel, and then continues with $\mathtt{e_i}$, where C_i is the class that best fits the class of the object sent. The meaning of $\mathtt{receiveCase(x)}\{C_1 \rhd \mathtt{e}_1 \,[]\, \ldots \,[]\, C_n \rhd \mathtt{e}_n\}$ is the obvious one[4].
- $\mathtt{sendWhile(e)}\,\{C_1 \rhd \mathtt{e}_1 \,[]\, \ldots \,[]\, C_n \rhd \mathtt{e}_n\}$ is similar to $\mathtt{sendCase(e)}\,\{C_1 \rhd \mathtt{e}_1 \,[]\, \ldots \,[]\, C_n \rhd \mathtt{e}_n\}$, except that it allows for enclosed continue. Thus, $\mathtt{receiveWhile(x)}\{C_1 \rhd \mathtt{e}_1 \,[]\, \ldots \,[]\, C_n \rhd \mathtt{e}_n\}$ has the obvious meaning.

[3] k and $\tilde{\mathtt{k}}$ plays for channels a role similar to that of this for objects.

[4] Now, choice is based on the class of the object received.

```
1  sessiontype Shopping_ST =
2    !Item.?Money.μα.!{ OK ▷ !AccntNr.!Money.?{ OK ▷ ?Date,
3                                               NoMoney ▷ ε} ,
4                       NoDeal ▷ ε ,
5                       MakeAnOffer ▷ ? { Money ▷ α,
6                                         NoDeal ▷ ε } }
7
8  sessiontype Sell_ST =
9    ?Item.!Money.μα.?{ OK ▷ !{ OK ▷ ?AccntNr.?Money.!{ OK▷!Date,
10                                                      NoMoney ▷ ε} },
11                      NoDeal ▷ ε,
12                      MakeAnOffer ▷ ! { Money ▷ α,
13                                        NoDeal ▷ ε } }
14
15 sessiontype CreditCheck_ST = ?AccntNr.?Money.!{OK ▷ !Date,
16                                                 NoMoney ▷ ε}
```

Fig. 2. Session types for shopping, for selling, and for checking balance

- The expression `continue` continues execution at the nearest enclosing `sendWhile` or `receiveWhile`.
- The expression `e•s{}` evaluates the object `e`, and delegates to it the current session. The body of the session `s` in the class of that object is executed concurrently, using the current session. At the end, the final value of the body is passed to the current thread.

3 An Example

We adapt a popular example involving a buyer, a seller and a bank. The buyer negotiates a price from a seller, and if and when they have reached agreement, he sends his bank account number so that it gets verified that he has enough money. If he has enough money, he receives the delivery date, otherwise the deal falls through. The seller *delegates* to a bank the part of the session that checks the money in the account. Such delegation has traditionally been expressed through higher order sessions; instead, `STOOP` can delegate the current session.

The negotiation allows several rounds: the buyer may either accept the price, or break the negotiation, or require a better deal; in the latter case, the seller might respond with the better price, or might break the negotiation. In other words, the negotiation may be broken (and thus the iteration might be exited) at two different places in the protocol, which belong to different parties. Therefore, regular expressions as in [7] are insufficient to express the protocol; instead we need recursive types.

The session type `Shopping_ST`, from figure 2, describes the above protocol from the point of view of the buyer. The part `!Item.?Money` indicates sending an `Item` followed by receipt of a `Money`. The recursive type $\mu\alpha.!\{$ `OK` $\triangleright$ ..., `NoDeal` $\triangleright$ ...,

```
1  \begin{lstlisting}
2  class Buyer {
3    Item prodId; AccntNr accnt; Date delivDate; Seller seller;
4
5   Object Shopping_ST shopping
6       {  prodId = ....;
7          seller.sell{
8             send( prodID );
9             Money price = receive;
10            Response resp = ... price ...;
11            sendWhile( resp ){
12               OK ▷ { send(accnt); send(price);
13                      receiveCase( x ) { OK ▷ delivDate=receive; []
14                                         NoMoney ▷ null; } }[]
15               NoDeal ▷ null; []
16               MakeAnOffer ▷ {
17                   receiveCase( x ) { Money ▷ { resp = ... x ...;
18                                                continue; } []
19                                      NoDeal ▷ null; } }
20                         }
21               }
22            }
23      }
```

Fig. 3. The class `Buyer`

`MakeOnOffer` ▷ ... } describes the negotiation part, whereby an object is sent, and then, depending on whether the actual object sent belongs to class `OK`, `NoDeal`, or `MakeAnOffer`, the first, second or third branch is taken. In the first branch, the account number and the price is sent; then, either `OK` followed by a `Date`, or a `NoMoney` is received. In the third branch, a further object is received, and if that object is a `Money`, then the negotiations resume on the basis of it, whereas if it is a `NoDeal`, the negotiation ends.

In figure 3 we show the implementation of the class `Buyer`. It has the fields `prodId`, `accnt`, `delivDate`, and `seller`, which (will) contain the identity of the product to be bought, the account number, the delivery date, and the seller.

The class supports one session, called `shopping`, with session type `Shopping_ST` and return type `Object`. In the body of that session the desirable product is determined and stored in `prodId` (line 6). Then, a session request is made to the `seller` to run session `sell` (line 7). Thus, the seller will run the body of `sell` in parallel with the remaining part of the session body of `shopping`, and a connection will be created between the two threads. On this connection, the `Buyer` will send an `Item` (line 8), receive a `Money` and store it in `price` (line 9), and based on its value will calculate his response based on the `price` and store it in `resp` (line 10). On line 11, the buyer enters a loop with `sendWhile`, where he sends `resp`, and branches according to its class. If `resp` is `OK`, indicating acceptance of the price, then the buyer will send his account

```
1  class Seller {
2    Bank bank; Item prodId; BetterPriceOrDeal resp;
3
4    Object Sell_ST sell
5       { String prodID = receive;
6         Money price = ... prodId ...;
7         send(price);
8         receiveWhile( x ){
9              OK ▷ { case( bank•check{ } )
10                          { OK ▷ send(... calc.del.date...); []
11                            NoMoney ▷ null; } } []
12             NoDeal ▷ null; []
13             MakeAnOffer ▷ { resp = ....price .... ;
14                              sendCase { Money ▷ continue ; []
15                                         NoDeal ▷ null; } }
16           }
17      }
18   }
```

Fig. 4. The class `Seller`

number, and price (line 12); and will receive an object which may be `OK`, in which case he will receive a `Date` and store it in `delivDate` (line 13), or will receive a `NoMoney` (line 14). If `resp` is `NoDeal`, indicating that the price is unacceptable, then the session terminates. If the response is `MakeAnOffer`, inviting the seller to make a better offer, then the rest depends on the other party's response. Line 17 contains `receiveCase` indicating that a value will be received, and the remaining steps will be determined by its class. If the value received is a `Money`, then it will be stored in `price`, and a response will be calculated, and stored in `resp` (line 17) and the recursion will continue (line 18). If the value received is `Nodeal`, then the loop will be abandoned.

The session type `Sell_ST` from figure 2 describes the above protocol from the point of view of the seller, and is "dual" to `Shopping_ST`. We now consider the class `Seller`, from figure 4. The session body for `sell` starts by receiving the description of an `Item`, calculating and sending its price. Then, in line 8, it enters a `receiveWhile` loop, which is the counterpart to the `sendWhile` loop from `shopping` and performs all the seller's negotiation. The *interesting new feature* shown here is *delegation*, on line 9, whereby, the `bank` is requested to continue the session, using the current connection, and by application of the session body `check`.

The session type for `check` from class `Bank` in figure 5 is the receipt of a `AccntNr` and a `Money` followed by sending either `OK`, or a `NoMoney` object. Note, that a session body for `check` is *not aware* whether it will be called through a session request, or through delegation. The return type of `check` is the class `OKorNoMoney` which is the common superclass of `OK` and `NoMoney`.

```
class Bank {
    OKorNoMoney CreditCheck_ST check
       {  AccntNr accnt = receive;
          Money amt = receive;
          response = ... acct ... amt ...
          sendCase(response){ OK ▷ null; [] NoMoney ▷ null; }
       }
   }
```

Fig. 5. The class Bank

4 Discussion of STOOP

As we said in the introduction, STOOP is a "language kernel", in the sense that it is only concerned with the best amalgamation of the object oriented features with the sessions part, but is agnostic wrt to the remaining features of the language, such as whether the language is distributed or concurrent, and the features for synchronization.

A core STOOP design choice is the amalgamation of session and method. In particular, each session request causes a new thread/process to be spawned. This makes the language semantics very elegant, but, of course, might end up in creating an undesired number of threads/processes. It would be a straightforward extension of the semantics to ensure the execution of "method-like" (*i.e.*, those which first receive values, then compute but do not communicate, and then send a value) sessions in the same thread-/process. Slightly more sophisticated schemes would be needed in the distributed setting to ensure that co-located objects execute in the same process.

In STOOP, each object has the capacity to execute at any point in time an unlimited number of threads/processes. Thus an ATM would have the capability to serve any number of customers, even if its cash dispenser were empty. This is clearly an oversimplification, and it would be straightforward to add "readiness"-fields to STOOP so that an object can only execute a certain number of each of its sessions at each point in time, depending also on its internal state. Such schemes can be adapted from those in [5, 9].

So far we have not concerned ourselves with whether we are dealing with a concurrent or a distributed setting. From the point of view of types, and assuming that in the distributed setting all threads would share a class table, the issue is irrelevant. This is why our core language, $\mathcal{F}eather^{\texttt{STOOP}}$, is concurrent, thus making the semantics much simpler.

However, when we go to a full language, for a concurrent setting we will need features to ensure synchronization, which we have left to further work. On the other hand, in a distributed setting we might need to consider issues to do with data marshalling, ownership, and heap partitioning, which we have also left to further work.

5 $\mathcal{F}eather^{\texttt{STOOP}}$ overview, and encoding of STOOP in $\mathcal{F}eather^{\texttt{STOOP}}$

In order to explain and evaluate the STOOP approach, we designed a small, concurrent, imperative language $\mathcal{F}eather^{\texttt{STOOP}}$, whose expressions only support the basic oo features

and also session request, session delegation, and branching sending/receiving loops. In more detail, *Feather*$^{\texttt{STOOP}}$ encompasses the following features:

- classes, inheritance, fields, session bodies,
- field access, sequence, object creation,
- the constructs $\texttt{receiveW}(\texttt{x})\{\overline{C \triangleright \texttt{e}}\}$ and $\texttt{sendW}(\texttt{e})\{\overline{C \triangleright \texttt{e}}\}$ which combine receive/send with branching and loops,
- `continue`.

Thus, we omit from *Feather*$^{\texttt{STOOP}}$ several of the communication constructs used in the example in section 3, *i.e.*, `send` for sending, `sendCase` for sending and branching depending on the class of the object being sent. In *Feather*$^{\texttt{STOOP}}$ we only include the `sendWhile` construction, because the other two can be encoded in terms of the latter.

In actual fact, the expression $\texttt{sendWhile}(\texttt{e})\{\texttt{Obj} \triangleright \texttt{null}^{\texttt{Obj}}\}$ encodes $\texttt{send}(\texttt{e})$. Furthermore, $\texttt{sendWhile}(\texttt{e})\{\overline{C \triangleright \texttt{e}}\}$ encodes $\texttt{sendCase}(\texttt{e})\{\overline{C \triangleright \texttt{e}}\}$. In a similar way, `receive`, `receiveCase` can be encoded in terms of `receiveWhile` [5]. Furthermore, we abbreviate `sendWhile` and `receiveWhile` to `sendW` and `receiveW` and `continue` to `cont`.

Finally, we discuss how methods are special cases of *Feather*$^{\texttt{STOOP}}$ sessions. Thus, the method declaration

$C\ \texttt{m}\ (C_1\ \texttt{x}_1, \ldots, C_n\ \texttt{x}_n)\{\ \texttt{e}\ \}$

can be encoded as

$\texttt{Obj}\ \mu\alpha_1.?\{C_1 \triangleright \ldots \mu\alpha_n.?\{C_n \triangleright \mu\alpha.!\{C \triangleright \varepsilon\}\}\ldots\}\ \texttt{m}$
$\{\texttt{receiveW}(\texttt{x}_1)\{C_1 \triangleright \{\ldots \texttt{receiveW}(\texttt{x}_n)\{C_n \triangleright \texttt{sendW}(\texttt{e})\{C \triangleright \texttt{null}^{\texttt{Obj}}\}\}\ldots\}\}$

Note that the recursion operator μ is irrelevant in this session type, since there is no iteration here.

Similarly, method calls are special cases of session requests. In fact, in a session request of form $\texttt{o.s}\{\texttt{e}\}$, the object `o` receives the request and `s` is the session name. Thus, a method call

$\texttt{o.m}(\texttt{v}_1, \ldots, \texttt{v}_n)$

can be encoded as

$\texttt{o.m}\{\texttt{sendW}(\texttt{v}_1)\{C_1 \triangleright \ldots \texttt{sendW}(\texttt{v}_n)\{C_n \triangleright \texttt{receiveW}(\texttt{x})\{C \triangleright \texttt{x}\}\}\ldots\}\}$

where $C_1, \ldots, C_n, C$ are respectively the parameter and return classes of the method `m` in the class of the object `o`.

6 *Feather*$^{\texttt{STOOP}}$ Syntax

In figure 6 we describe the syntax of *Feather*$^{\texttt{STOOP}}$, a featherweight representation of our language. We use gray to indicate runtime expressions, i.e. expressions that are

[5] Indeed the encoding of `SendCase` with `SendWhile` in *Feather*$^{\texttt{STOOP}}$ works only if there are no occurrences of `continue` in the body of a `SendCase` internal to a loop (as it happens, for instance, in the definition of the session `shopping`, see figure 3, line 18). To recover the full `STOOP` expressivity we should add `SendCase` as a separate construct. We avoid doing this since we want to study the basic features of the language, and the treatment of `SendCase` does not introduce any novelty with respect to that of `SendWhile`.

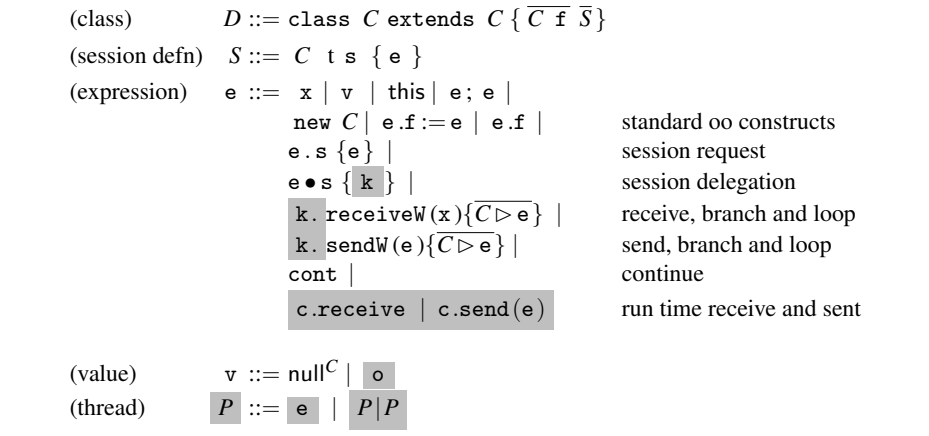

Fig. 6. *Feather*$^{\texttt{STOOP}}$ Syntax, where syntax occurring only at runtime appears shaded.

produced in the reduction process but are not expected to occur in the source code of a program. We also use the standard convention of denoting by $\overline{\xi}$ a sequence of elements $\xi_1, \ldots, \xi_n$. In the same way, we denote $\#(\overline{\xi})$ the length n of the sequence and with ξ_i the i-th element.

Programs are defined from a collection of classes ranged over by C, D. Each class is represented by a sequence of *fields* of the form $\texttt{f}\ C$, where $\texttt{f}$ represent the field name and C its class and by a list of session expressions of the form $C\ \texttt{t s}\ \{\texttt{e}\}$, where C is the return class, $\texttt{t}$ the session type, $\texttt{s}$ the session name, and $\texttt{e}$ the session body. All classes are defined as extensions of the topmost class $\texttt{Obj}$.

The first line in the definition of expressions describes standard syntax of object oriented languages, except for method call. The successive four lines describe the syntactic constructs which are characteristic of our approach, *i.e.*, the request of a session, the delegation of a session, and the constructs for receiving and sending objects (*communication expressions*).

The body of $\texttt{sendW}$ and $\texttt{receiveW}$ is a sequence of *alternatives*, $\{\overline{C \rhd \texttt{e}}\}$, whose choice depends on the class of the value sent or received. The alternatives $\texttt{e}_i$ ending with a $\texttt{cont}$ expression express repetition of the loop, the other ones represent exit points.

Channels are implicit in the source language syntax. At runtime, communication channels $\texttt{k}$ are introduced at each new session request. We indicate with the operation $\tilde{.}$ the dual, where $\tilde{\texttt{k}}$ is again a runtime channel, and where the operation is such that $\tilde{\tilde{\texttt{k}}} = \texttt{k}$. In each thread we use two channels: one for receiving, and its dual for sending. Note that the meaning of polarities is different from that in [11], where polarities simply represent the two ends of a (unique) session channel.

The run time receive and sent expressions $\texttt{c.receive}$ and $\texttt{c.send(e)}$ are handy to terminate the session delegation, see rule (SessDel-R). A fresh channel $\texttt{c}$ is created to allow this (unique) communication.

Objects are ranged over by $\texttt{o}$ and parallel threads by P.

7 $\mathcal{F}eather^{\texttt{STOOP}}$ Operational Semantics

Field lookup

$$fields(\texttt{Obj}) = \bullet \qquad \frac{fields(C') = \overline{\texttt{f}'D'} \qquad \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,\texttt{s}}\} \in \texttt{CT}}{fields(C) = \overline{\texttt{f}'D'}, \overline{\texttt{f}\,D}}$$

Field type lookup

$$\frac{\texttt{f}\,D \in fields(C)}{fType(C, \texttt{f}) = D}$$

Session lookup

$$sessions(\texttt{Obj}) = \bullet \qquad \frac{sessions(C') = \overline{S'} \qquad \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT}}{sessions(C) = \overline{S'}, \overline{S}}$$

Session type lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \quad C''\ \texttt{t}\ \texttt{s}\ \{\texttt{e}\} \in \overline{S}}{sType(\texttt{s}, C) = \texttt{t}}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \quad C''\ \texttt{t}\ \texttt{s}\ \{\texttt{e}\} \in \overline{S}}{rType(\texttt{s}, C) = C''}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \quad \texttt{s} \notin \overline{S}}{sType(\texttt{s}, C) = sType(\texttt{s}, C')} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \quad \texttt{s} \notin \overline{S}}{rType(\texttt{s}, C) = rType(\texttt{s}, C')}$$

Session body lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \qquad C''\ \texttt{t}\ \texttt{s}\ \{\texttt{e}\} \in \overline{S}}{sBody(\texttt{s}, C) = \texttt{e}}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\} \in \texttt{CT} \qquad \texttt{s} \notin \overline{S}}{sBody(\texttt{s}, C) = sBody(\texttt{s}, C')}$$

Fig. 7. Lookup Functions.

We assume a fixed, global class table CT, which as usual contains Obj as topmost class. In figure 7 we define the auxiliary functions for table lookup used in the operational semantics and typing rules.

Objects and values passed in asynchronous communications are stored in a *heap*. A heap h is a finite mapping with domain objects and channel names. Its syntax is given by:

$$h \ ::= \ [\,] \ \mid \ \texttt{o} \mapsto (C, \overline{\texttt{f} : \texttt{v}}) \ \mid \ \texttt{k} \mapsto \overline{\texttt{v}} \ \mid \ \texttt{c} \mapsto \texttt{v} \ \mid \ \texttt{c} \mapsto () \ \mid \ h{+}h$$

where $+$ denotes heap concatenation.

Standard OO and Thread Reduction

$$\frac{}{\mathtt{o.f} := \mathtt{v}, h \longrightarrow \mathtt{v}, h[\mathtt{o} \mapsto h(\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]]}\ (\textsc{FldAss-R}) \qquad \frac{}{\mathtt{v;e}, h \longrightarrow \mathtt{e}, h}\ (\textsc{Seq-R}) \qquad \frac{h(\mathtt{o}) = (C, \overline{\mathtt{f} : \mathtt{v}})}{\mathtt{o.f_i}, h \longrightarrow \mathtt{v_i}, h}\ (\textsc{Fld-R})$$

$$\frac{\mathit{fields}(C) = \overline{C\ \mathtt{f}} \qquad \mathtt{o} \notin h}{\mathtt{new}\ C, h \longrightarrow \mathtt{o}, h[\mathtt{o} \mapsto (C, \overline{\mathtt{f} : \mathsf{null}^C})]}\ (\textsc{NewC-R}) \qquad \frac{\mathtt{e}, h \longrightarrow \mathtt{e}', h'}{E[\mathtt{e}], h \longrightarrow E[\mathtt{e}'], h'}\ (\textsc{Cong-R}) \qquad \frac{\mathtt{e}, h \longrightarrow \mathtt{e}', h'}{P \mid \mathtt{e}, h \longrightarrow P \mid \mathtt{e}', h'}\ (\textsc{Par-R})$$

Communication Reduction

$$\frac{h(\mathtt{o}) = (C, \_) \quad \mathit{sBody}(\mathtt{s}, C) = \mathtt{e}' \quad \mathtt{k}, \tilde{\mathtt{k}} \notin h}{E[\mathtt{o.s}\ \{\mathtt{e}\}],\ h \quad \longrightarrow \quad E[\mathtt{e}\lfloor \mathtt{k} \rfloor] \mid \mathtt{e}'[\mathtt{o}/\mathsf{this}]\lfloor \tilde{\mathtt{k}} \rfloor,\ h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]}\ (\textsc{SessReq-R})$$

$$\frac{h(\mathtt{o}) = (C, \_) \quad \mathit{sBody}(\mathtt{s}, C) = \mathtt{e}' \quad \mathtt{c} \notin h}{E[\mathtt{o} \bullet \mathtt{s}\ \{\mathtt{k}\}],\ h \quad \longrightarrow \quad E[\mathtt{c.receive}] \mid \mathtt{c.send}(\mathtt{e}'[\mathtt{o}/\mathsf{this}]\lfloor \mathtt{k} \rfloor),\ h[\mathtt{c} \mapsto ()]}\ (\textsc{SessDel-R})$$

$$\frac{\mathtt{e}, h \longrightarrow^* \mathtt{v}, h' \qquad h'(\tilde{\mathtt{k}}) = \overline{\mathtt{v}} \qquad h'(\mathtt{v}) = (C, \_) \quad C \Downarrow \overline{C} = C_i}{\mathtt{k.sendW(e)}\{\overline{C \rhd \mathtt{e}}\},\ h \quad \longrightarrow \quad \mathtt{e}_i\lfloor \mathtt{k.sendW(e)}\{\overline{C \rhd \mathtt{e}}\}/\mathtt{cont} \rfloor,\ h'[\tilde{\mathtt{k}} \mapsto \overline{\mathtt{v}} :: \mathtt{v}]}\ (\textsc{sendWhile-R})$$

$$\frac{h(\mathtt{k}) = \mathtt{v} :: \overline{\mathtt{v}} \qquad h(\mathtt{v}) = (C, \_) \qquad C \Downarrow \overline{C} = C_i}{\mathtt{k.receiveW(x)}\{\overline{C \rhd \mathtt{e}}\},\ h \quad \longrightarrow \quad \mathtt{e}_i[\mathtt{v}/\mathtt{x}]\lfloor \mathtt{k.receiveW(x)}\{\overline{C \rhd \mathtt{e}}\}/\mathtt{cont} \rfloor,\ h[\mathtt{k} \mapsto \overline{\mathtt{v}}]}\ (\textsc{receiveWhile-R})$$

$$\frac{}{\mathtt{c.send(v)},\ h \quad \longrightarrow \quad \mathsf{null}^{\mathtt{Obj}},\ h[\mathtt{c} \mapsto \mathtt{v}]}\ (\textsc{send-R}) \qquad \frac{h(\mathtt{c}) = \mathtt{v}}{\mathtt{c.receive},\ h \quad \longrightarrow \quad \mathtt{v},\ h \backslash \mathtt{c}}\ (\textsc{receive-R})$$

Fig. 8. Expression Reduction.

Objects names are mapped to pairs consisting of the class of the object and the list of its fields associated with their values. Channel names are mapped to queues of values. The notation $h[\mathtt{o} \mapsto (C, \overline{\mathtt{f} : \mathtt{v}})]$ (where C is the class name and $\overline{\mathtt{f} : \mathtt{v}}$ denotes the association of field names with their values (in the instance considered)) represents both the update of the object $\mathtt{o}$ in h, if $\mathtt{o}$ is already in the domain of h, and the addition of the object $\mathtt{o}$ to h otherwise. The heap produced by $h[\mathtt{k} \mapsto \overline{\mathtt{v}}]$ maps the live channel $\mathtt{k}$ to the queue $\overline{\mathtt{v}}$ with the same convention as before. The channels $\mathtt{c}$ created for handling session delegation are instead used to store at most a single value and have no dual.

$$\mathtt{e}\lfloor \mathtt{k} \rfloor = \begin{cases} \mathtt{e}_1\lfloor \mathtt{k} \rfloor \bullet \mathtt{s}\,\{\mathtt{k}\} & \text{if } \mathtt{e} = \mathtt{e}_1 \bullet \mathtt{s}\,\{\,\}, \\ \mathtt{k.receiveW(x)}\{\overline{C \rhd \mathtt{e}\lfloor \mathtt{k} \rfloor}\} & \text{if } \mathtt{e} = \mathtt{receiveW(x)}\{\overline{C \rhd \mathtt{e}}\}, \\ \mathtt{k.sendW(x)}\{\overline{C \rhd \mathtt{e}\lfloor \mathtt{k} \rfloor}\} & \text{if } \mathtt{e} = \mathtt{sendW(x)}\{\overline{C \rhd \mathtt{e}}\}, \\ \mathtt{e}_1\lfloor \mathtt{k} \rfloor ; \mathtt{e}_2\lfloor \mathtt{k} \rfloor & \text{if } \mathtt{e} = \mathtt{e}_1 ; \mathtt{e}_2, \\ \mathtt{e}_1\lfloor \mathtt{k} \rfloor .\mathtt{f} := \mathtt{e}_2\lfloor \mathtt{k} \rfloor & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2, \\ \mathtt{e}_1\lfloor \mathtt{k} \rfloor .\mathtt{f} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}, \\ \mathtt{e}_1\lfloor \mathtt{k} \rfloor .\mathtt{s}\,\{\mathtt{e}_2\} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{s}\,\{\mathtt{e}_2\}, \\ \mathtt{e} & \text{otherwise.} \end{cases}$$

Fig. 9. Channel Replacement.

$$\mathtt{e}\lfloor \mathtt{e}'/\mathtt{cont} \rfloor = \begin{cases} \mathtt{e}' & \text{if } \mathtt{e} = \mathtt{cont}, \\ \mathtt{e}_1 ; \mathtt{e}_2\lfloor \mathtt{e}'/\mathtt{cont} \rfloor & \text{if } \mathtt{e} = \mathtt{e}_1 ; \mathtt{e}_2, \\ \mathtt{e} & \text{otherwise.} \end{cases}$$

Fig. 10. Continuation Replacement.

Heap membership for object identifiers and channels is checked using standard set notation, by identifying h with its domain, we can also write $\mathtt{o} \in h$, $\mathtt{k} \in h$, and $\mathtt{c} \in h$. We convene that $h(\mathsf{null}^C) = (C, \_)$.

In order to discuss the operational semantics of $\mathcal{F}eather^{\mathtt{STOOP}}$ we start by listing the evaluation contexts (based obviously on runtime syntax).

$$E ::= [-] \mid E.\mathtt{f} \mid E;\mathtt{e} \mid E.\mathtt{f} := \mathtt{e} \mid \mathtt{o}.\mathtt{f} := E \mid E.\mathtt{s}\{\mathtt{e}\} \mid E \bullet \mathtt{s}\,\{\mathtt{k}\} \mid \mathtt{c.send}(E)$$

Figure 8 gives the reduction rules of $\mathcal{F}eather^{\mathtt{STOOP}}$. Since value are passed and stored in the heap the reduction relation is of the form $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$. The first six rules define execution of standard object oriented constructions, execution within a context, and execution for threads. In the sixth rule, *i.e.*, rule (Par-R) parallel composition is considered modulo structural equivalence. As usual, we define structural equivalence rules asserting that parallel composition is associative and commutative:

$$P \mid P_1 \equiv P_1 \mid P \quad P \mid (P_1 \mid P_2) \equiv (P \mid P_1) \mid P_2 \quad P \equiv P' \Rightarrow P \mid P_1 \equiv P' \mid P_1$$

The communication rules of figure 8 are more interesting, especially the first four. We make use of the special channel update operation $\lfloor ... \rfloor$, and of the continuation replacement operation $\lfloor .../\mathtt{cont} \rfloor$, defined in figures 9 and 10. Thus, $\mathtt{e}\lfloor \mathtt{k} \rfloor$ is the expression $\mathtt{e}$ in which all occurrences of receive, send, and delegation expression which *are not* within the body of a session request are extended so that they explicitly mention

the channel $\mathtt{k}$ they will use. Also, $\mathtt{e}\lfloor \mathtt{e}'/\mathtt{cont} \rfloor$ is the expression $\mathtt{e}$ in which all occurrences of $\mathtt{cont}$ that *are not* within the body of a send/receive loop are replaced by $\mathtt{e}'$. Thus, occurrences of $\mathtt{cont}$ preserve the correct nested structure of while expressions. The typing rules assure that $\mathtt{cont}$ as proper subexpression can only occur as the last expression in some sequential composition (operator ";") or as an alternative of a while expression (see Proposition 8.1(1)).

The send communication rule puts the value $\mathtt{v}$, *i.e.*, the result of evaluating the expression $\mathtt{e}$ in the current thread, in the queue associated to the dual channel $\tilde{\mathtt{k}}$ of the communication channel $\mathtt{k}$. The computation then proceeds with the expression $\mathtt{e}_i\lfloor \mathtt{k.sendW}(\mathtt{e})\{\overline{C \rhd \mathtt{e}}\}/\mathtt{cont} \rfloor$, *i.e.*, with the expression $\mathtt{e}_i$ where all outermost occurrences of $\mathtt{cont}$ are replaced by $\mathtt{k.sendW}(\mathtt{e})\{\overline{C \rhd \mathtt{e}}\}$, if C_i is the smallest class in $\overline{C}$ to which the value $\mathtt{v}$ belongs. This is given by the conditions $h'(\mathtt{v}) = (C, \_)$ and $C \Downarrow \overline{C} = C_i$, where we define:

$$C \Downarrow \overline{C} = \begin{cases} C_i & \text{if } \; C <: C_i \text{ and } \; \forall j.\, C <: C_j \implies C_i <: C_j, \\ \bot & \text{otherwise.} \end{cases}$$

Dually the receive communication rule takes a value $\mathtt{v}$ from the queue associated to channel $\mathtt{k}$ and returns the expression $\mathtt{e}_i[\mathtt{v}/\mathtt{x}]\lfloor \mathtt{k.receiveW}(\mathtt{x})\{\overline{C \rhd \mathtt{e}}\}/\mathtt{cont} \rfloor$, if C_i is the smallest class in $\overline{C}$ to which the value $\mathtt{v}$ belongs.

Notice that in well typed expressions $C \Downarrow \overline{C}$ is always defined: this can be easily proved by inspecting the typing rules.

Note also that $\mathtt{k.sendW}(E)\{\overline{C \rhd \mathtt{e}}\}$ is not a context. This allows us to write a simpler semantics for $\mathtt{sendW}$, since no communication is allowed in E (this is assured by the typing rules). Note also the mixture of small and large step semantics in rule (SendWhile-R).

Rule (SessReq-R) creates a pair of fresh dual channels $\mathtt{k}, \tilde{\mathtt{k}}$ and replaces the occurrence of the session request $\mathtt{o.s}\ \{\mathtt{e}\}$ in the context $E[\ ]$ by the expression $\mathtt{e}\lfloor \mathtt{k} \rfloor$, *i.e.*, the expression $\mathtt{e}$ where all the communication and delegation commands of the current session use the channel $\mathtt{k}$. Moreover it generates the new thread $\mathtt{e}'[\mathtt{o}/\mathtt{this}]\lfloor \tilde{\mathtt{k}} \rfloor$, where $\mathtt{e}'$ is the body of the session $\mathtt{s}$ in the class of the object $\mathtt{o}$. Thus, an object can service *any number* of session requests; this is the object-oriented counterpart to the (conn) rule from [16].

Rule (SessDel-R) creates a fresh channel $\mathtt{c}$ which allows to synchronize the end of the execution of the body of the session $\mathtt{s}$ opened by a delegation. The session delegation $\mathtt{o} \bullet \mathtt{s}\ \{\mathtt{k}\}$ is replaced by the expression $\mathtt{c.receive}$, which expects a value via channel $\mathtt{c}$ and returns that value (see rule (Receive-R)). Moreover the new thread $\mathtt{c.send}(\mathtt{e}'[\mathtt{o}/\mathtt{this}]\lfloor \mathtt{k} \rfloor)$ is generated, where $\mathtt{e}'$ is the body of the session $\mathtt{s}$ in the class of the object $\mathtt{o}$. This thread first evaluates the expression $\mathtt{e}'[\mathtt{o}/\mathtt{this}]\lfloor \mathtt{k} \rfloor$, then sends the resulting value via channel $\mathtt{c}$ to the originating thread and lastly returns $\mathsf{null}^{\mathtt{Obj}}$ (see rule (Send-R)). This allows the delegation of a part of the communication via the channel $\mathtt{k}$ to the object $\mathtt{o}$: this delegation is transparent for the thread using the dual channel $\tilde{\mathtt{k}}$. When the delegated job is over the original thread can resume the communication via the channel $\mathtt{k}$. Note that $\mathtt{c.send(E)}$ is an evaluation context, so a delegated session can perform communications and delegations.

8 *Feather*$^{\texttt{STOOP}}$ Types and Typing Rules

8.1 Types

Session types, $\mathtt{t}$, describe the communications that take place during a session. The syntax of session types is:

$$\mathtt{t} \quad ::= \quad \varepsilon \mid \mathtt{t}.\mathtt{t} \mid \mu\alpha.\dagger\{\overline{C \triangleright \mathtt{t}}\} \mid \alpha \mid \square$$

where we use $\dagger$ as a convenient abbreviation that ranges over $\{!,?\}$. By ε we denote the *empty* communication, and the *concatenation* $\mathtt{t}_1.\mathtt{t}_2$ expresses the communications in $\mathtt{t}_1$ followed by those in $\mathtt{t}_2$. The session type ε is the neutral element of concatenation, so that $\varepsilon.\mathtt{t} = \mathtt{t} = \mathtt{t}.\varepsilon$ for all $\mathtt{t}$.

The types $\mu\alpha.!\{\overline{C \triangleright \mathtt{t}}\}$ and $\mu\alpha.?\{\overline{C \triangleright \mathtt{t}}\}$ express respectively the sending and the receiving of a value: depending on the class of this value the communication will proceed with one of the $\overline{\mathtt{t}}$. More precisely if C is the class of the value and $C \Downarrow \overline{C} = C_i$, then the communication will proceed with $\mathtt{t}_i$. Notice that the variable α can occur in $\overline{\mathtt{t}}$ with the usual meaning of representing the whole session type.

The type $\square$ is used only to type the command `cont`: it plays the role of a place holder which will be replaced by a type variable when the while expression is completed (see rules (RecW-T) and (SendW-T)).

Notice that the current session types are considerably simpler than those in [7].

8.2 Typing of Channel Free Expressions

Environments and Well-formed Environments

$$\textbf{Emp}\ \frac{}{\emptyset \vdash \mathtt{ok}} \qquad \textbf{EVar}\ \frac{C \in \mathsf{dom}(\mathtt{CT}) \quad \mathtt{x} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathtt{x} : C \vdash \mathtt{ok}} \qquad \textbf{EOid}\ \frac{C \in \mathsf{dom}(\mathtt{CT}) \quad \mathtt{o} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathtt{o} : C \vdash \mathtt{ok}} \qquad \textbf{Ethis}\ \frac{C \in \mathsf{dom}(\mathtt{CT}) \quad \mathsf{this} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{this} : C \vdash \mathtt{ok}}$$

Subtyping

$$\frac{C \in \mathsf{dom}(\mathtt{CT})}{C <: C} \qquad \frac{C <: C' \quad C' <: C''}{C <: C''} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\mathtt{f}\,D\,S}\} \in \mathtt{CT}}{C <: C'}$$

Fig. 11. Environments and Subtyping.

In this subsection we define typing for user expressions, in which communication channels are implicit. For technical reasons it is useful to consider also expressions with occurrences of object identifiers, which are not directly expressible in user syntax. We

Standard OO Typing Rules

$$\frac{}{\Gamma;\varepsilon \vdash \mathtt{null}^C : C}\ (\textsc{Null-T}) \qquad \frac{}{\Gamma;\varepsilon \vdash z : \Gamma(z)}\ (\textsc{Axiom-T}) \qquad \frac{}{\Gamma;\varepsilon \vdash \mathtt{new}\ C : C}\ (\textsc{New-T})$$

$$\frac{\Gamma;\mathtt{t} \vdash \mathtt{e} : C \qquad C <: C'}{\Gamma;\mathtt{t} \vdash \mathtt{e} : C'}\ (\textsc{Subs-T}) \qquad \frac{\Gamma;\mathtt{t}_1 \vdash \mathtt{e}_1 : C_1 \qquad \Gamma;\mathtt{t}_2 \vdash \mathtt{e}_2 : C_2^*}{\Gamma;\mathtt{t}_1 \circ \mathtt{t}_2 \vdash \mathtt{e}_1;\mathtt{e}_2 : {C_2}^*}\ (\textsc{Seq-T})$$

$$\frac{\Gamma;\mathtt{t} \vdash \mathtt{e} : C}{\Gamma;\mathtt{t} \vdash \mathtt{e.f} : fType(C,\mathtt{f})}\ (\textsc{FldR-T}) \qquad \frac{\Gamma;\mathtt{t} \vdash \mathtt{e} : C \qquad \Gamma;\mathtt{t}' \vdash \mathtt{e}' : fType(C,\mathtt{f})}{\Gamma;\mathtt{t} \circ \mathtt{t}' \vdash \mathtt{e.f} = \mathtt{e}' : fType(C,\mathtt{f})}\ (\textsc{FldW-T})$$

Communication Typing Rules

$$\frac{\Gamma;\mathtt{t} \vdash \mathtt{e} : C \quad \Gamma;\mathtt{t}' \vdash \mathtt{e}' : C' \quad sType(C,\mathtt{s}) \bowtie \mathtt{t}'}{\Gamma;\mathtt{t} \vdash \mathtt{e.s}\{\mathtt{e}'\} : C'}\ (\textsc{SessReq-T}) \qquad \frac{\Gamma;\mathtt{t} \vdash \mathtt{e} : C \quad sType(C,\mathtt{s}) = \mathtt{t}'}{\Gamma;\mathtt{t} \circ \mathtt{t}' \vdash \mathtt{e} \bullet \mathtt{s}\{\} : rType(C,\mathtt{s})}\ (\textsc{SessDel-T})$$

$$\frac{\Gamma,\mathtt{x}:C_i;\mathtt{t}_i \vdash \mathtt{e}_i : C^* \quad (1 \le i \le \#(\overline{C})) \qquad \alpha \text{ fresh in } \overline{\mathtt{t}}}{\Gamma;\mu\alpha.?\{\overline{C \rhd \mathtt{t}[\alpha/\Box]}\} \vdash \mathtt{receiveW}(\mathtt{x})\{\overline{C \rhd \mathtt{e}}\} : C}\ (\textsc{RecW-T}) \qquad \frac{}{\Gamma;\Box \vdash \mathtt{cont} : \Box}\ (\textsc{Cont-T})$$

$$\frac{\Gamma;\varepsilon \vdash \mathtt{e} : C \qquad \Gamma;\mathtt{t}_i \vdash \mathtt{e}_i : C'^* \ (1 \le i \le \#(\overline{C})) \qquad C \not\approx \overline{C} \qquad \alpha \text{ fresh in } \overline{\mathtt{t}}}{\Gamma;\mu\alpha.!\{\overline{C \rhd \mathtt{t}[\alpha/\Box]}\} \vdash \mathtt{sendW}(\mathtt{e})\{\overline{C \rhd \mathtt{e}}\} : C'}\ (\textsc{SendW-T})$$

Fig. 12. Typing of Channel Free Expressions.

call *channel free expressions* these expressions. The environments therefore will contain also type assignment to object identifiers, this allows a simpler integration of the user expressions typing rules in the runtime typing rules. The typing judgment has the shape

$$\Gamma;\mathtt{t} \vdash \mathtt{e} : C,$$

where the environment Γ maps this, variables and objects to classes, and $\mathtt{t}$ represents the session type of the active channel. Type environments are defined by the syntax:

$$\Gamma ::= \emptyset \mid \Gamma,\mathtt{x}:C \mid \Gamma,\mathtt{this}:C \mid \Gamma,\mathtt{o}:C$$

Figure 11 defines well formed environments and subclassing in the standard way. Figure 12 gives the typing rules. We take the metavariable z to range over this, or a variable x or an object identifier $\mathtt{o}$. We use of the operation $\circ$, which is the concatenation,

provided that the first session type does not terminate with $\Box$.

$$\mathtt{t}_{\circ}\mathtt{t}' = \begin{cases} \bot, & \text{if } \mathtt{t} = \mathtt{t}''.\Box, \\ \mathtt{t}.\mathtt{t}', & \text{otherwise.} \end{cases}$$

This operation is used for example in rule (Seq-T) to represent that first the communications in $\mathtt{e}_1$ and then those in $\mathtt{e}_2$ are performed.

If C is a class we use C^* to denote either C itself or $\Box$.

To assure a safe communication between two threads we must require that their session types are *dual*, *i.e.*, that each send will correspond to a receive and vice versa. The exchanged values must also be of one of the classes expected by the receiver. Lastly all possible choices on the basis of the class of the exchanged value must continue with session types which are dual of each other. The duality is then the symmetric relation generated by:

$$\varepsilon \bowtie \varepsilon \qquad \alpha \bowtie \alpha \qquad \frac{\mathtt{t}_1 \bowtie \mathtt{t}_1{}' \quad \mathtt{t}_2 \bowtie \mathtt{t}_2{}'}{\mathtt{t}_1.\mathtt{t}_2 \bowtie \mathtt{t}_1{}'.\mathtt{t}_2{}'}$$

$$\frac{\overline{C} <: \overline{C'} \quad (C_i \Downarrow \overline{C'} = C'_j \text{ or } C'_j \Downarrow \overline{C} = C_i) \implies \mathtt{t}_i \bowtie \mathtt{t}'_j}{\mu\alpha.!\{\overline{C \rhd \mathtt{t}}\} \bowtie \mu\alpha.?\{\overline{C' \rhd \mathtt{t}'}\}}$$

where $\overline{C} <: \overline{C'}$ iff $\forall C_i\ \exists C_j.\ C_i <: C'_j$, *i.e.*, a sequence of classes is smaller than another one if each class in the first sequence is subclass of a class in the second sequence.

The duality relation is used in rule (SessReq-T) to assure that the body of the session `s` in class C and the expression $\mathtt{e}'$ will properly communicate.

In typing session delegation (rule (SessDel-T)) we take into account that the whole expression will be replaced by the result of the evaluation of the session body (cf. the reduction rule (SessDel-R)).

The rules (RecW-T) and (SendW-T) require that all possible alternative expressions which do not end in `cont` have the same class, but they can implement different communication sequences. Note that the type of `cont` allows to use it only as last statement of a "while" branch (see Proposition 8.1(2)). Note also that it is allowed that all alternatives of a "while" expression end with `cont`, determining a never ending loop. In this case the result type is an arbitrary class. We could easily avoid this by requiring that at least one of the alternatives be typed by a class.

Rule (SendW-T) prescribes that the evaluation of the value to be sent does not involve communications and the *compatibility* of the class C of this value with the classes $\overline{C}$ (notation $C \asymp \overline{C}$), *i.e.*, that C is a superclass of each class in $\overline{C}$, and that any subclass of C occurring in the class table is also a subclass of one of the $\overline{C}$.

$$C \asymp \overline{C} \text{ iff } \begin{cases} \forall C' \in \overline{C}.\ C' <: C \\ C' <: C \implies \exists C'' \in \overline{C}.\ C' <: C''. \end{cases}$$

Figure 13 defines well-formed class tables. Rule **S-ok** type-checks the session bodies with respect to a class C taking as environments the association between `this` and C.

We conclude this section by showing that in a well formed expression all occurrences of `cont` properly represent loop repetitions.

$$\frac{\{\texttt{this}:C\};\texttt{t} \vdash \texttt{e} : C'}{C'\ \texttt{t}\,\texttt{s}\,\{\texttt{e}\}:\texttt{ok in}\ C}\ (\text{S-OK}) \qquad \frac{\overline{S}:\texttt{ok in}\ C}{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\}:\texttt{ok}}\ (\text{C-OK})$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\}:\texttt{ok} \qquad \texttt{CT}:\texttt{ok}}{\texttt{CT},\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\texttt{f}\,D\,S}\}:\texttt{ok}}\ (\text{CT-OK})$$

Fig. 13. Well-formed Class Tables.

Proposition 8.1. *1. All occurrences of* `cont` *as proper subexpressions of well typed expressions are last subexpressions of some sequential compositions or form an alternative of a while expression.*
2. All occurrences of `cont` *in well typed session requests or session bodies are last subexpressions of some alternatives of while expressions.*

Proof. (1) The type of `cont` is $\Box$ and only the typing rules for bodies of while expressions and for sequential composition allow the type $\Box$.

(2) The type $\Box$ has no dual type and so the conclusion follows from (1) and from the typing rules for while expressions, session body and session request.

8.3 Typing of Runtime Expressions

In typing run time expressions we need to take into account the types of the live channels. For this reason the typing judgements for runtime expressions have the shape

$$\Gamma;\Sigma \models_{\mathtt{r}} \texttt{e} : C,$$

where Σ denote *session environments* which map live channels to session types and are defined by the following syntax:

$$\Sigma ::= \emptyset \mid \Sigma,\ \texttt{k}:\texttt{t}\,.$$

We will call *standard environments* the environments Γ.

Figure 14 defines well formed session environments and gives the typing rules for run time expressions, which differ from those for channel free expressions for having session environments instead of an unique session type. For this reason we need to extend the composition of session types to session environments. The *composition of session environments* is the natural generalisation of the composition of session types:

$$\Sigma_\circ\Sigma' = \begin{cases} \bot, & \text{if } \exists \texttt{k} \text{ with } \Sigma(\texttt{k})_\circ\Sigma'(\texttt{k}) = \bot \\ \Sigma'', & \text{otherwise, where} \\ & \mathsf{dom}(\Sigma'') = \mathsf{dom}(\Sigma) \cup \mathsf{dom}(\Sigma'), \quad \text{and} \\ & \Sigma''(\texttt{k}) \ = \ \begin{cases} \Sigma(\texttt{k})_\circ\Sigma'(\texttt{k}), & \text{if } \texttt{k} \in \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma'); \\ \Sigma(\texttt{k}), & \text{if } \texttt{k} \in \mathsf{dom}(\Sigma) \setminus \mathsf{dom}(\Sigma'); \\ \Sigma'(\texttt{k}), & \text{otherwise.} \end{cases} \end{cases}$$

Well formed Session Environments

$$\text{(SEmp)}\ \frac{}{\emptyset \vdash \mathtt{ok}} \qquad \text{(SERC)}\ \frac{\mathtt{k} \notin \mathsf{dom}(\Sigma)}{\Sigma, \mathtt{k} : \mathtt{t}\ \mathtt{ok}}$$

Runtime Typing Rules

$$\text{(Null-RT)}\ \frac{}{\Gamma; \emptyset \models_{\mathtt{r}} \mathtt{null}^{C} : C} \qquad \text{(Rec-RT)}\ \frac{}{\Gamma; \emptyset \models_{\mathtt{r}} \mathtt{c.receive} : C} \qquad \text{(Send-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C}{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{c.send(e)} : C}$$

$$\text{(Axiom-RT)}\ \frac{}{\Gamma; \emptyset \models_{\mathtt{r}} z : \Gamma(z)} \qquad \text{(Subs-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C \qquad C <: C'}{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C'} \qquad \text{(Seq-RT)}\ \frac{\Gamma; \Sigma_1 \models_{\mathtt{r}} \mathtt{e}_1 : C_1 \qquad \Gamma; \Sigma_2 \models_{\mathtt{r}} \mathtt{e}_2 : C_2^*}{\Gamma; \Sigma_1 \circ \Sigma_2 \models_{\mathtt{r}} \mathtt{e}_1; \mathtt{e}_2 : {C_2}^*}$$

$$\text{(New-RT)}\ \frac{}{\Gamma; \emptyset \models_{\mathtt{r}} \mathtt{new}\ C : C} \qquad \text{(FldR-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C}{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e.f} : \mathit{fType}(C, \mathtt{f})} \qquad \text{(FldW-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C \quad \Gamma; \Sigma' \models_{\mathtt{r}} \mathtt{e}' : \mathit{fType}(C, \mathtt{f})}{\Gamma; \Sigma \circ \Sigma' \models_{\mathtt{r}} \mathtt{e.f} = \mathtt{e}' : \mathit{fType}(C, \mathtt{f})}$$

$$\text{(SessReq-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C \quad \Gamma; \mathtt{t}' \vdash \mathtt{e}' : C' \quad \mathit{sType}(C, \mathtt{s}) \bowtie \mathtt{t}'}{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e.s}\{\mathtt{e}'\} : C'}$$

$$\text{(SessDel-RT)}\ \frac{\Gamma; \Sigma \models_{\mathtt{r}} \mathtt{e} : C \quad \mathit{sType}(C, \mathtt{s}) = \mathtt{t}'}{\Gamma; \Sigma \circ \{\mathtt{k} : \mathtt{t}'\} \models_{\mathtt{r}} \mathtt{e} \bullet \mathtt{s}\{\mathtt{k}\} : \mathit{rType}(C, \mathtt{s})}$$

$$\text{(RecW-RT)}\ \frac{\Gamma, \mathtt{x} : C_i; \Sigma, \mathtt{k} : \mathtt{t}_i \models_{\mathtt{r}} \mathtt{e}_i : C^* \quad 1 \leq i \leq \#(\overline{C}) \qquad \alpha \text{ fresh in } \overline{\mathtt{t}}}{\Gamma; \Sigma, \mathtt{k} : \mu\alpha.!\{\overline{C \triangleright \mathtt{t}[\alpha/\Box]}\} \models_{\mathtt{r}} \mathtt{k.receiveW(x)}\{\overline{C \triangleright \mathtt{e}}\} : C} \qquad \text{(Cont-T)}\ \frac{}{\Gamma; \Box \models_{\mathtt{r}} \mathtt{cont} : \Box}$$

$$\text{(SendW-RT)}\ \frac{\Gamma; \emptyset \models_{\mathtt{r}} \mathtt{e} : C' \qquad \Gamma; \Sigma, \mathtt{k} : \mathtt{t}_i \models_{\mathtt{r}} \mathtt{e}_i : C^* \quad 1 \leq i \leq \#(\overline{C}) \qquad C' \not\approx \overline{C} \qquad \alpha \text{ fresh in } \overline{\mathtt{t}}}{\Gamma; \Sigma, \mathtt{k} : \mu\alpha.!\{\overline{C \triangleright \mathtt{t}[\alpha/\Box]}\} \models_{\mathtt{r}} \mathtt{k.sendW(e)}\{\overline{C \triangleright \mathtt{e}}\} : C}$$

Fig. 14. Typing of Runtime Expressions.

Notice that in rule (SessReq-T) we are making use of the judgment $\Gamma; \mathtt{t}' \vdash \mathtt{e}' : C'$, where the expression $\mathtt{e}'$ does not contain channels, but it can contain object identifiers.

9 Fundamental Properties

In this section we state the fundamental properties that assure that our system is well founded: subject reduction and progress. Subject reduction assures that well typing is preserved during reduction, excluding the possibility of unexpected run time errors (in our case values that could not be handled by the operational semantics). Progress assures that a well typed expression will run until a meaningful value will be produced.

9.1 Subject reduction

A first observation is that a user expression which can be typed in the system of Subsection 8.2 can also be typed in the system of Subsection 8.3 once the current channel is made explicit. The proof by induction on the structure of expressions is standard.

Proposition 9.1. $\Gamma; \mathtt{t} \vdash \mathtt{e} : C$ *implies* $\Gamma; \{\mathtt{k}:\mathtt{t}\} \models_{\mathtt{r}} \mathtt{e}\lfloor \mathtt{k} \rfloor : C$.

In order to state subject reduction we start by defining a judgment, $\mathtt{t} \sqsubseteq \mathtt{t}'$, which describes that a session $\mathtt{t}'$ is at a later stage than another session $\mathtt{t}$.

$$\frac{}{\mathtt{t} \sqsubseteq \mathtt{t}}\ (\textsc{Later-1}) \qquad \frac{\mathtt{t} \sqsubseteq \mathtt{t}'' \quad \mathtt{t}'' \sqsubseteq \mathtt{t}'}{\mathtt{t} \sqsubseteq \mathtt{t}'}\ (\textsc{Later-2}) \qquad \frac{}{\mu\alpha.\dagger\{\overline{C \rhd \mathtt{t}}\} \sqsubseteq \mathtt{t}_i[\mu\alpha.\dagger\{\overline{C \rhd \mathtt{t}}\}/\alpha]}\ (\textsc{Later-3}) \qquad \frac{\mathtt{t} \sqsubseteq \mathtt{t}'}{\mathtt{t}.\mathtt{t}'' \sqsubseteq \mathtt{t}'.\mathtt{t}''}\ (\textsc{Later-4})$$

Notice that not all types which are related by $\sqsubseteq$ describe session evaluations: for example, $\mu\beta?\{D \rhd \varepsilon\}$ is *not* a possible later stage of $\mu\alpha?\{C \rhd \alpha\}.\mu\beta?\{D \rhd \varepsilon\}$.

We now also extend the definition of $\_ \sqsubseteq \_$ to session environments in the obvious way:

$$\Sigma \sqsubseteq \Sigma' \quad \text{iff} \quad \begin{cases} \mathsf{dom}(\Sigma) \subseteq \mathsf{dom}(\Sigma'), & \text{and} \\ \forall \mathtt{k} \in \mathsf{dom}(\Sigma) : \ \Sigma'(\mathtt{k}) \sqsubseteq \Sigma(\mathtt{k}). \end{cases}$$

Below we define the judgment $h \vdash \mathtt{v} \ : \ C$ which expresses agreement of a value with a class w.r.t. a heap, and the judgment $h \vdash \overline{\mathtt{v}} \ : \ \mathtt{t}$ which expresses agreement of a sequence of values with a session type wrt to a heap:

$$\frac{h(\mathtt{v}) = (C', \_) \quad C' <: C}{h \vdash \mathtt{v} \ : \ C}\ (\textsc{AggrValue}) \qquad \frac{}{h \vdash () \ : \ \mathtt{t}}\ (\textsc{AggrQueue-1}) \qquad \frac{h \vdash \overline{\mathtt{v}} \ : \ \mathtt{t}_i.\mathtt{t} \quad h(\mathtt{v}) = (C, \_) \quad C \Downarrow \overline{C} = C_i}{h \vdash \mathtt{v} :: \overline{\mathtt{v}} \ : \ ?\{\overline{C \rhd \mathtt{t}}\}.\mathtt{t}}\ (\textsc{AggrQueue-2})$$

Notice that in the judgement $\overline{\mathtt{v}}$: t the session type t is intended as the type of a session that can safely *accept* the values in $\overline{\mathtt{v}}$.

A heap is well formed iff all every live channel has its dual, and only one of them is associated to a non empty queue of values. Moreover all object fields must contain objects of the expected classes.

$$wf(h) \quad \text{iff} \quad \begin{cases} \mathtt{k} \in \mathsf{dom}(h) \ \Rightarrow \ \tilde{\mathtt{k}} \in \mathsf{dom}(h), \\ h(\mathtt{k}) \neq () \ \Rightarrow \ h(\tilde{\mathtt{k}}) = (), \\ h \vdash \mathtt{o} \ : \ C, \ \ fType(C, \mathtt{f}) = C' \ \Rightarrow \ h \vdash h(\mathtt{o})(\mathtt{f}) \ : \ C'. \end{cases}$$

A standard environment Γ, a session environment Σ, and a heap h agree, iff the heap is well formed, the classes of objects in the heap are subclasses of the classes associated to them by the standard environment, all subjects of the session environment occur in the heap and the session types agree with the queues of values.

$$wf(\Gamma;\Sigma;h) \quad \text{iff} \quad \begin{cases} wf(h), \\ \forall \mathtt{o} \in \mathsf{dom}(\Gamma):\ C <: \Gamma(\mathtt{o}) \quad \text{where } h(\mathtt{o}) = (C, -) \\ \mathsf{dom}(\Sigma) \subseteq \mathsf{dom}(h), \\ \forall \mathtt{k} \in \mathsf{dom}(\Sigma): \quad h \vdash h(\mathtt{k}) \ : \ \Sigma(\mathtt{k}). \end{cases}$$

We are now able to state the Type Preservation Theorem.

Theorem 9.2 (Type Preservation). *If $wf(\Gamma;\Sigma;h)$ and $\Gamma;\Sigma \models_{\mathtt{r}} \mathtt{e} : C$ then*

- $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$ *implies* $\exists \Sigma'$, Γ' *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \sqsubseteq \Sigma'$, *and* $wf(\Gamma';\Sigma';h')$, *and* $\Gamma';\Sigma' \models_{\mathtt{r}} \mathtt{e}' : C$.
- $\mathtt{e}, h \longrightarrow \mathtt{e}_1 \mid \mathtt{e}_2,\ h'$ *implies* $\exists \Sigma_1, \Sigma_2, C', \mathtt{k}, \mathtt{t}, \mathtt{t}'$ *with* $\Gamma;\Sigma_1 \models_{\mathtt{r}} \mathtt{e}_1 : C$, *and* $\Gamma;\Sigma_2 \models_{\mathtt{r}} \mathtt{e}_2 : C'$, *and* $wf(\Gamma;\Sigma_1;h')$, *and* $wf(\Gamma;\Sigma_2;h')$, *and*
  - *either* $\Sigma(\mathtt{k}) = \mathtt{t}.\mathtt{t}'$ *and* $\Sigma_1 = \Sigma \backslash \mathtt{k} \cup \{\mathtt{k} : \mathtt{t}'\}$ *and* $\Sigma_2 = \{\mathtt{k} : \mathtt{t}\}$ *and* $h' = h[\mathtt{c} \mapsto ()]$ *for some* $\mathtt{c} \notin h$,
  - *or* $\Sigma_1 = \Sigma, \mathtt{k} : \mathtt{t}$, *and* $\Sigma_2 = \{\tilde{\mathtt{k}} : \mathtt{t}'\}$, *and* $\mathtt{t} \bowtie \mathtt{t}'$, *and* $h' = h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]$ *for some* $\mathtt{k}, \tilde{\mathtt{k}} \notin h$.

The proof of this theorem can be given with a technique similar to that developed in [4].

9.2 Progress

The *progress property* assures that computations of a well typed expression cannot go wrong, *i.e.*, either a parallel of values is obtained, or a field/session of a null object is required, or a further reduction step is feasible.

In the remaining of the present section we will outline the progress proof. The present progress proof is based on those of [4, 7] but need major modifications.

Let's first give a more formal definition of progress. An expression $\mathtt{e}$ is *initial* if it is a user expression and $\emptyset; \varepsilon \models_{\mathtt{r}} \mathtt{e} : C$ for some C. We will consider here only computations starting from initial expressions.

Definition 9.3.
- *An expression* $\mathtt{e}$ *is a* nullpointer failure *if* $\mathtt{e}$ *has one of the following forms:*
  - $E[\mathsf{null}^C.\mathtt{f}\,]$;
  - $E[\mathsf{null}^C.\mathtt{s}\,\{\mathtt{e}\}\,]$;
  - $E[\mathsf{null}^C \bullet \mathtt{s}\,\{\mathtt{k}\}]$.
- *An expression* $\mathtt{e}$ *has the* progress *property if* $\mathtt{e}, [\,] \longrightarrow^* P, h$ *implies that*
  - *either in P all expressions are values, i.e., $P = \prod_{0 \le \mathtt{i} < \mathtt{n}} \mathtt{v}_\mathtt{i}$;*
  - *or $P, h \longrightarrow P', h'$;*
  - *or there is a subexpression of P that is a nullpointer failure.*

We call *communication* (comm for short) channels the channels which are created in rule (SessReq-R) and *synchronization* (sync for short) channels the ones created in rule (SessDel-R).

In the following we convene that the fresh communication channels created reducing a thread take successive indexes according to the order of creation, i.e. they are named $\mathtt{k}_0$, $\mathtt{k}_1, \ldots$. This means that if $P,h \longrightarrow\!\!\!\!\rightarrow Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$ and $\mathtt{k}_i$ is a channel created in the reduction $P,h \longrightarrow\!\!\!\!\rightarrow Q,h'$, and $\mathtt{k}_j$ is a channel created in the reduction $Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$, then $i < j$.

Synchronization channels are created only in evaluating a delegation expression. We convene to name them with a couple of indexes: the index of the channel they send (*primary* index) and a progressive index to individuate them (*secondary* index). So $\mathtt{c}_i^0$, $\mathtt{c}_i^1, \ldots$ are the sync channels sending channels $\mathtt{k}_i$ or $\tilde{\mathtt{k}}_i$. It will also be useful to decorate sync channels with the same duality as the comm channels send in the corresponding delegation session. So $\tilde{\mathtt{c}}_i^j$ is a sync channel created in opening a delegation session on channel $\tilde{\mathtt{k}}_i$.

The *subject* of a communication expression is the channel specified in its syntax on which the communication takes place. The index of a communication expression is the (primary) index of its subject.

Definition 9.4. *1. Let* $\mathtt{e}$ *be an expression and* $\mathtt{e}_1, \mathtt{e}_2$ *be two subexpressions of* $\mathtt{e}$. *We say that* $\mathtt{e}_1$ precedes $\mathtt{e}_2$ *in* $\mathtt{e}$ *if, for some contexts* $C[-]$, $E[-]$ *and* $C'[-]$ *we have* $\mathtt{e} = C[\mathtt{e}']$ *and* $\mathtt{e}' = E[\mathtt{e}_1] = C'[\mathtt{e}_2]$.

2. A channel ψ *(where* ψ *can be either a comm or a sync channel) is the* active *channel in* $\mathtt{e}$ *if* $\mathtt{e} = E[\mathtt{e}_1]$, ψ *is the subject of a subexpression* $\mathtt{e}_2$ *of* $\mathtt{e}$ *such that* $\mathtt{e}_1$ *precedes* $\mathtt{e}_2$ *in* $\mathtt{e}$ *and for no communication expressions* $\mathtt{e}'$ *we have that* $\mathtt{e}_1$ *precedes* $\mathtt{e}'$ *and* $\mathtt{e}'$ *precedes* $\mathtt{e}_2$.

Informally the active channel in an expression is the channel on which the next communication will be performed. Obviously if $\mathtt{e}$ is a communication expression the active channel of $E[\mathtt{e}]$ is the subject of $\mathtt{e}$ itself. We say that a subexpression $\mathtt{e}'$ is the *last* subexpression in $\mathtt{e}$ if $\mathtt{e}'$ does not precede any other subexpression of $\mathtt{e}$.

Lemma 9.5. *Let* $\mathtt{e}, [\,] \longrightarrow^* P,h$ *where* $\mathtt{e}$ *is an initial expression. Then*

1. *No thread in P can contain occurrences of both* $\mathtt{k}$ *and* $\tilde{\mathtt{k}}$ *for some channel* $\mathtt{k}$.
2. *A comm channel can be active in only one thread of P.*
3. *A sending expression with a sync subject* $\mathtt{c}_i^j$ *can occur only as last expression of a thread in P which was opened by a delegation expression on* $\mathtt{k}_i$.

Proof. By induction on reduction.

Lemma 9.6. *Let* $\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}' \mid P,h$ *where* $\mathtt{e}$ *is an initial expression. Then the index of the active channel in* $\mathtt{e}'$ *is greater than or equal to the (primary) index of any other channel occurring in* $\mathtt{e}'$.

Proof. Induction on reduction. Note that when a new thread is opened by delegation the sync channel takes the same index of the sent channel.

Theorem 9.7 (Progress). *If* $\mathtt{e}$ *is an initial expression, then* $\mathtt{e}$ *has the progress property.*

Proof. If $\mathtt{e}$ is initial we have $\emptyset;\mathtt{e} \vdash \mathtt{e} : C$. Assume that $\mathtt{e}$ does not have the progress property. Then $\mathtt{e},[\,] \longrightarrow^* \mathtt{e}_1 \mid ... \mid \mathtt{e}_n, h$ which is irreducible and is not a parallel composition of values and it does not contain nullpointer failures. By the subject reduction property we have that there are $\Gamma, \Sigma_1, ..., \Sigma_n$ and $C_1, ..., C_n$ such that $wf(\Gamma;\Sigma_i;h)$ and $\Gamma;\Sigma_i \models_{\mathtt{r}} \mathtt{e}_i : C_i$ for all $1 \leq i \leq n$. We can assume without loss of generality that $\mathtt{e}_1, ..., \mathtt{e}_m$ are not values. The evaluation can only be stopped by a receiving expression waiting for data in the associated channel. So for all $1 \leq i \leq m$ we must have $\mathtt{e}_i = E[\mathtt{e}'_i]$ where $\mathtt{e}'_i$ is a receiving expression whose subject is the active channel in $\mathtt{e}_i$. Let's say that $\mathtt{e}_i$ is *blocked* on the subject of $\mathtt{e}'_i$. First note that in $\mathtt{e}_1, ..., \mathtt{e}_m$ the following property hold:

(**P**) If an expression $\mathtt{e}_i$ is blocked on a sync channel $\mathtt{c}^l_p$ then there must exist another expression $\mathtt{e}_j$ which is blocked on a comm channel $\mathtt{k}_q$ with $q \geq p$.

Proof. Assume that $\mathtt{e}_i$ is blocked on a sync channel $\mathtt{c}^l_p$. Then the (unique) sending expression on $\mathtt{c}^l_p$ must occur in some other expression $\mathtt{e}_j$ which, by Lemma 9.6, is blocked on a communication expression whose index is greater than or equal to p. If its subject is a is a comm channel we are done. Otherwise an iterative argument concludes the proof.

Let's consider an expression $\mathtt{e}'_i$ such that its subject is a channel $\mathtt{k}_p$ where p is the highest among the index of channels which are subjects of $\mathtt{e}'_1, \ldots, \mathtt{e}'_m$. Property (**P**) assures that such an expression must exist. Since the corresponding queue in the heap is empty and all expressions are well typed and consistent with the heap, then there must exist a corresponding sending expression $\mathtt{e}'$ whose subject is $\tilde{\mathtt{k}}_p$ and which occur in $\mathtt{e}_j$ for some $j \neq i$ by Lemma 9.5.1 . Now we have two cases:

1. $\mathtt{e}_j$ is blocked on a receiving comm expression on $\tilde{\mathtt{k}}_p$ (by definition p is the highest index of blocked channels);

2. $\mathtt{e}_j$ is blocked on a receiving expression on a sync channel $\tilde{\mathtt{c}}^q_p$.

Case 1 is impossible by the well typing and well forming heap conditions. In case 2, by Lemma 9.5.3 we have that there must exist a sending expression with subject $\tilde{\mathtt{c}}^q_p$ which is the last expression of a blocked expression $\mathtt{e}_l$ opened by a delegation on $\tilde{\mathtt{k}}_p$. Since p is the greatest index, then the active channel of $\mathtt{e}_l$ must be either $\tilde{\mathtt{k}}_p$ or a sync channel $\tilde{\mathtt{c}}^l_p$. In the former case we conclude as in case 1, in the latter case we conclude by an iterative argument.

10 Summary, Conclusions, and Further Work

We believe that a good combination of session types with object oriented programming will be a crucial vehicle for the development of safe concurrent and/or distributed, collaborating programs. We believe that such a combination should be achieved thorough *amalgamation* rather than extension of the paradigms.

We have suggested such an amalgamation, whereby sessions subsume methods, threads consist of the execution of session bodies on objects and communicate with each other through asynchronously sending/receiving objects on channels, where all choices are based on the classes of objects, and where data are exchanged between

threads running sessions via channels using asynchronous communication, and where sessions can be delegated to other sessions.

One major deviation of STOOP wrt other sessions languages, is that STOOP sessions are not first class, and instead we are using the concept of delegation. This significantly simplifies the system, although it restricts the expressive power. We believe that this restriction has small practical relevance, *i.e.*, that most useful applications can be written with this restriction.

We want to investigate the expressive power of the paradigm in terms of more examples. We also want to explore how slightly extend the expressivity of delegation, so that it supports an initial and a final dialogue before and after the delegation. In terms of the example from Section 3, the seller might have an initial dialogue with the bank before delegating *e.g.*, giving precise instruction for the credit check, and might have a final dialogue after the credit check, *e.g.*, receiving more detailed information about the mode of payment.

After that, we plan to explore full language designs based on STOOP, *e.g.*, adding concurrency and synchronization for kernel applications, or, alternatively considering distribution and data duplication issues for web services applications.

Acknowledgements

We are greatly indebted to Dimitris Mostrous for many interesting discussions which, among others, lead to the idea of amalgamating sessions with methods. We are very grateful to Nobuko Yoshida for extensive feedback.

References

1. Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.
2. Marco Carbone, Kohei Honda, and Nobuko Yoshida. A Theoretical Basis of Communication-centered Concurrent Programming. Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report.
3. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In Rocco De Nicola, editor, *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.
4. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer-Verlag, 2007.
5. Ferruccio Damiani, Elena Giachino, Paola Giannini, Nick Cameron, and Sophia Drossopoulou. A State Abstraction for Coordination in Java-like Programming. *Acta Informatica*, 2007. To appear.
6. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. In Frank de Boer and Marcello Bonsangue, editors, *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer-Verlag, 2007.

7. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
8. Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. A Distributed Object Oriented Language with Session Types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.
9. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In Willy Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
10. Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In Michael Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.
11. Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
12. Simon Gay, Vasco T. Vasconcelos, and António Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.
13. Kohei Honda. Types for Dyadic Interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
14. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In Chris Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
15. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web Services, Mobile Processes and Types. *EATCS Bulletin*, 91:160–188, 2007.
16. Dimitris Mostrous and Nobuko Yoshida. Two Sessions Typing Systems for Higher-Order Mobile Processes. In Simona Ronchi Della Rocca, editor, *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer-Verlag, 2007.
17. Stephen Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2):14–23, 2006.
18. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
19. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In Antonio Brogi and Jean-Marie Jacquet, editors, *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.
20. Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theorical Computer Science*, 368(1-2):64–87, 2006.
21. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.