Session and Union Types for Object Oriented Programming

Session and Union Types for Object Oriented Programming*

Lorenzo Bettini[1], Sara Capecchi[1], Mariangiola Dezani-Ciancaglini[1], Elena Giachino[1], and Betti Venneri[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Dipartimento di Sistemi e Informatica, Università di Firenze

Dedicated to Ugo Montanari on the Occasion of his 65th Birthday

Abstract. In network applications it is crucial to have a mechanism to guarantee that communications evolve correctly according to the agreed protocol. Session types offer a method for abstracting and validating structured communication sequences (sessions). In this paper we propose union types for refining and enhancing the flexibility of session types in the context of communication centred and object oriented programming. We demonstrate our ideas through an example and a calculus formalising the main issues of the present approach. The type system garantees that, in well-typed executable programs, after a session has started, the values sent and received will be of the appropriate type, and no process can get stuck forever.

Keywords: Sessions, Object Oriented Programming, Session Types, Union Types.

1 Introduction

Writing safe communication protocols has become a central issue in the theory and practice of concurrent and distributed computing. The actual standards still leave to the programmer much of the responsibility in guaranteeing that the communication will evolve as agreed by all the involved agents.

Session types [26, 27] offer a method for abstracting and validating structured communication sequences (*sessions*). This is achieved 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. Since the specification of a session is a type, the conformance test of programs with respect to specifications becomes type checking.

The popularity of class-based *object oriented* languages justifies the interest in searching for class definitions which naturally include communication primitives. For

* This work has been partially supported by MIUR project EOS DUE and by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09).

P. Degano et al. (Eds.): Montanari Festschrift, LNCS 5065, pp. 659–680, 2008.

this reason, an amalgamation of communication centred and object oriented programming has been first proposed in [18], where methods are unified with sessions and choices are based on the classes of exchanged objects.

Union types have been shown useful for enhancing the flexibility of subtyping in various settings [1, 21, 11, 10, 29]. For example a bank can answer `yes` or `not` according to the balance between an account and an item price. If `yes` and `not` are objects of classes `OK` and `NoMoney` respectively, then the class of the object `answer` is naturally the union of the two classes `OK` and `NoMoney`, *i.e.* $\texttt{OK} \vee \texttt{NoMoney}$. Without union types typing `answer` would require a superclass of both `OK` and `NoMoney` to be already defined, and this superclass could include unwanted objects. With union types we can express communications between parties which manipulate heterogeneous objects just by sending and receiving objects which belong to subclasses of one of the classes in the union. In this way the flexibility of object-oriented depth-subtyping is enhanced, by strongly improving the expressiveness of choices based on the classes of sent/received objects.

The aim of the present paper is to discuss and formalise the use of union types for session-centred communications in a core object-oriented calculus. A preliminary version of the basic calculus, without union types, is defined in [18]. In the present paper, the calculus of [18] is formally revised, so that typing and semantics are rather cleaner and simpler. Furthermore, the extension to union types, which is the main novelty of the present proposal, poses specific problems in formulating reduction and typing rules to ensure that communications are safe while flexible.

We first present an example which illustrates the main features of our approach and then we formalise these features through a featherweight representation. We call $\mathsf{SAM}^{\vee}$ (*S*essions *A*malgamated with *M*ethods plus union types) the language of the example and $\mathscr{F}\mathsf{SAM}^{\vee}$ the formalising calculus.

$\mathsf{SAM}^{\vee}$ *Overview.* $\mathsf{SAM}^{\vee}$, as the language of [18], is concerned with the amalgamation of the object oriented features with the session part, but it is agnostic w.r.t. to the remaining features of the language, such as whether the language is distributed or concurrent, and the features for synchronisation.

In $\mathsf{SAM}^{\vee}$, sessions and methods are "amalgamated": invocation takes place on an object and the execution takes place immediately and concurrently with the requesting thread (indeed, $\mathsf{SAM}^{\vee}$ is multi-threaded and the communication is asynchronous). Thus, it keeps the method-like invocation mechanism while involving two threads, typical of session based communication mechanisms. The body is determined by the class of the receiving object (avoiding in this way the usual branch/select primitives [27]), and any number of communications interleaved with computation is possible. Sessions are defined in a class, which can have also fields. We believe that the above amalgamated model of session naturally reflects our intuition of services. Furthermore, it can neatly encode "standard" methods.

A thread can make a session request through $\texttt{e.s}\{\texttt{e}'\}$, where `e` is an expression denoting an object, `s` is the name of a session defined in the object's class; then, `e` is evaluated to an object `o`, and the session body of `s` in `o`'s class is executed concurrently with $\texttt{e}'$, introducing a new pair of fresh channels `k` and $\tilde{\texttt{k}}$ (one for each communication

direction) to perform communications between the session body and $\mathtt{e}'$[1]. Notice that channels are implicit, and are not written by the programmer. At every step, in each thread, there is only one single active channel on which communications are performed.

The expressions $\mathtt{send}(\mathtt{e})$ and $\mathtt{rec}(\mathtt{x})$ send and receive objects on the active channel, respectively. The expression $\mathtt{sendC}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \ldots \,[]\, \mathtt{C}_n \Rightarrow \mathtt{e}_n\}$ (where C means Case) evaluates $\mathtt{e}$ to an object and sends it on the active channel, and then continues with $\mathtt{e}_i$, where $\mathtt{C}_i$ is the class that best fits the class of the object sent. The counter part of $\mathtt{sendC}$ is the expression $\mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \ldots \,[]\, \mathtt{C}_n \Rightarrow \mathtt{e}_n\}$, where the choice is based on the class of the object received. The expression $\mathtt{sendW}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \ldots \,[]\, \mathtt{C}_n \Rightarrow \mathtt{e}_n\}$ (where W means While) is similar to $\mathtt{sendC}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \ldots \,[]\, \mathtt{C}_n \Rightarrow \mathtt{e}_n\}$, except that it allows for enclosed $\mathtt{cont}$, which continues the execution at the nearest enclosing $\mathtt{sendW}$. The expression $\mathtt{recW}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \ldots \,[]\, \mathtt{C}_n \Rightarrow \mathtt{e}_n\}$ has the obvious meaning. Finally, $\mathtt{e} \bullet \mathtt{s}\{\}$ delegates the current session to the object resulting from the evaluation of $\mathtt{e}$; the body of the session $\mathtt{s}$ 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.

Related Papers. We describe $\mathscr{F}\mathsf{SAM}^{\vee}$ following *Featherweight Java* [30], which today has become a standard for class based object calculi.

Session Types have been first introduced to model communication protocols between π-calculus processes [26, 32, 27]. They have been made more expressive by enriching them with correspondence assertions [3], subtyping [24], bounded polymorphism [23] and safer by assuring deadlock-freedom [14]. More recently session types have been extended to multi-party communications [2, 9].

Session types have been developed also for CORBA [33], for functional languages [25, 34], for boxed ambients [22], for the W3C standard description language for Web Services CDL [8, 35, 31, 28], for operating systems [19], and for object oriented programming languages [17, 16, 13, 15, 18, 6]. In [6] generic types are added to a language/-calculus based on the approach of [18]; independently from the different typing extensions, $\mathscr{F}\mathsf{SAM}^{\vee}$ improves the definition of the calculi of [18] and [6], both in syntax and in operational semantics.

Union types have been proved useful for functional languages [1, 11], for object-oriented languages [29], for languages manipulating semi-structured data [21] and for the π-calculus [10]. We will tell more on the relations between the present paper and [29] at the end of Subsection 6.1.

There are many concurrent object-oriented languages and calculi in the literature; for this topic we refer to the related work section of [20].

Paper Structure. In Section 2 we describe $\mathsf{SAM}^{\vee}$ in terms of an example. We then proceed by formalising the calculus $\mathscr{F}\mathsf{SAM}^{\vee}$, its typing and semantics. Section 7 draws some future work directions.

2 An Example

In this section, we describe $\mathsf{SAM}^{\vee}$ through an example, which expresses a typical collaboration pattern, *c.f.* [35, 7, 8], and which refines the example of [18]. This simple

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

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

Fig. 1. Session types for the buyer-seller example

protocol contains essential features which demonstrate the expressiveness of the idioms of SAM$^{\vee}$. 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, SAM$^{\vee}$ can delegate the current session through a session call as in [18,6]. The negotiation allows several rounds: the buyer may either accept the price, or break the negotiation, or require a better deal by sending different kinds of answers; in the latter case, the seller might respond by sending a better price, or might break the negotiation sending a negative answer. Thus, branch selection in control structures is based on the dynamic class of an object sent.

The session types `Shopping_ST` and `Sell_ST` (see Figure 1) describe the communication pattern between the `Buyer` and the `Seller`.

The session type `Shopping_ST` 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` $\Rightarrow$ ..., `NoDeal` $\Rightarrow$ ..., `MakeOnOffer` $\Rightarrow$ ... $\}$ 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 negotiation resumes on the basis of it, whereas if it is a `NoDeal`, the negotiation ends.

Note that in both `Shopping_ST` and `Sell_ST` the recursion variable is nested inside multiple choices, so that this behaviour could not have been expressed using regular expressions as in [15]. The use of recursive types has also other advantages, like that of allowing iterative expressions with multiple exit points and multiple recursions. In

```
1  class Buyer {
2   AccntNr accnt; Seller seller;
3
4   String∨Date Shopping_ST shopping
5       {  Item prodId := ....;
6          seller.sell{
7             send(prodID);
8             Money price := rec;
9             examinePrice{send(price);
10                         OK∨NoDeal∨MakeAnOffer resp := rec};
11            sendW(resp){
12               OK ⇒ { send(accnt); send(price);
13                      recC(x) { OK ⇒ Date delivDate:=rec; []
14                                     NoMoney ⇒ new String("no money"); } }[]
15               NoDeal ⇒ new String("refusing proposed price") []
16               MakeAnOffer ⇒ {
17                       recC(x) { Money ⇒ examinePrice{send(x);
18                                                      resp := rec};
19                                                      cont; []
20                                        NoDeal ⇒ new String("offer refused")}
21                               }
22                     } //end of session call sell
23                } //end of session shopping
24
25  Object ExaminePrice_ST examinePrice
26      { Money price := rec
27        ... //code for decision
28        send(resp)
29       } // end of session examinePrice
30  }
```

Fig. 2. The class `Buyer`

Figure 2 we show the implementation of the class `Buyer`. It has the fields `accnt` and `seller`, which will contain the account number and the `seller` used to buy products.

The class `Buyer` supports two sessions called `shopping` and `examinePrice`. Session `shopping` has session type `Shopping_ST` and return type `String∨Date`. The union type `String∨Date` describes the possible results of the negotiation: in case of success the session ends returning the date of the delivery of the item; in case of failure it returns a string describing the reason. In the body of this session the desired product is determined and stored in `prodId` (line 5). Then, a session request is made to the `seller` to run session `sell` (line 6). 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 7), receive a `Money` and store it in `price` (line 8). Based on its value the `buyer` will calculate his response calling session `examinePrice` which returns the answer in `resp` (lines 9 and 10). Let us notice that `resp`'s type is the union of all the possible answers' type. On line 11, the `buyer` enters a loop with `sendW`, where he sends `resp`, and branches according to its class. If `resp` is `OK`, indicating acceptance of the price,

```
1  class Seller {
2    Bank bank;
3
4    String∨Item Sell_ST sell
5      { Item prodID := rec;
6        Money price=...;
7        send(price);
8        recW( x ){
9            OK ⇒ { sendC( bank•check{ } )
10                    {OK ⇒ calDelDate{send(prodID);
11                                    Date date := rec};
12                          send(date); prodID []
13                     NoMoney ⇒ new String("failed bank transaction");
                             } } []
14           NoDeal ⇒ new String("proposal refused by buyer") []
15           MakeAnOffer ⇒ { calNewPrice{send(price);
16                            Money∨NoDeal resp := rec} ;
17                            sendC(resp){ Money ⇒ cont []
18                                       NoDeal ⇒ new String("refusing
                                             proposed price"); } }
19         }
20     } //end of session sell
21
22   Object CalDelDate_ST calDelDate
23     { Item item := rec
24       ... //code for calculate date
25       send(date)
26     } // end of session calDelDate
27
28   Object CalNewPrice_ST calNewPrice
29     { Money price := rec
30       ... //code for response (if the answer is positive then the field
31         //price is updated with the new price)
32       sendC(resp)
33     } // end of session calNewPrice
34   }
```

Fig. 3. The class `Seller`

then the `buyer` will send his account 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). In this case the reason of failure is stored in the string `failure`. 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, indeed Line 17 contains `recC` 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 session `examinePrice` is called, which returns the `buyer`'s reaction in `resp`, and the recursion will continue (line 19). If the value received is `NoDeal`, then the loop will be abandoned.

```
1  class Bank {
2      Ok∨NoMoney CreditCheck_ST check
3         {  AccntNr accnt := rec;
4            Money amt := rec;
5            .... // code for check
6            If(response) then new Ok else new NoMoney;
7         }// end of session check
8    }
```

Fig. 4. The class `Bank`

Notice that in order to get an arbitrary number of repetitions, it is crucial to allow objects of different classes to be sent in the different iterations of while loops.

The session type `Sell_ST` describes the protocol from the point of view of the `seller`, and is "dual" to `Shopping_ST`. We now consider the class `Seller`, from Figure 3. 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 `recW` loop, which is the counterpart to the `sendW` loop from `shopping` and performs all the `seller`'s negotiation. The interesting 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`. At the end of the execution of `check` the session will continue according to the bank answer (`OK` or `NoMoney`).

The session type for `check` from class `Bank` in Figure 4 is the receipt of a `AccntNr` and a `Money` followed by sending either `OK`, or a `NoMoney` object. Note that the 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 union of the types of the possible answers, *i.e.* `OK∨NoMoney`.

Notice that the sessions `examinePrice`, `calNewDate` and `calNewPrice` are examples of the implementation in $\mathsf{SAM}^{\vee}$ of methods, since they start by receiving arguments and after elaborating them send a result.

3 Syntax

This section presents the syntax of $\mathscr{F}\mathsf{SAM}^{\vee}$ (Figure 5), a minimal concurrent and imperative core calculus, based on *Featherweight Java* [30] (abbreviated with FJ). $\mathscr{F}\mathsf{SAM}^{\vee}$ supports the basic object-oriented features and session request, session delegation, branching sending/receiving and loops. In details, $\mathscr{F}\mathsf{SAM}^{\vee}$ encompasses the following linguistic features: basic object oriented expressions, session bodies and communication constructs that combine send/receive with branching and loops.

We use grey to indicate expressions that are produced during the reduction process, but do not occur in the source code of a program. We also use the standard convention of denoting with $\overline{\xi}$ a sequence of elements $\xi_1, ..., \xi_n$.

Union types are defined as in [29]: they are built out of class names by the union operator (denoted by $\vee$).

Programs are defined from a collection of classes. The metavariables `C` and `D`, possibly with subscripts, range over class names. Each class has a name, a list of *fields* of the

(union type)	T	::=	C | T∨T
(class)	L	::=	class C ◁ C { $\overline{\texttt{T f}}$; $\overline{\texttt{S}}$ }
(session)	S	::=	T t s{ e }
(expression)	e	::=	x | this | cont | o | e; e | e.f:= e | e.f | new C($\overline{\texttt{e}}$)
		|	e.s{e} | e•s{ k }
		|	k. sendC(e){C ⇒ e [] C ⇒ e}
		|	k. recC(x){C ⇒ e [] C ⇒ e}
		|	k. sendW(e){C ⇒ e [] C ⇒ e}
		|	k. recW(x){C ⇒ e [] C ⇒ e}
(parallel threads)	*P*	::=	e | *P* || *P*

Fig. 5. Syntax, where syntax occurring only at runtime appears shaded. Syntax for session types t is in Figure 9.

form $\overline{\texttt{T f}}$, where f represents the field name and T its type, and a list of *sessions* of the form T t s{ e }, where T is the return type, t the session type, s the session name, and e the session body. For the sake of conciseness the symbol ◁ represents class extension, as in [30]. All classes are defined as extensions of the topmost class Object.

Expressions include variables, that are both standard term variables x and the special variables this and cont. The variable this is considered implicitly bound in any session declaration. Instead, sendW and recW are the only binders for cont, that represents the continuation by recursive computation. Let us notice that free occurrences of cont in e are not bound in the expression sendW(e){...}: actually no occurrence of cont can appear in e if this expression is typable (see rule SENDW-T in Figure 11).

In a *session request* e.s{e′} we call the expression e′ the *co-body* of the request (since it will be evaluated concurrently with the body of requested session).

In the *session delegation* expression, e•s{k}, the channel k is added by the operational semantics in order to keep track of the channel to pass to the delegated session.

Channels are implicit in the source language syntax. At runtime, communication channels k are introduced at each new session request. We denote the dual with $\tilde{..}$, where $\tilde{\texttt{k}}$ is again a runtime channel, and where $\tilde{..}$ is an involution: $\tilde{\tilde{\texttt{k}}} = \texttt{k}$. Whenever a thread uses a channel k, the other participant in the communication uses its dual $\tilde{\texttt{k}}$. The operational semantics associates to k and $\tilde{\texttt{k}}$ two different queues of messages; when a thread, which uses the channel k, wants to receive a message it will inspect the queue associated to k, while, when it sends a message it will add it to the queue associated to $\tilde{\texttt{k}}$ (see Section 5).

The body of a *communication expression* is a pair of alternatives {C$_1$ ⇒ e$_1$ [] C$_2$ ⇒ e$_2$}, whose choice depends on the class of the object that is sent or received. In particular, in case of sendW and recW the expressions e$_i$ can contain cont, representing recursive computations.

With respect to SAM$^\vee$ we only have binary choices as bodies of communication expressions in $\mathscr{F}$SAM$^\vee$, since the other forms can be encoded with them. First of all, a unary choice {C ⇒ e} can be simply encoded as {C ⇒ e [] C ⇒ new Object()}, since sending or receiving an object of class C will always choose the first alternative. With

unary choices we can encode the two communication constructs used in the example, i.e., `send` for sending, and `rec` for receiving, that we omit from $\mathscr{F}\mathsf{SAM}^{\vee}$: the expression $\mathtt{sendC(e)\{Object \Rightarrow new\ Object()\}}$ encodes $\mathtt{send(e)}$ and in a similar way, $\mathtt{recC(x)\{Object \Rightarrow x\}}$ encodes `rec`.

With binary choices we can also encode n-ary choices for $n > 2$, getting in this way the constructs used in the example of Section 2. The informal idea of such encoding, given the set of classes to be used in the choices, is to take the first (in the left to right order) relative minimum from this set and use it to define the first branch; the second branch will use `Object` and we iterate this procedure to write the expression of this second branch, until we remain with only two choices. Thus, for instance, consider the choice $\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2 \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}_3\}$, where $\mathtt{C}_1$ is not related to the other classes and $\mathtt{C}_2$ is a subclass of $\mathtt{C}_3$; we can encode this choice with the (nested) binary choice: $\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{Object} \Rightarrow \{\mathtt{C}_2 \Rightarrow \mathtt{e}_2 \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}_3\}\}$. Notice that this encoding is correct also if $\mathtt{C}_2 = \mathtt{C}_3$.

The types used for selecting branches in a choice are class names. This simplifies the formal treatment and the proofs, but, again, we can encode choices with arbitrary union types by n-ary choices in a straightforward way; e.g., $\{\mathtt{C}_1 \vee \mathtt{C}_2 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}'\}$ is encoded as $\{\mathtt{C}_1 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}'\}$.

A *runtime expression* is either a user expression (i.e. an expression in Figure 5 without shaded syntax) or an expression containing channels and/or object indentifiers. Furthermore, threads of runtime expressions can occur at runtime (see the operational semantics). Parallel threads are ranged over by P. Fully evaluated objects will be represented by object identifiers denoted by `o`.

The main novelty of $\mathscr{F}\mathsf{SAM}^{\vee}$ w.r.t. FJ is that session invocation can involve the creation of concurrent and communicating threads. Other minor differences are: we do not have cast and overriding, which are orthogonal to our approach; we do not have explicit constructors, then in the object instantiation expression $\mathtt{new\ C}(\overline{\mathtt{e}})$, the values $\overline{\mathtt{o}}$ to which $\overline{\mathtt{e}}$ reduce are the initial values of the fields.

Notice that standard methods can be seen as special cases of sessions. In fact, a method declaration can be (informally) encoded as a session with nested `recC`s (one for each parameter) and with one `sendC` returning the method body. Similarly, method calls are special cases of session requests: the passing of arguments is encoded as nested `sendC`s (one for each argument) and the object returned by the method body is retrieved via one `recC`. This encoding will use unary choices that can be rendered with binary choices as explained above.

4 Auxiliary Functions

As in FJ, a class table CT is a mapping from class names to class declarations with domain $\mathscr{D}(CT)$. Then a program is a pair $(CT, \mathtt{e})$ of a class table (containing all the class definitions of the program) and an expression `e` (an expression belonging to the source language representing the program's main entry point). The class `Object` has no fields and its declaration does not appear in CT. As in FJ, from any CT we can read off the subtype relation between classes, as the transitive closure of $\lhd$ clause; moreover this relation is extended in order to relate types built out of union (Figure 6). As usual considering union types modulo the equivalence relation induced by $<:$ we get the commutativity and associativity of $\vee$. Therefore each union type can be written

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{T} <: \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}''}{\mathtt{T} <: \mathtt{T}''} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \in \mathtt{CT}}{\mathtt{C} <: \mathtt{D}}$$

$$\mathtt{T} <: \mathtt{T} \vee \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T} \vee \mathtt{T}' \qquad \frac{\mathtt{T}' <: \mathtt{T} \quad \mathtt{T}'' <: \mathtt{T}}{\mathtt{T}' \vee \mathtt{T}'' <: \mathtt{T}}$$

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \frac{\mathit{fields}(\mathtt{D}) = \overline{\mathtt{T}'\,\mathtt{f}'} \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T\,f}}, \overline{\mathtt{T}'\,\mathtt{f}'}}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T\,f}}}{\mathit{ftype}_w(\mathtt{f}_i, \mathtt{C}) = \mathit{ftype}_r(\mathtt{f}_i, \mathtt{C}) = \mathtt{T}_i}$$

$$\mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1 \vee \mathtt{T}_2) = \begin{cases} \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1) & \text{if } \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1) <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_2), \\ \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_2) & \text{if } \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_2) <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1), \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathit{ftype}_r(\mathtt{f}, \mathtt{T}_1 \vee \mathtt{T}_2) = \mathit{ftype}_r(\mathtt{f}, \mathtt{T}_1) \vee \mathit{ftype}_r(\mathtt{f}, \mathtt{T}_2)$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{T\,t\,s}\{\ \mathtt{e}\ \} \in \overline{\mathtt{S}}}{\mathit{stype}(\mathtt{s}, \mathtt{C}) = \{\mathtt{t}\}} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{s} \notin \overline{\mathtt{S}}}{\mathit{stype}(\mathtt{s}, \mathtt{C}) = \mathit{stype}(\mathtt{s}, \mathtt{D})}$$

$$\mathit{stype}(\mathtt{s}, \mathtt{T}_1 \vee \mathtt{T}_2) = \mathit{stype}(\mathtt{s}, \mathtt{T}_1) \cup \mathit{stype}(\mathtt{s}, \mathtt{T}_2)$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{T\,t\,s}\{\ \mathtt{e}\ \} \in \overline{\mathtt{S}}}{\mathit{rtype}(\mathtt{s}, \mathtt{C}) = \mathtt{T}} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{s} \notin \overline{\mathtt{S}}}{\mathit{rtype}(\mathtt{s}, \mathtt{C}) = \mathit{rtype}(\mathtt{s}, \mathtt{D})}$$

$$\mathit{rtype}(\mathtt{s}, \mathtt{T}_1 \vee \mathtt{T}_2) = \mathit{rtype}(\mathtt{s}, \mathtt{T}_1) \vee \mathit{rtype}(\mathtt{s}, \mathtt{T}_2)$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{T\,t\,s}\{\ \mathtt{e}\ \} \in \overline{\mathtt{S}}}{\mathit{sbody}(\mathtt{s}, \mathtt{C}) = \mathtt{e}} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T\,f}};\ \overline{\mathtt{S}}\ \} \qquad \mathtt{s} \notin \overline{\mathtt{S}}}{\mathit{sbody}(\mathtt{s}, \mathtt{C}) = \mathit{sbody}(\mathtt{s}, \mathtt{D})}$$

Fig. 6. Subtyping and Lookup Functions

as $\mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$ for $n \geq 1$: we say that the classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ *build* the union type $\mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$. A union type $\mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$ is *proper* if $n > 1$.

We assume a fixed *CT* that satisfies some usual sanity conditions as in FJ [30]. Thus, in the following, instead of writing $\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \ldots$ we will simply write $\mathtt{class\ C}\ \ldots$.

We define auxiliary functions (see Figure 6) to lookup fields and sessions from *CT*; these functions are used in the typing rules and in the operational semantics. The *fields* lookup function is as in FJ. As for field type lookup we distinguish between the contexts where the field is used for reading (ftype_r) from those where it is for writing (ftype_w). The *stype* and *rtype* return a set of session types and the return type of a session, respectively, while *sbody* returns the body of a session. As in FJ these functions may have to inspect the class hierarchy in case the required element is not present in the current class.

Notice that the type lookup functions take a type as argument (not simply a class name) because the receiver expression of a field/session access may be of a proper union type. As for field type lookup, when the field is used in read mode, in case of a proper union type, we simply return the union type of the result of ftype_r invoked on the argument types (if both retrievals succeed). On the contrary, when a field is updated, due to the contravariance relation, in case of a proper union type we must

$$\mathtt{e}\langle\mathtt{k}\rangle = \begin{cases} \mathtt{e}_1\langle\mathtt{k}\rangle;\mathtt{e}_2\langle\mathtt{k}\rangle & \text{if } \mathtt{e} = \mathtt{e}_1;\mathtt{e}_2, \\ \mathtt{e}_1\langle\mathtt{k}\rangle.\mathtt{f} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}, \\ \mathtt{e}_1\langle\mathtt{k}\rangle.\mathtt{f}:=\mathtt{e}_2\langle\mathtt{k}\rangle & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}:=\mathtt{e}_2, \\ \mathtt{e}_1\langle\mathtt{k}\rangle.\mathtt{s}\{\mathtt{e}_2\} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{s}\{\mathtt{e}_2\}, \\ \mathtt{e}_1\langle\mathtt{k}\rangle\bullet\mathtt{s}\{\mathtt{k}\} & \text{if } \mathtt{e} = \mathtt{e}_1\bullet\mathtt{s}\{\ \}, \\ \mathtt{k}.\mathtt{sendC}(\mathtt{e}_0\langle\mathtt{k}\rangle)\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\langle\mathtt{k}\rangle}\} & \text{if } \mathtt{e} = \mathtt{sendC}(\mathtt{e}_0)\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{k}.\mathtt{recC}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\langle\mathtt{k}\rangle}\} & \text{if } \mathtt{e} = \mathtt{recC}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{k}.\mathtt{sendW}(\mathtt{e})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\langle\mathtt{k}\rangle}\} & \text{if } \mathtt{e} = \mathtt{sendW}(\mathtt{e})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{k}.\mathtt{recW}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\langle\mathtt{k}\rangle}\} & \text{if } \mathtt{e} = \mathtt{recW}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{e} & \text{otherwise.} \end{cases}$$

$$\mathtt{e}\lfloor\mathtt{e}'/\mathtt{cont}\rfloor = \begin{cases} \mathtt{e}_1\lfloor\mathtt{e}'/\mathtt{cont}\rfloor;\mathtt{e}_2\lfloor\mathtt{e}'/\mathtt{cont}\rfloor & \text{if } \mathtt{e} = \mathtt{e}_1;\mathtt{e}_2, \\ \mathtt{e}_1\lfloor\mathtt{e}'/\mathtt{cont}\rfloor.\mathtt{f} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}, \\ \mathtt{e}_1\lfloor\mathtt{e}'/\mathtt{cont}\rfloor.\mathtt{f}:=\mathtt{e}_2\lfloor\mathtt{e}'/\mathtt{cont}\rfloor & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}:=\mathtt{e}_2, \\ \mathtt{e}_1\lfloor\mathtt{e}'/\mathtt{cont}\rfloor.\mathtt{s}\{\mathtt{e}_2\} & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{s}\{\mathtt{e}_2\}, \\ \mathtt{e}_1\lfloor\mathtt{e}'/\mathtt{cont}\rfloor\bullet\mathtt{s}\{\mathtt{k}\} & \text{if } \mathtt{e} = \mathtt{e}_1\bullet\mathtt{s}\{\mathtt{k}\}, \\ \mathtt{k}.\mathtt{sendC}(\mathtt{e}_0)\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\lfloor\mathtt{e}'/\mathtt{cont}\rfloor}\} & \text{if } \mathtt{e} = \mathtt{k}.\mathtt{sendC}(\mathtt{e}_0)\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{k}.\mathtt{recC}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}\lfloor\mathtt{e}'/\mathtt{cont}\rfloor}\} & \text{if } \mathtt{e} = \mathtt{k}.\mathtt{recC}(\mathtt{x})\{\overline{\mathtt{C} \Rightarrow \mathtt{e}}\}, \\ \mathtt{e}' & \text{if } \mathtt{e} = \mathtt{cont}, \\ \mathtt{e} & \text{otherwise.} \end{cases}$$

Fig. 7. Channel Addition and Continuation Replacement

return the intersection of the result of $ftype_w$ on the arguments; however, in the absence of multiple inheritance, the only possible cases are those listed in Figure 6, thus we can avoid introducing intersection types.

As for the *stype* lookup function, it returns a set of session types; in case it is invoked with a class name as argument, it will return a singleton. The interesting case is when it is invoked with a proper union type: it will return the union of the sets corresponding to the argument types, so that we have all the session types of the classes that build the union type (see how it is used in the typing system, Figures 11 and 13). The *rtype* lookup function behaves in a covariant way since the resulting object cannot be used in writing mode. We notice that *sbody* is only invoked with a class name as type argument, since we invoke sessions on objects only, and all objects have a class name as type.

It is easy to verify that all lookup functions applied to equivalent union types return either equivalent union types or the same sets of session types, whenever they are defined.

5 Operational Semantics

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

$$h \ ::= \ [\,] \ \mid \ \mathtt{o} \mapsto (\mathtt{C}, \overline{\mathtt{f} : \mathtt{o}}) \ \mid \ \mathtt{k} \mapsto \overline{\mathtt{o}} \ \mid \ h :: h$$

where :: denotes heap concatenation.

During evaluation, any expression $\texttt{new C}(\overline{\texttt{o}})$ will be replaced by a new object identifier o. The heap will then maps the object identifier o to the pair $(\texttt{C}, \overline{\texttt{f} : \texttt{o}})$ of its class name C and the list of its fields with corresponding objects $\overline{\texttt{o}}$; this mapping is denoted by $\texttt{o} \mapsto (\texttt{C}, \overline{\texttt{f} : \texttt{o}})$.

The form $h[\texttt{o} \mapsto h(\texttt{o})[\texttt{f} \mapsto \texttt{o}']]$ denotes the update of the field f of the object o with the object $\texttt{o}'$.

Channel names are mapped to queues of objects: $\texttt{k} \mapsto \overline{\texttt{o}}$. The heap produced by $h[\texttt{k} \mapsto \overline{\texttt{o}}]$ maps the channel k to the queue $\overline{\texttt{o}}$. With some abuse of notation we write $\texttt{o} :: \overline{\texttt{o}}$ and $\overline{\texttt{o}} :: \texttt{o}$ to denote the queue whose first and last element is o, respectively.

Heap membership for object identifiers and channels is checked using standard set notation, by identifying h with its domain, we can also write $\texttt{o} \in h$, and $\texttt{k} \in h$.

The values that can result from normal termination are parallel threads of fully evaluated objects.

In the reduction rules we make use of the special *channel addition* operation $\wr \ldots \wr$, and of the *continuation replacement* operation $\lfloor \ldots / \texttt{cont} \rfloor$ (their formal definitions are in Figure 7, where $\{\overline{\texttt{C} \Rightarrow \texttt{e}}\}$ is short for $\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\}$). We denote by $\texttt{e}\wr\texttt{k}\wr$ the source expression e in which all occurrences of receive, send, and delegation expressions which *are not* within the co-body of a session request are extended, so that they explicitly mention the channel k they will use (remember that channel names are not written by the programmer). Also, we denote by $\texttt{e}\lfloor \texttt{e}'/\texttt{cont} \rfloor$ the expression e in which all occurrences of cont, that *are not* within the co-body of a session request or within the body of a send/receive loop, are replaced by $\texttt{e}'$, thus preserving the correct nested structure of while expressions. For example $\texttt{recC}(\texttt{x})\{\texttt{C}_1 \Rightarrow \texttt{x} \,[]\, \texttt{C}_2 \Rightarrow \texttt{cont}\}\wr\texttt{k}\wr\lfloor \texttt{e}'/\texttt{cont} \rfloor = \texttt{k.recC}(\texttt{x})\{\texttt{C}_1 \Rightarrow \texttt{x} \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}'\}$.

The reduction is a relation between pairs of threads and heaps:

$$P, h \longrightarrow P', h'$$

Reduction rules use evaluation contexts (based on runtime syntax) that capture the notion of the "next subexpression to be reduced":

$$\begin{array}{c} \mathscr{E} ::= [-] \mid \mathscr{E}; \texttt{e} \mid \mathscr{E}.\texttt{f} \mid \mathscr{E}.\texttt{f} := \texttt{e} \mid \texttt{o.f} := \mathscr{E} \mid \mathscr{E}.\texttt{s}\{\texttt{e}\} \mid \\ \mathscr{E} \bullet \texttt{s}\{\texttt{k}\} \mid \texttt{k.sendC}(\mathscr{E})\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\} \end{array}$$

The explicit mention of the evaluation context is needed in rule SESSREQ-R (Figure 8), in which a new thread is generated in parallel with the evaluation context.

Reduction rules are in Figure 8. Rule PAR-R models the execution of parallel threads. In this rule parallel composition is considered modulo structural equivalence. As usual, we define structural equivalence rules asserting that parallel composition is associative and commutative:

$$P \,||\, P_1 \equiv P_1 \,||\, P \quad P \,||\, (P_1 \,||\, P_2) \equiv (P \,||\, P_1) \,||\, P_2 \quad P \equiv P' \;\Rightarrow\; P \,||\, P_1 \equiv P' \,||\, P_1$$

The successive four rules define the execution of standard object-oriented constructions.

Rule SESSREQ-R models the connection between the co-body e of a session request $\texttt{o.s}\{\texttt{e}\}$ and the body $\texttt{e}'$ of the session s, in the class of the object o. This connection is

$$\textsc{Par-R}\ \frac{\mathtt{e},h \longrightarrow P,h'}{\mathtt{e} \mid\mid P_1,h \longrightarrow P \mid\mid P_1,h'} \qquad \textsc{Seq-R}\ \mathscr{E}[\mathtt{o};\mathtt{e}],h \longrightarrow \mathscr{E}[\mathtt{e}],h \qquad \textsc{Fld-R}\ \frac{h(\mathtt{o}) = (\mathtt{C},\overline{\mathtt{f}:\mathtt{o}})}{\mathscr{E}[\mathtt{o}.\mathtt{f}_i],h \longrightarrow \mathscr{E}[\mathtt{o}_i],h}$$

$$\textsc{NewC-R}\ \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\,\mathtt{f}} \qquad \mathtt{o} \notin h}{\mathscr{E}[\mathtt{new}\,\mathtt{C}(\overline{\mathtt{o}})],h \longrightarrow \mathscr{E}[\mathtt{o}],h::[\mathtt{o} \mapsto (\mathtt{C},\overline{\mathtt{f}:\mathtt{o}})]} \qquad \textsc{FldAss-R}\ \mathscr{E}[\mathtt{o}.\mathtt{f} := \mathtt{o}'],h \longrightarrow \mathscr{E}[\mathtt{o}'],h[\mathtt{o} \mapsto h(\mathtt{o})[\mathtt{f} \mapsto \mathtt{o}']]$$

$$\textsc{SessReq-R}\ \frac{h(\mathtt{o}) = (\mathtt{C},\_) \qquad \mathit{sbody}(\mathtt{s},\mathtt{C}) = \mathtt{e}' \qquad \mathtt{k},\tilde{\mathtt{k}} \notin h}{\mathscr{E}[\mathtt{o}.\mathtt{s}\,\{\mathtt{e}\}],h \longrightarrow \mathscr{E}[\mathtt{e}\wr\mathtt{k}\wr] \mid\mid [\mathtt{o}/\mathtt{this}]\mathtt{e}'\wr\tilde{\mathtt{k}}\wr,h[\mathtt{k},\tilde{\mathtt{k}} \mapsto ()]} \qquad \textsc{SessDel-R}\ \frac{h(\mathtt{o}) = (\mathtt{C},\_) \qquad \mathit{sbody}(\mathtt{s},\mathtt{C}) = \mathtt{e}}{\mathscr{E}[\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}],h \longrightarrow \mathscr{E}[[\mathtt{o}/\mathtt{this}]\mathtt{e}\wr\mathtt{k}\wr],h}$$

$$\textsc{SendCase-R}\ \frac{h(\tilde{\mathtt{k}}) = \overline{\mathtt{o}} \qquad h(\mathtt{o}) = (\mathtt{C},\_) \qquad \mathtt{C} \Downarrow \{\mathtt{C}_1,\mathtt{C}_2\} = \mathtt{C}_i}{\mathscr{E}[\mathtt{k}.\mathtt{sendC}(\mathtt{o})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}],h \longrightarrow \mathscr{E}[\mathtt{e}_i],h[\tilde{\mathtt{k}} \mapsto \overline{\mathtt{o}} :: \mathtt{o}]}$$

$$\textsc{ReceiveCase-R}\ \frac{h(\mathtt{k}) = \mathtt{o} :: \overline{\mathtt{o}} \qquad h(\mathtt{o}) = (\mathtt{C},\_) \qquad \mathtt{C} \Downarrow \{\mathtt{C}_1,\mathtt{C}_2\} = \mathtt{C}_i}{\mathscr{E}[\mathtt{k}.\mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}],h \longrightarrow \mathscr{E}[[\mathtt{o}/\mathtt{x}]\mathtt{e}_i],h[\mathtt{k} \mapsto \overline{\mathtt{o}}]}$$

$$\textsc{SendWhile-R}\ \mathscr{E}[\mathtt{k}.\mathtt{sendW}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}],h \longrightarrow \mathscr{E}[\mathtt{k}.\mathtt{sendC}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}],h$$
$$\text{where } \mathtt{e}'_i = \lfloor \mathtt{e}_i/\mathtt{cont} \rfloor \mathtt{k}.\mathtt{sendW}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}$$

$$\textsc{ReceiveWhile-R}\ \mathscr{E}[\mathtt{k}.\mathtt{recW}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}],h \longrightarrow \mathscr{E}[\mathtt{k}.\mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}],h$$
$$\text{where } \mathtt{e}'_i = \lfloor \mathtt{e}_i/\mathtt{cont} \rfloor \mathtt{k}.\mathtt{recW}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}$$

Fig. 8. Reduction Rules

established through a pair of fresh channels $\mathtt{k},\tilde{\mathtt{k}}$. For this purpose the expression $\mathtt{o}.\mathtt{s}\,\{\mathtt{e}\}$ reduces, in the same context, to its own co-body $\mathtt{e}\wr\mathtt{k}\wr$ and in parallel, outside the context, it spawns the body $[\mathtt{o}/\mathtt{this}]\mathtt{e}'\wr\tilde{\mathtt{k}}\wr$ of the called session. The explicit substitution of $\mathtt{k}$ in $\mathtt{e}$ and of $\tilde{\mathtt{k}}$ in $\mathtt{e}'$ ensures that the communication is on the fresh dual channels $\mathtt{k}$ and $\tilde{\mathtt{k}}$. Thus, an object can serve *any number* of session requests. For example,

$$\begin{array}{c}\mathtt{o}.\mathtt{s}\,\{\mathtt{sendC}(5)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}\};\mathtt{new}\ \mathtt{C}(\,) \longrightarrow \\ \mathtt{k}.\mathtt{sendC}(5)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1\wr\mathtt{k}\wr \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\wr\mathtt{k}\wr\};\mathtt{new}\ \mathtt{C}(\,) \mid\mid \\ \tilde{\mathtt{k}}.\mathtt{recC}(\mathtt{x})\{\mathtt{C}'_1 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_1\wr\tilde{\mathtt{k}}\wr \,[]\, \mathtt{C}'_2 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_2\wr\tilde{\mathtt{k}}\wr\}\end{array}$$

if $\mathtt{recC}(\mathtt{x})\{\mathtt{C}'_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{e}'_2\}$ is the body of session $\mathtt{s}$ in the class of the object $\mathtt{o}$. Notice that there is no ambiguity in this rule, since

$$\begin{array}{c}(\mathtt{k}.\mathtt{sendC}(5)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1\wr\mathtt{k}\wr \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\wr\mathtt{k}\wr\} \mid\mid \\ \tilde{\mathtt{k}}.\mathtt{recC}(\mathtt{x})\{\mathtt{C}'_1 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_1\wr\tilde{\mathtt{k}}\wr \,[]\, \mathtt{C}'_2 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_2\wr\tilde{\mathtt{k}}\wr\});\mathtt{new}\ \mathtt{C}(\,)\end{array}$$

is not a thread according to the syntax of $\mathscr{F}\mathsf{SAM}^{\vee}$.

Rule SessDel-R replaces the session delegation $\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}$ by $[\mathtt{o}/\mathtt{this}]\mathtt{e}\wr\mathtt{k}\wr$, where $\mathtt{e}$ is the body of the session $\mathtt{s}$, in the class of the object $\mathtt{o}$. This allows a part of the communication to be delegated 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 $\mathtt{k}$. For example
$\mathtt{o} \bullet \mathtt{s}\{\mathtt{k}\} \longrightarrow \mathtt{k}.\mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}_1 \langle \mathtt{k} \rangle \,[]\, \mathtt{C}_2 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}_2 \langle \mathtt{k} \rangle\}$
if $\mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$ is the body of session $\mathtt{s}$ in the class of the object $\mathtt{o}$.

The communication rule for `sendC`, SENDCASE-R, puts the object $\mathtt{o}$, *i.e.* the result of evaluating the expression $\mathtt{e}$, in the queue associated to the dual channel $\tilde{\mathtt{k}}$ of the communication channel $\mathtt{k}$. The computation then proceeds with the expression $\mathtt{e}_i$, if $\mathtt{C}_1 \neq \mathtt{C}_2$ and $\mathtt{C}_i$ is the smallest class in $\{\mathtt{C}_1, \mathtt{C}_2\}$ to which the object $\mathtt{o}$ belongs. Otherwise, if $\mathtt{C}_1 = \mathtt{C}_2$ and $\mathtt{o}$ belongs both to $\mathtt{C}_1$ and to $\mathtt{C}_2$, then the computation proceeds with $\mathtt{e}_1$[2]. This is given by the condition $h(\mathtt{o}) = (\mathtt{C}, \_)$ and by the following definition of $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$, using the subtyping relation (Figure 6):

$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \begin{cases} \mathtt{C}_i & \text{if } \mathtt{C} <: \mathtt{C}_i \text{ and } \mathtt{C} <: \mathtt{C}_j \text{ with } i \neq j \text{ imply } \mathtt{C}_j \not<: \mathtt{C}_i, \\ \mathtt{C}_1 & \text{if } \mathtt{C} <: \mathtt{C}_1 = \mathtt{C}_2, \\ \bot & \text{otherwise.} \end{cases}$$

Dually the receive communication rule takes an object $\mathtt{o}$ from the queue associated to channel $\mathtt{k}$ and returns the expression $[\mathtt{o}/\mathtt{x}]\mathtt{e}_i$, if $h(\mathtt{o}) = (\mathtt{C}, \_)$ and $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$.

In rules SENDCASE-R and RECEIVECASE-R it is understood that the transition cannot fire if $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \bot$. However we will see that $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\}$ is always defined in well-typed expressions.

Rules SENDWHILE-R and RECEIVEWHILE-R simply realize the repetition using the case communication expressions. Note that $\mathtt{sendW}(\mathscr{E})\{\mathtt{e}_1 \Rightarrow \mathtt{C}_1 \,[]\, \mathtt{e}_2 \Rightarrow \mathtt{C}_2\}$ is not an evaluation context, since we do not want to reduce the expression which controls the loop before the application of rule SENDWHILE-R, in which the `sendW` expression is unfolded.

Only communication and delegation expressions containing explicit channels can be reduced. So, for example, `sendC(o){e}` and $\mathtt{o} \bullet \mathtt{s}\{\}$ are stuck; however, as we will see in Subsection 6.1, the latter cannot be typed and the former is not an initial expression (type soundness is only guaranteed for initial expressions).

6 Typing

Session types, ranged over by $\mathtt{t}$, describe the communications that take place during a session. The syntax of session types is in Figure 9, where we use † as a symbol that stands for either ! or ?. 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 $!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ and $?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ express the sending and the receiving of an object, respectively: depending on the class of this object the communication will proceed with one of the $\mathtt{t}_i$. In $\mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ the *session type variable* α can occur inside $\mathtt{t}_i$ with the usual meaning of representing the whole session type. We consider recursive session types modulo fold/unfold: i.e., $\mu\alpha.\mathtt{t} =$

[2] In this particular case, there is no other motivation for selecting the smallest index but to avoid introducing non-deterministic choices. From this point of view, alternative solutions could be just as sound: for instance, the selection of the greatest index or linguistic restrictions on the expressions $\mathtt{e}_i$, e.g., the condition $\mathtt{e}_1 = \mathtt{e}_2$ whenever $\mathtt{C}_1 = \mathtt{C}_2$.

$$\begin{array}{lll} \dagger ::= ! \mid ? & & \text{direction} \\ \mathtt{t} ::= \varepsilon \mid \alpha \mid \circledcirc \mid \dagger\{\mathtt{C} \Rightarrow \mathtt{t} \,[]\, \mathtt{C} \Rightarrow \mathtt{t}\} \mid \mu\alpha.\dagger\{\mathtt{C} \Rightarrow \mathtt{t} \,[]\, \mathtt{C} \Rightarrow \mathtt{t}\} \mid \mathtt{t}.\mathtt{t} & & \text{session type} \end{array}$$

Fig. 9. Syntax of Session Types

$[\mu\alpha.\mathtt{t}/\alpha]\mathtt{t}$. So we equate $\mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ to $\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ when α does not occur in $\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$.

The type $\circledcirc$ is used only as session type for 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 SENDW-T and RECEIVEW-T in Figure 11).

We say that a session type is *closed* if it does not contain occurrences of free session type variables and of $\circledcirc$. Therefore, each closed session type has one of the following shapes:

- ε;
- $\mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ or $\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$;

or a concatenation of the session types above. For simplicity we will use in definitions unfolded recursive types whenever possible.

6.1 Typing of Channel Free Expressions

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 call these expressions *channel free expressions*. The term environments therefore will contain also type assignments to object identifiers. This permits a simpler formulation of the runtime typing rules, as we will see in next subsection.

The typing judgement has the shape

$$\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}$$

where Γ is a term environment, which maps `this`, `cont`, variables and objects to types, and `t` represents the session type of the (implicit) active channel.

In order to allow (possible) multiple occurrences of a variable or `cont` with different types inside an expression, we define the following "update" operation on Γ(`z` ranges over `this`, `cont`, term variables, and object identifiers):

$$\Gamma(\mathtt{z} : \mathtt{T})(\mathtt{z}') = \begin{cases} \mathtt{T} & \text{if } \mathtt{z}' = \mathtt{z} \\ \Gamma(\mathtt{z}') & \text{otherwise.} \end{cases}$$

Thus, the operation $\Gamma(\mathtt{z} : \mathtt{T})$ has the effect of adding $\mathtt{z} : \mathtt{T}$ to Γ, but after deleting a declaration of `z` from Γ (if there is one). This will avoid checking well-formedness of term environments and does not require an explicit weakening rule to add an assumption on the `cont` variable (when typing nested while communication expressions).

To assure a safe communication between two threads we must require their session types to be *dual*, i.e., that each send will correspond to a receive and vice versa. The duality is then the symmetric relation generated by the rules of Figure 10, in which we consider folded recursive types, otherwise the definition would not be well-founded.

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

$$\frac{\mathtt{C}_1 \vee \mathtt{C}_2 <: \mathtt{C}'_1 \vee \mathtt{C}'_2 \quad \mathtt{C}_i \Downarrow \{\mathtt{C}'_1, \mathtt{C}'_2\} = \mathtt{C}'_j \Rightarrow \mathtt{t}_i \bowtie \mathtt{t}'_j \quad \mathtt{C}'_l \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_k \Rightarrow \mathtt{t}_k \bowtie \mathtt{t}'_l}{\mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\} \bowtie \mu\alpha.?\{\mathtt{C}'_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{t}'_2\}}$$

Fig. 10. Duality Relation

The exchanged values must also be of one of the classes expected by the receiver. All possible choices on the basis of the class of the exchanged value must continue with session types which are dual of each other. For this reason we have to perform checks on the type of the exchanged values in both directions:

- for any sent value of type $\mathtt{C}_i$ such that $\mathtt{C}_i \Downarrow \{\mathtt{C}'_1, \mathtt{C}'_2\} = \mathtt{C}'_j$ for some $1 \leq j \leq 2$ we require $\mathtt{t}_i \bowtie \mathtt{t}'_j$;
- for any received value of type $\mathtt{C}'_l$ such that $\mathtt{C}'_l \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_k$ for some $1 \leq k \leq 2$ we require $\mathtt{t}_k \bowtie \mathtt{t}'_l$.

For instance, let us consider the session types $!\{\mathtt{Shape} \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{String} \Rightarrow \mathtt{t}_2\}$ and $?\{\mathtt{Triangle} \Rightarrow \mathtt{t}_3 \,[]\, \mathtt{Object} \Rightarrow \mathtt{t}_4\}$ where $\mathtt{Triangle} <: \mathtt{Shape}$. At run time a `Triangle` can be sent as a `Shape`, thus the types $\mathtt{t}_1$ and $\mathtt{t}_3$ have to be dual. Notice that, thanks to the absence of generics we can be more flexible w.r.t. [6]: the types used in the choices (actually their union) of the send can be subtypes of the ones expected (in the dual receive).

Typing rules for channel free expressions are in Figure 11. For the sake of simplicity in rule NEWC-T we require that the initialisation of an object does not involve communications. Notice that in rule SEQ-T we use session type concatenation to represent that first the communications in $\mathtt{e}_1$ and then those in $\mathtt{e}_2$ are performed.

The rule for session request SESSREQ-T relies on the duality relation (Figure 10) to assure that all the bodies of the session `s` in the classes which build the union type `T` and the co-body $\mathtt{e}'$ of the request will communicate properly. In typing session delegation (rule SESSDEL-T) we take into account that the whole expression will be replaced by the session body defined in the class of the expression to which the session is delegated (cf. the reduction rule SESSDEL-R, Figure 8). Notice that the condition $stype(\mathtt{s}, \mathtt{T}) = \{\mathtt{t}'\}$ does not imply `T` be one class, but only that all definitions of `s` in the classes which build `T` have the same session types. If a session has session type ε, then it is meaningless to use it in a delegation, while it is sensible to use it in a request. For this reason we require $\mathtt{t}' \neq \varepsilon$ in rule SESSDEL-T.

Rules SENDC-T and RECEIVEC-T require all possible alternative expressions to have the same type `T`, but they can implement different communication sequences $\mathtt{t}_i$. Rule SENDC-T prescribes that the class type of `e` is the union type of the classes used in the choice. Without union types the typing rule for the same construct in [18] was much more demanding and less clear. The typing rules for the while communication expressions are similar, but they also discharge the assumption on `cont` and replace the occurrences of $\circledcirc$ in session types by a fresh variable α which will be bound by μ. In rule SENDW-T typing `e` with session type ε prevents `e` from containing occurrences

$$\textsc{Axiom-T}\quad \Gamma \vdash \mathtt{z} : \Gamma(\mathtt{z}) \fatsemi \varepsilon \qquad \mathtt{z} \neq \mathtt{cont}$$

$$\textsc{Cont-T}\quad \Gamma \vdash \mathtt{cont} : \Gamma(\mathtt{cont}) \fatsemi \circledcirc$$

$$\textsc{Sub-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t} \qquad \mathtt{T} <: \mathtt{T}'}{\Gamma \vdash \mathtt{e} : \mathtt{T}' \fatsemi \mathtt{t}}$$

$$\textsc{NewC-T}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\,\mathtt{f}} \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \fatsemi \varepsilon}{\Gamma \vdash \mathtt{new}\,\mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C} \fatsemi \varepsilon}$$

$$\textsc{Fld-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}}{\Gamma \vdash \mathtt{e.f} : \mathit{ftype}_r(\mathtt{f},\mathtt{T}) \fatsemi \mathtt{t}}$$

$$\textsc{Seq-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t} \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}' \fatsemi \mathtt{t}'}{\Gamma \vdash \mathtt{e};\mathtt{e}' : \mathtt{T}' \fatsemi \mathtt{t}.\mathtt{t}'}$$

$$\textsc{FldAss-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t} \qquad \Gamma \vdash \mathtt{e}' : \mathit{ftype}_w(\mathtt{f},\mathtt{T}) \fatsemi \mathtt{t}'}{\Gamma \vdash \mathtt{e.f} := \mathtt{e}' : \mathit{ftype}_w(\mathtt{f},\mathtt{T}) \fatsemi \mathtt{t}.\mathtt{t}'}$$

$$\textsc{SessReq-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t} \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}' \fatsemi \mathtt{t}' \qquad \mathtt{t}' \bowtie \mathtt{t}'' \ \forall \mathtt{t}'' \in \mathit{stype}(\mathtt{s},\mathtt{T})}{\Gamma \vdash \mathtt{e.s}\{\mathtt{e}'\} : \mathtt{T}' \fatsemi \mathtt{t}}$$

$$\textsc{SessDel-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t} \qquad \mathit{stype}(\mathtt{s},\mathtt{T}) = \{\mathtt{t}'\} \qquad \mathtt{t}' \neq \varepsilon \qquad \mathit{rtype}(\mathtt{s},\mathtt{T}) = \mathtt{T}'}{\Gamma \vdash \mathtt{e} \bullet \mathtt{s}\{\} : \mathtt{T}' \fatsemi \mathtt{t}.\mathtt{t}'}$$

$$\textsc{SendC-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \mathtt{t} \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i}{\Gamma \vdash \mathtt{sendC}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \mathtt{t}.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}}$$

$$\textsc{ReceiveC-T}\quad \frac{\Gamma(\mathtt{x} : \mathtt{C}_i) \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i}{\Gamma \vdash \mathtt{recC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}}$$

$$\textsc{SendW-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \varepsilon \qquad \Gamma(\mathtt{cont} : \mathtt{T}) \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash \mathtt{sendW}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}}$$

$$\textsc{ReceiveW-T}\quad \frac{\Gamma(\mathtt{cont} : \mathtt{T})(\mathtt{x} : \mathtt{C}_i) \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash \mathtt{recW}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}}$$

Fig. 11. Typing Rules for Channel Free Expressions

of communications and `cont`[3]. Notice that requiring that both branch expressions in a choice operation have the same union type T does not imply that we require them to be of the same class: in fact, T can be a proper union type. For instance, we may have that $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \mathtt{t}_1$ and $\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \mathtt{t}_2$; by subsumption (rule Sub-T) we also have that $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \mathtt{t}_1$ and $\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \mathtt{t}_2$. Then, $\mathtt{T} = \mathtt{T}_1 \vee \mathtt{T}_2$.

Figure 12 defines well-formed class tables. Rule Sess-WF type checks the session bodies with respect to the current class C taking as term environment the association between `this` and C. Notice that $\circledcirc$ has no dual type, so sessions whose bodies would be typed with types containing $\circledcirc$ would be useless. This justifies the condition that t must be closed in rule Sess-WF.

A last remark is that, since no typing rule generates free session type variables, then all session types in typing judgements are closed unless they contain occurrences of $\circledcirc$.

[3] Note that this typing allows e to contain session requests, since the execution of these requests will use different channels to communicate.

$$\textsc{Sess-WF}\ \frac{\{\texttt{this} : \texttt{C}\} \vdash \texttt{e} : \texttt{T} \fatsemi \texttt{t} \qquad \texttt{t is closed}}{\texttt{T t s}\{\ \texttt{e}\ \}\ \texttt{ok in C}} \qquad \textsc{Class-WF}\ \frac{\texttt{D ok} \qquad \overline{\texttt{S}}\ \texttt{ok in C}}{\texttt{class C} \lhd \texttt{D}\ \{\ \overline{\texttt{T f}};\ \overline{\texttt{S}}\ \}\ \texttt{ok}}$$

Fig. 12. Well-formed Class Tables

The rules, presented in this section, define how the type system checks that the declarative part of the program and the main part are well-typed, also with respect to session types used in declarations of sessions. However, when considering well-typed executable programs, we require that they are closed with respect to term variables and that all communication expressions occur inside session co-bodies, that is, that they are typed in the empty type environment with an empty session type. Namely, an *initial* expression e is such that $\emptyset \vdash \texttt{e} : \texttt{T} \fatsemi \varepsilon$ for some T. It is easy to verify that the set of initial expressions is the set of closed and well-typed user expression. For example, let us consider the stuck expressions sendC(o){e} and o • s{}:

- sendC(o){e} is well-typed but its session type is not empty,
- o • s{} is not well-typed.

We conclude this subsection by comparing $\mathscr{F}\mathsf{SAM}^{\vee}$ with FJ$\vee$, an extension of FJ with union types, proposed by Igarashi and Nagira in [29]. They define union types as in the present paper: the essential difference is that they have traditional methods instead of sessions.

The method signatures are of the shape $\overline{\texttt{T}} \to \texttt{T}$, where both the parameter types $\overline{\texttt{T}}$ and the return type T are union types. The method type lookup function applied to a method name m and to a union type T gives a set of method signatures, *i.e.* all the signatures which m has in the classes which build T. This is similar to our *stype* function, which returns a set of session types.

The rule of method call checks that the types of the parameters agree with all the signatures found by the method type lookup function for the union type of the object. Also our rule SessReq-T requires the session type of the co-body be dual to all the session types returned by the *stype* function.

It is easy to check that the encoding of methods by sessions sketched at the end of Section 3 extends without changes to methods with union types.

6.2 Typing of Runtime Expressions

During evaluation of well-typed programs, channel names are made explicit in send and receive expressions, as well as in session delegation. Thus, in order to show how well-typedness is preserved under evaluation, we need to define new typing rules for runtime expressions. Furthermore, in typing runtime expressions, we must take into account the session types of more than one channel: runtime expressions contain explicit channel names (used for communication) thus session types must be associated with channel names in an appropriate way. Then judgements have the form

$$\Gamma \vdash_{\mathtt{r}} \texttt{e} : \texttt{T} \fatsemi \Sigma$$

where Σ denotes a *session environment* which maps channels to session types.

$$\textsc{Axiom-RT}\quad \Gamma \vdash_{\mathtt{r}} \mathtt{z} : \Gamma(\mathtt{z}) \fatsemi \emptyset \qquad \mathtt{z} \neq \mathtt{cont}$$

$$\textsc{Fld-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma}{\Gamma \vdash_{\mathtt{r}} \mathtt{e.f} : \mathit{ftype}_r(\mathtt{f},\mathtt{T}) \fatsemi \Sigma}$$

$$\textsc{Sub-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma \qquad \mathtt{T} <: \mathtt{T}'}{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma}$$

$$\textsc{NewC-RT}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\,\mathtt{f}} \qquad \Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}_i \fatsemi \emptyset}{\Gamma \vdash_{\mathtt{r}} \mathtt{new\,C}(\overline{\mathtt{e}}) : \mathtt{C} \fatsemi \emptyset}$$

$$\textsc{Cont-RT}\quad \Gamma(\mathtt{cont} : \mathtt{T}) \vdash_{\mathtt{r}} \mathtt{cont} : \mathtt{T} \fatsemi \{\mathtt{k} : \circledcirc\}$$

$$\textsc{Seq-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma \qquad \Gamma \vdash_{\mathtt{r}} \mathtt{e}' : \mathtt{T}' \fatsemi \Sigma'}{\Gamma \vdash_{\mathtt{r}} \mathtt{e};\mathtt{e}' : \mathtt{T}' \fatsemi \Sigma.\Sigma'}$$

$$\textsc{FldAss-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma \qquad \Gamma \vdash_{\mathtt{r}} \mathtt{e}' : \mathit{ftype}_w(\mathtt{f},\mathtt{T}) \fatsemi \Sigma'}{\Gamma \vdash_{\mathtt{r}} \mathtt{e.f} := \mathtt{e}' : \mathit{ftype}_w(\mathtt{f},\mathtt{T}) \fatsemi \Sigma.\Sigma'}$$

$$\textsc{SessReq-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}' \fatsemi \mathtt{t}' \qquad \mathtt{t}' \bowtie \mathtt{t}'' \; \forall \mathtt{t}'' \in \mathit{stype}(\mathtt{s},\mathtt{T})}{\Gamma \vdash_{\mathtt{r}} \mathtt{e.s}\{\mathtt{e}'\} : \mathtt{T}' \fatsemi \Sigma}$$

$$\textsc{SessDel-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma \qquad \mathit{stype}(\mathtt{s},\mathtt{T}) = \{\mathtt{t}\} \qquad \mathtt{t} \neq \varepsilon \qquad \mathit{rtype}(\mathtt{s},\mathtt{T}) = \mathtt{T}'}{\Gamma \vdash_{\mathtt{r}} \mathtt{e} \bullet \mathtt{s}\{\mathtt{k}\} : \mathtt{T}' \fatsemi \Sigma.\{\mathtt{k} : \mathtt{t}\}}$$

$$\textsc{SendC-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \Sigma \qquad \Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\}}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \Sigma.\{\mathtt{k} : !\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}}$$

$$\textsc{ReceiveC-RT}\quad \frac{\Gamma(\mathtt{x} : \mathtt{C}_i) \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\}}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.recC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}}$$

$$\textsc{SendW-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \emptyset \qquad \Gamma(\mathtt{cont} : \mathtt{T}) \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\} \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}}$$

$$\textsc{ReceiveW}\quad \frac{\Gamma(\mathtt{cont} : \mathtt{T})(\mathtt{x} : \mathtt{C}_i) \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\} \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.recW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}}$$

Fig. 13. Typing Rules for Runtime Expressions

A session environment maps only a finite set of channels to session types different from ε, and all the remaining to ε. We can then represent one session environment with an infinite number of finite sets which give all the meaningful associations and some of the others. For example $\{\mathtt{k} : \mathtt{t}\}$ and $\{\mathtt{k} : \mathtt{t}, \mathtt{k}' : \varepsilon\}$ represent the same environment. This choice avoids an explicit weakening rule for session environments. Figure 13 gives the typing rules for runtime expressions, which differ from those for channel free expressions for having session environments instead of a unique session type. For this reason we extend the *concatenation* of session types to *session environments* as follows:

$$\Sigma.\Sigma'(\mathtt{k}) = \Sigma(\mathtt{k}).\Sigma'(\mathtt{k})$$

Notice that in rule SessReq-RT we are making use of the judgement $\Gamma \vdash \mathtt{e}' : \mathtt{T} \fatsemi \mathtt{t}'$, where the expression $\mathtt{e}'$ does not contain channels, but it can contain object identifiers. This justifies our choice of considering channel free expressions instead of user expressions in the typing rules of previous subsection. Notice also that the session environments of the branches in the communication expressions only contain the current channel as subject, since these expressions will never be reduced before the selection

has been done. In rule SENDW-RT we assume $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \,;\, \emptyset$, since the evaluation of `e` cannot start before the `sendW` expression has been unfolded to a `sendC`.

The typing rules for runtime expressions differ from the ones for user expressions only in assigning the session type to explicit channels, not in the union type.

6.3 Type Soundness

Our type system enjoys subject reduction and assures progress ($\longrightarrow^*$ is the reflexive and transitive closure or $\longrightarrow$):
If $\emptyset \vdash \mathtt{e}_0 : \mathtt{T}_0 \,;\, \varepsilon$ and $\mathtt{e}_0, [\,] \longrightarrow^* P, h$, where $P \equiv \mathtt{e}_1 \mid\mid \ldots \mid\mid \mathtt{e}_n$, then:

- for each $\mathtt{e}_i$ we get $\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}_i \,;\, \Sigma_i$ for some $\Gamma, \mathtt{T}_i, \Sigma_i$ $(1 \leq i \leq n)$, and there is j $(1 \leq j \leq n)$ such that $\mathtt{T}_j = \mathtt{T}_0$, and;
- either $P, h \longrightarrow P', h'$ for some P', h', or for all i $(1 \leq i \leq n)$ $\mathtt{e}_i$ is an object identifier.

The runtime errors which our type system prevents are:

1. the selection of a field and the request of a session which do not belong to the class of the current object;
2. the creation of a pair of dual channels whose communication sequences do not perfectly match.

Proofs, more examples and discussions can be found in the extended version of this paper, available at http://www.di.unito.it/∼dezani/papers/bcdgvfull.pdf.

7 Conclusion

In the present paper we showed, through the language $\mathsf{SAM}^{\vee}$, how the addition of union types to an object oriented language with session types enhances flexibility.

The amalgamation of communication centred and object oriented programming, as it has been developed in [18, 6] and in the present paper, can be extended in various directions. In particular we plan to integrate this approach with multi-party session communication [9] and with safe failure recovery [4].

Moreover, we want to study the extension of union and intersection to session types, following the intuition given by union/intersection of contracts in [12].

Lastly it would be interesting to integrate session primitives with name constraints as introduced in [5] in order to allow specification of Quality of Service requirements.

Acknowledgements. We thank the referees for their helpful comments. The final version of the paper improved due to their suggestions.

References

1. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Intersection and Union Types: Syntax and Semantics. Information and Computation 119, 202–230 (1995)
2. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (to appear, 2008)
3. Bonelli, E., Compagnoni, A., Gunter, E.: Correspondence Assertions for Process Synchronization in Concurrent Communications. Journal of Functional Programming 15(2), 219–248 (2005)

4. Boreale, M., Bruni, R., Caires, L., Nicola, R.D., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
5. Buscemi, M., Montanari, U.: CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007)
6. Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E.: Amalgamating Sessions and Methods in Object Oriented Languages with Generics (submitted, 2007)
7. Carbone, M., Honda, K., Yoshida, N.: A Calculus of Global Interaction Based on Session Types. In: Fernández, M., Kirchner, C. (eds.) SecReT 2006. ENTCS, vol. 171(3), pp. 127–151. Elsevier, Amsterdam (2007)
8. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
9. Carbone, M., Honda, K., Yoshida, N.: Multiparty Asynchronous Session Types. In: Necula, G.C., Wadler, P. (eds.) POPL 2008, pp. 273–284. ACM Press, New York (2008)
10. Castagna, G., De Nicola, R., Varacca, D.: Semantic Subtyping for the π-calculus. In: Panangaden, P. (ed.) LICS 2005, pp. 92–101. IEEE Computer Society Press, Los Alamitos (2005)
11. Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: Barahona, P., Felty, A.P. (eds.) PPDP 2005, pp. 198–199. ACM Press, New York (2005)
12. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. In: Necula, G.C., Wadler, P. (eds.) POPL 2008, pp. 261–272. ACM Press, New York (2008)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
14. Dezani-Ciancaglini, M., de' Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
15. Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E., Yoshida, N.: Bounded Session Types for Object-Oriented Languages. In: de Boer, F., Bonsangue, M. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 207–245. Springer, Heidelberg (2007)
16. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
17. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A Distributed Object Oriented Language with Session Types. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 299–318. Springer, Heidelberg (2005)
18. Drossopoulou, S., Dezani-Ciancaglini, M., Coppo, M.: Amalgamating the Session Types and the Object Oriented Programming Paradigms. In: MPOOL 2007 (2007), http://homepages.fh-regensburg.de/mpool/mpool07/programme.html
19. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language Support for Fast and Reliable Message-based Communication in Singularity OS. In: Zwaenepoel, W. (ed.) EuroSys 2006. ACM SIGOPS, pp. 177–190. ACM Press, New York (2006)
20. Fournet, C., Laneve, C., Maranget, L., Rémy, D.: Inheritance in the Join Calculus. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 397–408. Springer, Heidelberg (2000)
21. Gapeyev, V., Pierce, B.C.: Regular Object Types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 151–175. Springer, Heidelberg (2003)

22. Garralda, P., Compagnoni, A., Dezani-Ciancaglini, M.: BASS: Boxed Ambients with Safe Sessions. In: Maher, M. (ed.) PPDP 2006, pp. 61–72. ACM Press, New York (2006)
23. Gay, S.: Bounded Polymorphism in Session Types. In: MSCS (to appear, 2008)
24. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)
25. Gay, S., Vasconcelos, V.T., Ravara, A.: Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow (2003)
26. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
28. Honda, K., Yoshida, N., Carbone, M.: Web Services, Mobile Processes and Types. EATCS Bulletin 91, 160–188 (2007)
29. Igarashi, A., Nagira, H.: Union Types for Object Oriented Programming. Journal of Object Technology 6(2), 31–52 (2007)
30. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a Minimal Core Calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (2001)
31. Sparkes, S.: Conversation with Steve Ross-Talbot. ACM Queue 4(2), 14–23 (2006)
32. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
33. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the Behavior of Objects and Components using Session Types. In: Brogi, A., Jacquet, J.-M. (eds.) FOCLASA 2002. ENTCS, vol. 68(3), pp. 439–456. Elsevier, Amsterdam (2002)
34. Vasconcelos, V.T., Gay, S., Ravara, A.: Typechecking a Multithreaded Functional Language with Session Types. Theorical Computer Science 368(1-2), 64–87 (2006)
35. Web Services Choreography Working Group. Web Services Choreography Description Language (2002), http://www.w3.org/2002/ws/chor/