Math. Struct. in Comp. Science (2013), *vol.* 23, *pp.* 1163–1219. © Cambridge University Press 2013
doi:10.1017/S0960129512000886 First published online 9 May 2013

Deriving session and union types for objects†

LORENZO BETTINI‡, SARA CAPECCHI‡,
MARIANGIOLA DEZANI-CIANCAGLINI‡,
ELENA GIACHINO§ and BETTI VENNERI¶

‡*Dipartimento di Informatica, Università di Torino, corso Svizzera 185, 10131 Torino, Italy*
Email: bettini; capecchi; dezani@di.unito.it
§*Focus Research Team, Università di Bologna/INRIA, mura Anteo Zamboni 7, 40127 Bologna, Italy*
Email: giachino@cs.unibo.it
¶*Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, viale Morgagni 65, 50134 Firenze, Italy*
Email: betti.venneri@unifi.it

Received 20 December 2010; revised 24 September 2011

Guaranteeing that the parties of a network application respect a given protocol is a crucial issue. *Session types* offer a method for abstracting and validating structured communication sequences (sessions). *Object-oriented programming* is an established paradigm for large scale applications. *Union types*, which behave as the least common supertypes of a set of classes, allow the implementation of unrelated classes with similar interfaces without additional programming. We have previously developed an integration of the features above into a class-based core language for building network applications, and this successfully amalgamated sessions and methods so that data can be exchanged flexibly according to communication protocols (session types).
The first aim of the work reported in this paper is to provide a full proof of the type safety property for that core language by renewing syntax, typing and semantics. In this way, static typechecking guarantees that after a session has started, computation cannot get stuck on a communication deadlock.
The second aim is to define a constraint-based type system that reconstructs the appropriate session types of session declarations instead of assuming that session types are explicitly given by the programmer. Such an algorithm can save programming work, and automatically presents an abstract view of the communications of the sessions.

1. Introduction

When developing network applications it is crucial to have a linguistic mechanism to write safe communication protocols. The current mainstream programming languages, such as Java, still leave the programmer with most of the responsibility for guaranteeing that the communication will evolve as agreed by all the involved agents. The standard type systems can only provide a means of declaring the types of the exchanged data, but

† This work was partially supported by MIUR Projects DISCO – Distribution, Interaction, Specification, Composition for Object Systems, and IPODS–Interacting Processes in Open-ended Distributed Systems, and by EU Collaborative project n. 257414 ASCENS – Autonomic Service-Component Ensembles.

they cannot guarantee that a communication protocol is respected so that a client–server application avoids getting stuck because of an error in the communication sequence.

Session types (Honda 1993; Honda *et al.* 1998) were introduced as a mechanism for abstracting structured communication sequences (*sessions*) and for validating communication protocols. In this approach, communication channels are given types representing the values sent or received. For instance, the type `?int.!bool` expresses the fact that an integer will be received and then a boolean value will be sent (as usual in process calculi, `?` and `!` are used here for input and output, respectively). In order to respect a communication protocol, a session must involve channels of dual session types, thus guaranteeing that after a session has started, the values sent and received will be of the appropriate type and the communication will not get stuck. For instance, if one channel has the type `?int.!bool`, the other must have the dual type `!int.?bool`. Since the specification of a session is a type, the conformance test of programs with respect to specifications becomes type checking.

Furthermore, it is important in network applications for us to be able to rely on the type-safe flexibility of exchanged data. This means we need a mechanism for abstracting over the actual types that are communicated over a network protocol. This is even more crucial when execution paths are chosen according to the run-time type of the exchanged objects. For this reason, it seems natural for us to try to merge communication mechanisms into the popular *object-oriented* programming paradigm. However, mainstream object-oriented class-based programming languages do not provide linguistic constructs that deal directly with communications and protocols, and writing network communication programs typically involves relying on specific libraries. Instead, we would like to write class definitions that include communication primitives in a natural way. With this in mind, an amalgamation of communication centred and object-oriented programming was first proposed in Drossopoulou *et al.* (2007), where methods are unified with sessions and choices are based on the classes of exchanged objects.

In an object-oriented class-based context, reusability is based both on subclassing and on the substitutability implied by subtyping, which coincides with subclassing (or interface implementation) in Java-like languages. Thus, this form of reuse must be designed from the start by choosing the right base classes or interfaces, since, although two classes may share some features (methods and fields), if they do not belong to the same hierarchy, their reuse will require a refactoring of existing code. A solution to deal with these problems is provided by *union types*, which represent the set unions of objects of several types: a union type behaves as the least common supertype of a set of objects, without requiring the writing of a specific base class or interface. With union types, in an object-oriented programming scenario, developing independent classes with similar interfaces requires no additional programming (Igarashi and Nagira 2007).

For these reasons, union types seem to be very useful when communications involve data exchange in the shape of objects as class instances: we can express communications between parties that manipulate heterogeneous objects by just sending and receiving objects that belong to subclasses of one of the classes in the union type. For instance, consider a communication between a bank and a client, where the bank can answer `yes` or `no` to a client request, according to the account balance. If `yes` and `no` 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`, that is, `OK` $\vee$ `NoMoney`. Without union types, typing `answer` would require a superclass of both `OK` and `NoMoney` to be already defined, and, as well as the need for manual programming, and possible code refactoring, this superclass might also include unwanted objects. This does not happen with a union type (least common supertype). In this way, the flexibility of object-oriented depth-subtyping is enhanced by greatly improving the expressiveness of choices based on the classes of sent/received objects.

In this paper we merge union types in the amalgamation of sessions and methods in order to enhance the network communications of class-based programs relying on session types. In Bettini *et al.* (2008a), we introduced $\mathscr{F}\mathsf{SAM}^{\vee}$ (*F*eatherweight *S*essions *A*malgamated with *M*ethods plus union types), which formalises the use of union types for session-centred communications in a core object-oriented calculus. $\mathscr{F}\mathsf{SAM}^{\vee}$, like the language of Drossopoulou *et al.* (2007), is agnostic with respect to other aspects of the language, such as whether the language is distributed or concurrent, and the features used for synchronisation. In $\mathscr{F}\mathsf{SAM}^{\vee}$, sessions are defined in a class (which may also have fields). Sessions and methods are 'amalgamated' so that invocation is made on an object and the execution takes place immediately and concurrently with the requesting thread ($\mathscr{F}\mathsf{SAM}^{\vee}$ is indeed multi-threaded and the communication is asynchronous). Thus, it keeps the method-like invocation mechanism while involving two threads, which is typical for session-based communication mechanisms. Just like the dynamic binding of object-oriented method invocation, the body is determined by the class of the receiving object (in this way we avoid the usual branch/select primitives (Honda *et al.* 1998)), and any number of communications may be interleaved with computation. We believe that this amalgamated session model reflects our intuition of services in a natural way. Furthermore, it can neatly encode 'standard' methods.

This paper extends the work in Bettini *et al.* (2008a) in many ways. First, the syntax (and consequently the typing and semantics) are slightly modified. Second, we present a full formalisation of $\mathscr{F}\mathsf{SAM}^{\vee}$, together with proofs of the type soundness property (from a technical point of view, the amalgamation of union types and session-centred communications poses specific problems in formulating reduction and typing rules to ensure that communications are safe but flexible). Finally, we also introduce a type inference system for the session types of the sessions in classes. In particular, while the type system derives session types for expressions assuming that all session declarations are decorated with explicit session types (and expressions can have many types due to the presence of subsumption), the inference algorithm gives an expression its minimal type and calculates the constraints that must be satisfied in order to reconstruct the related session type (which will be proved unique).

With the type inference system, the programmer is no longer responsible for declaring the session types. Therefore, this inference has a pragmatic motivation since, due to their 'behavioural' nature, session types are often quite long to write out when the communication protocol is not a simple one (especially when recursive types are involved). Thus, having a type inference system for session types can save some programming work, and automatically presents an abstract view of the communications of the sessions. However, in

an implementation of our approach, the inference algorithm does not necessarily prevent the programmer from writing session types. For instance, the programmer might decide to write the session types explicitly, and then use the inference system as a tool for verifying the written protocols. Alternatively, the inference system might insert the inferred types in the text of the program so that the programmer can have an abstract view of the protocol and verify that the protocol is as it was intended. Finally, a mixed approach could be employed in which the programmer writes the explicit session types for at least one side of the protocol, and then lets the type inference system generate the session type for the other part. Summarising, in an implementation, the session type inference system does not necessarily require that all the session type declarations are removed from a program, but is meant as a tool to help the programmer while designing and implementing communication protocols. The aim of our presentation of the type inference system in this paper is just to study its theory and properties; we do not consider how it might be employed in practice by a language designer.

1.1. *Structure of the paper*

The calculus $\mathscr{F}\mathsf{SAM}^{\vee}$ is described together with its operational semantics in Sections 2, 3 and 4. Section 5 presents the type system, whose properties are then proved in Section 6. Sections 7 and 8 are devoted to the type inference system. Finally, in Sections 9 and 10, we discuss related work and draw some conclusions.

2. The calculus $\mathscr{F}\mathsf{SAM}^{\vee}$

In this section we present the syntax of $\mathscr{F}\mathsf{SAM}^{\vee}$ (Figure 1), which is a minimal concurrent and imperative core calculus based on *Featherweight Java* (Igarashi *et al.* 2001), which we will refer to as FJ for short. $\mathscr{F}\mathsf{SAM}^{\vee}$ supports the basic object-oriented features and session requests, session delegation, branching sending/receiving and loops.

Figure 1 shows *run-time expressions* against a grey background – these are produced during the reduction process and do not occur in *user expressions*. We use the standard convention of writing $\overline{\xi}$ to denote a sequence of elements $\xi_1, \ldots, \xi_n$.

Types, ranged over by $\mathtt{T}$, are defined as in Igarashi and Nagira (2007): they are built out of class names by the union operator (denoted by $\vee$).

Programs are defined from a collection of classes. The metavariables $\mathtt{C}$ and $\mathtt{D}$, possibly with subscripts, range over class names. Each class has a suite of *fields* of the form $\overline{\mathtt{T}\,\mathtt{f}}$, where $\mathtt{f}$ represents the field name and $\mathtt{T}$ its type, and a suite of *session declarations* $\overline{\mathtt{S}}$. As in FJ, the fields declared by a class are added to those of the superclass and the resulting sequence of fields is assumed to contain no duplicate names. We declare sessions in the same way as we declare methods in Java classes, with the new remarkable feature that their bodies can include communication operations. Since, as we shall see at the end of this section, sessions can encode methods, for simplicity, we will omit standard methods in our classes. Session declarations are of the form $\mathtt{T}\,\mathtt{t}\,\mathtt{s}\{\ \mathtt{e}\ \}$, where $\mathtt{s}$ is the session name, $\mathtt{e}$ the session body, $\mathtt{T}$ the return type and $\mathtt{t}$ is the session type, which describes the communication protocol in the way standard method types describe the protocols

(type)	T	::=	C | T ∨ T
(class)	L	::=	class C ◁ D { $\overline{\mathtt{T}\,\mathtt{f}}$; $\overline{\mathtt{S}}$ }
(session)	S	::=	T t s { e }
(expression)	e	::=	x | this | cont$^{\mathtt{T}}$ | o | e; e | e.f := e | e.f | new C($\overline{\mathtt{e}}$)
		|	e.s{e} | e • s{ k }
		|	k. sendC(e){C_1 ⇒ e [] C_2 ⇒ e}
		|	k. recvC(x){C_1 ⇒ e [] C_2 ⇒ e}
		|	k. sendW(e){C_1 ⇒ e [] C_2 ⇒ e}
		|	k. recvW(x){C_1 ⇒ e [] C_2 ⇒ e}
(parallel threads)	P	::=	e | P ‖ P
(session type)	t	::=	ε | t.t | α | †{C_1 ⇒ t [] C_2 ⇒ t} | μα.†{C_1 ⇒ t [] C_2 ⇒ t} | ◎

Fig. 1. Syntax, where syntax occurring only at run time appears shaded.

for method-call interactions. For conciseness, we use the symbol ◁ to represent class extension, as in Igarashi *et al.* (2001). The class `Object` is implicitly defined in every program; it has no fields and no sessions. A class definition always includes the superclass (even when it is `Object`).

Expressions include variables, which include both standard term variables `x` and the special variables `this` and cont$^{\mathtt{T}}$. The variable `this` is considered implicitly bound in any session declaration. `sendW` and `recvW` are the only binders for the free occurrences of cont$^{\mathtt{T}}$ inside their bodies, where cont$^{\mathtt{T}}$ represents the continuation by recursive computation. The intuition is that `sendW` and `recvW` expressions will, when necessary, be unfolded during evaluation by replacing the free occurrences of cont$^{\mathtt{T}}$ in their bodies with the whole expressions. Note that for any type `T`, a special variable cont$^{\mathtt{T}}$ is provided: it is decorated by the type `T` in order to represent the recursive computation of an expression of type `T`.

As usual, an expression is *closed* if it does not contain any free variables.

Object identifiers, denoted by `o`, are generated at run time when creating objects (by `new` expressions).

The expression e.s{e′} is a *session request*, where e′ is called the *co-body* of the request: by the operational rules, e is evaluated to an object o, and the session body of s in o's class is executed concurrently with e′ by introducing a new pair of fresh channels, k and its dual $\tilde{\mathtt{k}}$ (one for each communication direction), to perform communications between the session body and the co-body. This means that the evaluation of session requests has a crucial effect on the syntax: it generates parallel threads and introduces communication channels (which are implicit in the source language).

The expression e • s{k} represents the *session delegation* in the sense that the execution of the session s is delegated to the object resulting from the evaluation of e. This means that in order for the current object to continue the communication with its partner safely, it needs to borrow a capability from another object. In this sense, we are using the term 'delegation' in the same way as it is usually used in the session types literature. However, our notion of delegation diverges slightly from the standard one. In our case, the current object asks another object to provide a functionality in its place, without releasing control

of the session: the session channel is not moved around and the current thread executes the code of the delegated object. Technically, this is very close to standard method invocation. By contrast, the standard form of session delegation requires that a private channel is sent to another thread, which will take care of the session communication on the received channel, while the current thread is excluded from the session. It is not easy to express this higher order use of channels in our setting, where channels are only created at run time. The channel `k` corresponds to the subject of communication expressions inside the session body. See Section 4 for further details (in particular, for an explanation of the reduction rule SessDel-R).

The body of a *communication expression* is a pair of alternatives $\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$, whose choice depends on the class of the object that is sent or received. The expression $\mathtt{sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$ evaluates `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 (if $\mathtt{C}_1 = \mathtt{C}_2$, the whole expression evaluates to $\mathtt{e}_1$). The counterpart of `sendC` is the expression $\mathtt{recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$, where the choice is based on the class of the object received. The expression $\mathtt{sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$ (where W means While) is similar to $\mathtt{sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$, except that it allows for the possibility of an enclosed $\mathtt{cont}^{\mathtt{T}}$, which continues the execution at the nearest enclosing `sendW`. The expression $\mathtt{recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$ has the obvious meaning.

Note that in our setting, recursion on objects (via `this`) is not suitable for expressing cycles within single sessions since it would give rise to different sessions.

Parallel threads, ranged over by P, are run-time expressions or parallel compositions of run-time expressions, where a run-time expression is either a user expression (that is, an expression in Figure 1 without shaded syntax) or an expression containing channels and/or object identifiers.

In *session types*, we use $\dagger$ as a symbol that stands for either ! or ?. We use ε to denote the *empty* communication, and the *concatenation* $\mathtt{t}_1.\mathtt{t}_2$ denotes the communications in $\mathtt{t}_1$ followed by those in $\mathtt{t}_2$. Concatenation of session types is used for typing sequential composition of expressions – see rule Seq-T in Figure 9. The session type ε is the neutral element of concatenation, so $\varepsilon.\mathtt{t} = \mathtt{t} = \mathtt{t}.\varepsilon$ for all `t`.

The types $!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ and $?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$ are used to express the sending and receiving of an object, respectively: depending on the class $\mathtt{C}_i$ of this object, the communication will proceed with the one of type $\mathtt{t}_i$. 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: that is, $\mu\alpha.\mathtt{t} = [\mu\alpha.\mathtt{t}/\alpha]\mathtt{t}$. 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

$$\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.$$

```
1  sessiontype Sum_ST = μα.?{Int ⇒ α, Char ⇒ ?{Paper ⇒ ε, Video ⇒ ε} }
2
3  sessiontype Print_ST = ?{Paper ⇒ ε, Video ⇒ ε}
4
5  class Calculator{
6     Int value;
7     Video∨Paper Sum_ST sum{
8        recvW(x){
9          Int ⇒ value:=value+x; cont^{Video∨Paper};
10         []
11         Char ⇒ this•print;
12       }
13    }
14    Video∨Paper Print_ST print{
15       recvC(y){
16         Paper ⇒ ...; new Paper(); // print the result on paper
17         []
18         Video ⇒ ...; new Video(); // print the result on the screen
19       }
20    }
21 }
```

Fig. 2. The class Calculator.

The type ◎ is only used as session type for $\texttt{cont}^{\texttt{T}}$: it plays the role of a place holder that will be replaced by a type variable when the while expression is completed – see rules SendW-T and ReceiveW-T in Figure 9.

The following example shows the expressiveness of $\mathscr{F}\mathsf{SAM}^{\vee}$ in a typical collaboration pattern – see Bettini *et al.* (2008a) for further motivating examples of our language constructs.

Example 2.1. The interaction we show is between a calculator and a client (Figures 2 and 3). The `Client` sends integer values, which are summed by the `Calculator`. This interaction then iterates until the `Client` sends a character to signal that the addends are complete. The `Client` then sends the `Calculator` an object indicating the display-mode to be used for the result (`Paper` or `Video`). Finally, the `Calculator` displays the result.

The session types `Sum_ST` and `Print_ST` (Figure 2) describe the protocol from the point of view of the `Calculator`. The recursive type

$$\mu\alpha.\ ?\{\texttt{Int} \Rightarrow \alpha, \texttt{Char} \Rightarrow ?\{\texttt{Paper} \Rightarrow \varepsilon,\ \texttt{Video} \Rightarrow \varepsilon\}\}$$

describes the `Calculator` getting the addends from the `Client`. The first branch represents the case in which the `Calculator` receives an integer from the `Client`, in which case the iteration continues, that is, the `Calculator` receives the next input. The second branch represents the case in which the `Client` sends to the `Calculator` a character to signal that the addends are complete, in which case a further object is expected specifying the

```
1  sessiontype Request_ST = μα.!{Int ⇒ α, Char ⇒ !{Paper ⇒ ε, Video ⇒ ε} }
2
3  class Client{
4     Int∨Char msg;
5     Paper∨Video mode;
6     Calculator c;
7     ...
8
9     c.sum{
10       sendW(msg){
11         Int ⇒ update(msg); cont^{Video∨Paper}; // update the content of msg
12         []
13         Char ⇒ sendC(mode){
14             Paper ⇒ ...;
15             []
16             Video ⇒ ...;
17         }
18       }
19    }
20    ...
21 }
```

Fig. 3. The class Client.

mode for displaying the result. In the two branches $\{\texttt{Paper} \Rightarrow \varepsilon, \texttt{Video} \Rightarrow \varepsilon\}$, there is no further communications (the session type is ε) since the only action is the printing (or displaying) of the result.

Figure 2 shows the implementation of the class `Calculator`. It has the field `value`, which is used to store the sum of the addends. The class supports two sessions, called `sum` and `print`. The session `sum` has session type `Sum_ST` and return type `Video∨Paper`: this union type represents the possible results of the session, that is, the possible display modes of the result of the sum. Note that the return type of the session represents the type of the session body (exactly as return types in standard object-oriented languages). Indeed, it is used for dealing with session delegation when the body of the session is incorporated in the current execution. In this case, we know that the execution of the session body of `sum` will reduce to a value of type `Video∨Paper`. On the other hand, the session type is needed to deal with session invocation, and to check the correctness of the communication. In this case, we see that an invocation of the session `sum` must perform a dual communication with respect to its session type `Sum_ST`. The session `print` has session type `Print_ST` and return type `Video∨Paper` again. In the body of `sum`, the `Calculator` receives an object (line 8) which can be:

(i) of type `Int`, in which case it will be summed to `value` and then the recursion will continue ($\texttt{cont}^{\texttt{Video}\vee\texttt{Paper}}$); or

(ii) of type `Char`, in which case the remaining part of the session is delegated to the `Calculator` itself, which goes on with session `print` (line 10).

The body of the session `print` begins by receiving an object indicating the display mode (line 15): according to the class of the received object the field `value` will be printed on paper or displayed on the video.

The session type `Request_ST` (Figure 3) describes the protocol from the point of view of the `Client`. The recursive type

$$\mu\alpha.\ !\{\texttt{Int} \Rightarrow \alpha,\ \texttt{Char} \Rightarrow !\{\texttt{Paper} \Rightarrow \varepsilon,\ \texttt{Video} \Rightarrow \varepsilon\}\}$$

describes the `Client` sending the addends to the `Calculator`. The first branch represents the case in which the `Client` sends an integer to the `Calculator`, in which case the iteration goes on updating and sending the next message. The second branch represents the case in which the `Client` sends the `Calculator` a character to signal that the addends are complete, in which case a further object is sent to indicate the mode the result must be displayed in. Figure 3 shows the implementation of the class `Client`. It has a field of type `Calculator`, and two fields `msg` and `mode`, which are used to store the values sent to the `Calculator`. The types of the `msg` and `mode` fields, $\texttt{Int} \vee \texttt{Char}$ and $\texttt{Paper} \vee \texttt{Video}$, respectively, describe the possible classes of the sent values. Line 9 provides an example of session invocation: the `Client` invokes on the `Calculator` `c` the session `sum`. The body of the session invocation (lines 10-16) has session type `Request_ST`. This will be executed in parallel with the body of the session `sum` in the class `Calculator`. Note that the class `Client` is not fully specified: we just show the code of the session invocation, which must appear somewhere inside a session declaration of the `Client`.

Clearly, this example would not be typeable if we replaced `Char` by another type, say `Bool`, in the code of the `Client`.

We adopt some simplifications in $\mathscr{F}\mathsf{SAM}^{\vee}$. First, unary choices and n-ary choices are omitted since they can be simply encoded using binary choices (as shown in Bettini *et al.* (2008a)). Moreover, types used for selecting branches in a choice are required to be class names, rather than union types. This is not a limitation since, for instance,

$$\{\mathtt{C}_1 \vee \mathtt{C}_2 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}'\}$$

can be encoded as

$$\{\mathtt{C}_1 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e} \,[]\, \mathtt{C}_3 \Rightarrow \mathtt{e}'\}.$$

Unlike FJ, we do not have cast and overriding in $\mathscr{F}\mathsf{SAM}^{\vee}$ since they are orthogonal to the issues we are concerned with. We do not have explicit constructors either, so in the object instantiation expression $\texttt{new}\ \ \mathtt{C}(\overline{\mathtt{e}})$, the values $\overline{\mathtt{o}}$ to which $\overline{\mathtt{e}}$ reduce are the initial values of the fields. Furthermore, we omit standard methods since they are viewed as special cases of sessions. In fact, a method declaration can be encoded as a session with nested `recvC`s (one for each parameter) and with one `sendC` returning the method body. Similarly, method calls are just 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 using one `recvC`.

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

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

Fig. 4. Subtyping.

3. Auxiliary functions

As in FJ, a class table CT is a mapping from class names to class declarations with domain $\mathsf{dom}(CT)$. A program is then 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` does not appear in CT. We assume a fixed CT that satisfies some usual sanity conditions as in FJ (Igarashi *et al.* 2001). Thus, in the following, instead of writing $CT(\mathtt{C}) = \mathtt{class}\ \ldots$ we will simply write `class C` ....

From any CT we can read off the subtype relation between classes as the transitive closure of $\lhd$ clause. Moreover, subtyping is extended to union types as in Figure 4. As usual, by considering union types modulo the equivalence relation induced by $<:$, we get the commutativity and associativity of $\vee$. Therefore, each union type can be written as $\mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$ for $n \geqslant 1$, and 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 define auxiliary lookup functions (see Figure 5) for looking up fields and sessions in CT: these functions are used in the typing rules and the operational semantics. As in FJ, these functions have to inspect the class hierarchy when the required element is not present in the current class. The difference is that all these functions, apart from function *sbody*, take a type as argument (not just a class name) because the receiver expression of a field/session access may be of a proper union type.

For *field-type* lookup, we distinguish between the contexts where the field is used for reading ($ftype_r$) from those where it is used for writing ($ftype_w$). 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 other hand, when a field is updated, due to the contravariance relation, in the case of a proper union type, we must return the intersection of the results of $ftype_w$ on the arguments. However, in the absence of multiple inheritance, either the results are related by subtyping, that is, the intersection is exactly one of the classes, or they are not related at all, that is, the intersection is empty, so we can avoid introducing intersection types. For example, if the objects of class $\mathtt{C}_i$ have a field `f` of class $\mathtt{D}_i$ for $i \in \{1, 2\}$ with $\mathtt{D}_1 <: \mathtt{D}_2$, then $ftype_r(\mathtt{C}_1 \vee \mathtt{C}_2) = \mathtt{D}_1 \vee \mathtt{D}_2$ and $ftype_w(\mathtt{C}_1 \vee \mathtt{C}_2) = \mathtt{D}_1$. Otherwise, if $\mathtt{D}_1$ and $\mathtt{D}_2$ are unrelated, we again get $ftype_r(\mathtt{C}_1 \vee \mathtt{C}_2) = \mathtt{D}_1 \vee \mathtt{D}_2$, but $ftype_w(\mathtt{C}_1 \vee \mathtt{C}_2)$ is undefined.

The functions *stype* and *rtype* return a set of session types and the return type of a session, respectively, and *sbody* returns the body of a session. The *stype* function returns a singleton when it is invoked with a class name as argument. But the interesting case

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

$$\frac{fields(\texttt{C}) = \overline{\texttt{T}\,\texttt{f}}}{ftype_w(\texttt{f}_i, \texttt{C}) = ftype_r(\texttt{f}_i, \texttt{C}) = \texttt{T}_i}$$

$$ftype_r(\texttt{f}, \texttt{T}_1 \vee \texttt{T}_2) = ftype_r(\texttt{f}, \texttt{T}_1) \vee ftype_r(\texttt{f}, \texttt{T}_2)$$

$$\frac{ftype_w(\texttt{f}, \texttt{T}_i) <: ftype_w(\texttt{f}, \texttt{T}_j) \quad i \neq j \quad i, j \in \{1, 2\}}{ftype_w(\texttt{f}, \texttt{T}_1 \vee \texttt{T}_2) = ftype_w(\texttt{f}, \texttt{T}_i)}$$

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

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

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

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

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

Fig. 5. Lookup functions.

is when it is invoked with a proper union type, when it will return the union of the sets corresponding to the types of the classes that build the union type so that we have all the session types (see Figures 9 and 12 for how it is used in the type system). The *rtype* function behaves covariantly since the resulting object cannot be used in writing mode. Note that *sbody* is only invoked with a class name as type argument since we invoke sessions on objects only, and an object has a class name as its 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.

4. Operational semantics

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

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

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 map the object identifier o to the pair $(\texttt{C}, \overline{\texttt{f} = \texttt{o}})$, which consists

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

Fig. 6. Channel addition.

of its class name $\mathtt{C}$ and a list of its fields with corresponding objects $\overline{\mathtt{o}}$: this mapping is denoted by $\mathtt{o}\mapsto(\mathtt{C},\overline{\mathtt{f}=\mathtt{o}})$.

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

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

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

The queues of dual channels are used to exchange messages. A message receive on channel $\mathtt{k}$ takes the top object in the queue associated with $\mathtt{k}$, while a message send will add the object to the queue associated with $\tilde{\mathtt{k}}$. As usual, $.\tilde{}.$ is an involution, so $\tilde{\tilde{\mathtt{k}}}=\mathtt{k}$.

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

In the reduction rules, we make use of the special *channel addition* operation $\wr\ldots\int$ – see Figure 6 for a formal definition, where $\{\overline{\mathtt{C}\Rightarrow\mathtt{e}}\}$ is short for $\{\mathtt{C}_1\Rightarrow\mathtt{e}_1\,[\!]\,\mathtt{C}_2\Rightarrow\mathtt{e}_2\}$. We use $\mathtt{e}\wr\mathtt{k}\int$ to denote the source expression $\mathtt{e}$ in which all occurrences of communication (receive, send) and delegation expressions that *are not* within the co-body of a session request have been extended so that they explicitly mention the channel $\mathtt{k}$ they will use (remember that channel names are not written by the programmer).

We also use $\mathtt{e}\lfloor\mathtt{e}'/\mathtt{cont}\rfloor$ to denote the expression $\mathtt{e}$ in which all expressions $\mathtt{cont}^{\mathtt{T}}$ that are free in $\mathtt{e}$, independently of the type annotations $\mathtt{T}$, are replaced by $\mathtt{e}'$. Thus, this substitution preserves the correct nested structure of while expressions. Note that the type annotation $\mathtt{T}$ of $\mathtt{cont}^{\mathtt{T}}$ plays no role in the evaluation, it is only used to guide the typechecker.

For example,

$$\mathtt{recvC}(\mathtt{x})\{\mathtt{C}_1\Rightarrow\mathtt{x}\,[\!]\,\mathtt{C}_2\Rightarrow\mathtt{cont}^{\mathtt{T}}\}\wr\mathtt{k}\int\lfloor\mathtt{e}'/\mathtt{cont}\rfloor=\mathtt{k}.\mathtt{recvC}(\mathtt{x})\{\mathtt{C}_1\Rightarrow\mathtt{x}\,[\!]\,\mathtt{C}_2\Rightarrow\mathtt{e}'\}.$$

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

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

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

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

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

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

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

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

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

Fig. 7. Reduction rules.

Reduction rules use evaluation contexts (based on the run-time syntax) that capture the notion of the 'next subexpression to be reduced':

$$\mathscr{E} ::= [-] \mid \mathscr{E};\mathtt{e} \mid \mathscr{E}.\mathtt{f} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{o}}, \mathscr{E}, \overline{\mathtt{e}}) \mid \mathscr{E}.\mathtt{f} := \mathtt{e} \mid \mathtt{o}.\mathtt{f} := \mathscr{E} \mid \mathscr{E}.\mathtt{s}\,\{\mathtt{e}\} \mid$$
$$\mathscr{E} \bullet \mathtt{s}\,\{\mathtt{k}\} \mid \mathtt{k}.\mathtt{sendC}(\mathscr{E})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}\ .$$

The reduction rules are given in Figure 7, where any reducible expression is expressed as a composition of an evaluation context and a redex expression. The explicit inclusion of the evaluation context is needed in rule SessReq-R, in which a new thread is generated in parallel with the evaluation context. It is easy to verify that the set of redexes is defined by

$$\mathtt{o};\mathtt{e} \mid \mathtt{o}.\mathtt{f} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{o}}) \mid \mathtt{o}.\mathtt{f} := \mathtt{o} \mid \mathtt{o}.\mathtt{s}\,\{\mathtt{e}\} \mid \mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}$$
$$\mathtt{k}.\mathtt{sendC}(\mathtt{o})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} \mid \mathtt{k}.\mathtt{recvC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$$
$$\mathtt{k}.\mathtt{sendW}(\mathtt{e})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} \mid \mathtt{k}.\mathtt{recvW}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}.$$

We call redexes of the form $\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}$ *delegation redexes*, and those having one of the last four forms are called *communication redexes*.

An arbitrary expression is equal to at most one evaluation context filled with one redex, and if it reduces, then there is exactly one reduction rule that applies. So the evaluation strategy is deterministic.

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:

$$P \parallel P_1 \equiv P_1 \parallel P \quad P \parallel (P_1 \parallel P_2) \equiv (P \parallel P_1) \parallel P_2 \quad P \equiv P' \Rightarrow P \parallel P_1 \equiv P' \parallel P_1 .$$

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

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

if $\mathtt{recvC(x)}\{\mathtt{C}'_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{e}'_2\}$ is the body of session $\mathtt{s}$ in the class of the object $\mathtt{o}$.

Rule SessDel-R replaces the session delegation $\mathtt{o} \bullet \mathtt{s}\{\mathtt{k}\}$ by $[\mathtt{o}/\mathtt{this}]\mathtt{e} \wr \mathtt{k} \int$, where $\mathtt{e}$ is the body of the session $\mathtt{s}$, in the class of the object $\mathtt{o}$. This allows the current session to be enriched by the capabilities provided by the session $\mathtt{s}$ of the object $\mathtt{o}$. The current thread executes the body $\mathtt{e}$ in which the current session channel $\mathtt{k}$ is used as the subject for the communication, so the delegation remains transparent for the thread using the dual channel $\tilde{\mathtt{k}}$. When the delegated job is over, the communication may continue within the current session, possibly using the value of $[\mathtt{o}/\mathtt{this}]\mathtt{e} \wr \mathtt{k} \int$. Note that since the value produced by the execution of the delegated session body may be used after the delegation is over, we need both the return type and the session type of that body, and this is why we kept them both in the declaration of a session. See the explanation of Example 2.1 and the session declaration syntax in Figure 1. For instance,

$$\mathtt{o} \bullet \mathtt{s}\{\mathtt{k}\} \longrightarrow \mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}_1 \wr \mathtt{k} \int [] \mathtt{C}_2 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}_2 \wr \mathtt{k} \int\}$$

if

$$\mathtt{recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$$

is the body of session $\mathtt{s}$ in the class of the object $\mathtt{o}$.

To sum up, we can say that:

— *session invocation* creates a new channel and spawns the body of the called session;
— *session delegation* gives the active channel to another session whose body is executed in the same thread.

Since channels are implicit, only one session can be executed at a given time and the only possible interleaving of sessions is *nesting*. A session can be started while executing another session, but must complete before resuming the (outer scoped) previous session, so we can have nesting, but not general interleaving. This is the main reason the progress property holds for communications in our calculus (see Theorem 6.2).

The communication rule for `sendC`, SendCase-R, puts the object `o` in the queue associated with the dual channel $\tilde{\mathtt{k}}$ of the communication channel `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 `o` belongs. Otherwise, if $\mathtt{C}_1 = \mathtt{C}_2$ and `o` belongs to $\mathtt{C}_1$, then the computation proceeds with $\mathtt{e}_1$. This is given by the condition $h(\mathtt{o}) = (\mathtt{C}, \_)$ and the following definition of $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$ using the subtyping relation (see Figure 4):

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

Note that the only reason for selecting the smallest index is so that we avoid introducing non-deterministic choices. In a more realistic context, for instance, we could adopt linguistic restrictions on the expressions $\mathtt{e}_i$, for example, the condition $\mathtt{e}_1 = \mathtt{e}_2$ whenever $\mathtt{C}_1 = \mathtt{C}_2$. Dually, the receive communication rule takes an object `o` from the queue associated with the channel `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 the 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.

The rules SendWhile-R and ReceiveWhile-R simply realise the repetition through the case communication expressions, where the `sendW` and `recvW` expressions are unfolded in $\mathtt{e}_1$ and $\mathtt{e}_2$. Observe that $\mathtt{sendW}(\mathscr{E})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$ is not an evaluation context since we do not reduce the expression that controls the loop before the application of SendWhile-R. This means that the application of SendWhile-R and ReceiveWhile-R cannot create any free occurrences of $\mathtt{cont}^{\mathtt{T}}$.

We will write $P, h \longrightarrow P', h'$ to mean that either P is a parallel composition and $P, h \longrightarrow P', h'$ is obtained by rule Par-R (that is, by reducing one of the expressions that are in parallel in P) or P is an expression `e` that reduces to P' by a reduction rule different from Par-R.

We use the standard convention that the multi-step reduction $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$.

Note that communication and delegation expressions are reduced if and only if they contain explicit channels. So, for example, $\mathtt{sendC(o)}\{\ldots\}$ and $\mathtt{o} \bullet \mathtt{s}\{\}$ are stuck. We say that an expression `e` is *channel-complete* if all communication and delegation expressions of `e` without explicit channels occur inside session co-bodies. The shapes of closed and channel-complete expressions are easy to characterise by looking at the syntax of $\mathscr{F}\mathsf{SAM}^{\vee}$ (Figure 1).

Proposition 4.1. A closed and channel-complete expression is either an object identifier or an evaluation context filled with one redex.

By inspecting the rules in Figure 7, it is easy to verify that no reduction can create new free variables or destroy the channel-completeness starting from the empty heap.

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

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

Fig. 8. Duality relation.

Proposition 4.2. If $\mathtt{e}$ is closed and channel-complete and $\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}' \mid\mid P, h$, then $\mathtt{e}'$ is closed and channel-complete.

5. Typing

We consider two type systems, the first is for user expressions with occurrences of object identifiers, which are not directly expressible in the user syntax. We call these expressions *channel-free expressions*. The second system types run-time expressions. The use of channel-free expressions rather than user expressions simplifies the formulation of the run-time typing rules, as we will see in Section 5.2.

We say that a session type is *cont-free* if it does not contain occurrences of free session type variables or ◎. Therefore, each cont-free session type has one of the following forms:

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

5.1. *Typing of channel-free expressions*

In this section we consider channel-free expressions. This means that the term environments will also contain type assignments to object identifiers. The typing judgement has the form

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

where Γ is a term environment, which maps `this`, standard term variables and objects to types $\mathtt{T}$, and $\mathtt{t}$ represents the session type of the (implicit) active channel. Note that closed expressions can contain object identifiers, so term environments having those object identifiers in their domain are required to type them (unlike the usual notion of closed expressions, which are typable from empty environments).

To guarantee a safe communication between two threads, we must require their session types be *dual*, that is, that each send will correspond to a receive and *vice versa*. The duality is then the symmetric relation generated by the rules of Figure 8, in which we consider folded recursive types, since otherwise the definition would not be well founded. 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 that 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 $1 \leqslant j \leqslant 2$, 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 $1 \leqslant k \leqslant 2$, we require $\mathtt{t}_k \bowtie \mathtt{t}'_l$.

For instance, 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. Moreover, both a `Shape`, which is not a subclass of `Triangle`, and a `String` can be seen as `Objects`, so both $\mathtt{t}_1$ and $\mathtt{t}_2$ must be duals of $\mathtt{t}_4$. Notice that, thanks to the absence of generics, we can be more flexible than Capecchi *et al.* (2009): the types used in the choices (actually their union) of a send can be subtypes of the ones expected (in the dual receive).

The typing rules for channel-free expressions are given in Figure 9.

The axiom Cont-T means that $\mathtt{cont}^{\mathtt{T}}$ has type `T` from any Γ since it is explicitly decorated with its type `T`.

Following the standard notion of object instantiation, rule NewC-T requires that the initialisation of an object does not involve any communications.

In rule Seq-T, we use session type concatenation to represent the fact that the communications in `e` are performed first, and then those in $\mathtt{e}'$.

Rule FldAss-T exploits the writing and reading uses of $\mathtt{e}'$.

The rule for session requests, SessReq-T, relies on the duality relation (Figure 8) to ensure that all the bodies of the session `s` in the classes that build the type `T` and the co-body $\mathtt{e}'$ of the request will communicate properly. Since $\circledcirc$ has no dual session type, this rule ensures that there are no free occurrences of $\mathtt{cont}^{\mathtt{T}}$ in session bodies and co-bodies. For this reason, in well-typed expressions, the reduction rules SendWhile-R and ReceiveWhile-R never replace $\mathtt{cont}^{\mathtt{T}}$ in session bodies and co-bodies.

In typing session delegation (rule SessDel-T), we take into account the fact 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 SessDel-R reduction rule – see Figure 7). Note that the condition $\mathit{stype}(\mathtt{s}, \mathtt{T}) = \{\mathtt{t}'\}$ does not imply that `T` is one class, but only that all definitions of `s` in the classes that build `T` have the same session types. Moreover, if a session has session type ε, 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.

In the rules for communication expressions (SendC-T, ReceiveC-T, SendW-T and ReceiveW-T), the alternative branches $\mathtt{e}_i$ are both given type `T`. However, this does not require both of them to have the same type since `T` can be a proper union type. For

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

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

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

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

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

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

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

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

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

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

$$\textsc{SendW-T}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \varepsilon \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i \qquad \mathtt{T} <: \mathtt{T}' \qquad \forall \mathtt{T}' \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2) \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash \mathtt{sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi\ \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}}$$

$$\textsc{ReceiveW-T}\quad \frac{\Gamma, \mathtt{x} : \mathtt{C}_i \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i \qquad \mathtt{T} <: \mathtt{T}' \qquad \forall \mathtt{T}' \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2) \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash \mathtt{recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi\ \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}}$$

Fig. 9. Typing rules for channel-free expressions. The function *tc* is defined in Figure 10.

instance, we may have

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \mathtt{t}_1$$
$$\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \mathtt{t}_2,$$

but by subsumption (rule Sub-T), we also have

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \mathtt{t}_1$$
$$\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \mathtt{t}_2.$$

So $\mathtt{T} = \mathtt{T}_1 \vee \mathtt{T}_2$. Because of the lack union types, the typing rules for these constructs in Drossopoulou *et al.* (2007) were much more demanding and less clear.

The rules SendW-T and ReceiveW-T take into account the fact that the free occurrences of $\mathtt{cont^T}$ in the bodies are used to make recursive calls of the whole expression. This means that the type decorations of all these occurrences must be greater than or equal to the resulting type of the whole expression. This property is checked by the condition

$$tc(\mathtt{e}) = \begin{cases} tc(\mathtt{e}_1) \cup tc(\mathtt{e}_2) & \text{if } \mathtt{e} = \mathtt{e}_1\,;\mathtt{e}_2, \\ & \mathtt{e} = \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2, \\ & \mathtt{e} = \mathtt{e}_1\,.\,\mathtt{s}\,\{\mathtt{e}_2\}, \\ & \mathtt{e} = \mathtt{k}.\mathtt{sendC}(\mathtt{e}_0)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}, \\ & \mathtt{e} = \mathtt{k}.\mathtt{recvC}(\mathtt{x})\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}, \\ tc(\mathtt{e}_1) & \text{if } \mathtt{e} = \mathtt{e}_1.\mathtt{f}, \\ & \mathtt{e} = \mathtt{e}_1 \bullet \mathtt{s}\,\{\mathtt{k}\}, \\ \{\mathtt{T}\} & \text{if } \mathtt{e} = \mathtt{cont}^{\mathtt{T}}, \\ \varnothing & \text{otherwise.} \end{cases}$$

Fig. 10. The function tc.

$$\frac{\{\mathtt{this} : \mathtt{C}\} \vdash \mathtt{e} : \mathtt{T} \,;\, \mathtt{t} \qquad \mathtt{t} \text{ is cont-free}}{\mathtt{T}\,\mathtt{t}\,\mathtt{s}\,\{\,\mathtt{e}\,\}\ \mathtt{ok}\ \mathtt{in}\ \mathtt{C}}\ \textsc{Sess-WF} \qquad \frac{\mathtt{D}\ \mathtt{ok} \qquad \overline{\mathtt{S}}\ \mathtt{ok}\ \mathtt{in}\ \mathtt{C}}{\mathtt{class}\ \mathtt{C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T}\,\mathtt{f}};\ \overline{\mathtt{S}}\ \}\ \mathtt{ok}}\ \textsc{Class-WF}$$

Fig. 11. Well-formed class tables.

$\mathtt{T} <: \mathtt{T}'$ for all $\mathtt{T}' \in tc(\mathtt{e}_1) \cup tc(\mathtt{e}_2)$ using the function tc, which is defined in Figure 10. Moreover, the resulting session type is obtained by replacing the occurrences of $\circledcirc$ by a fresh variable α, which is bound by the μ operator.

Observe that in the rules SendC-T and SendW-T, typing e with session type ε prevents e from containing occurrences of communications and $\mathtt{cont}^{\mathtt{T}}$. However, this restriction is not significant. If e contained communications, a possible dual for the sendW expression should be able to perform the dual communications at each iteration before receiving the object that would select its continuation. Such a dual should be of the form

$$\mathtt{e}'\,;\mathtt{receiveW(x)}\{\mathtt{C}_1 \Rightarrow \ldots\,;\mathtt{e}'\,;\mathtt{cont}^{\mathtt{T}}\,;\ldots.\,[]\,\mathtt{C}_2 \Rightarrow \ldots\,;\},$$

where $\mathtt{e}'$ contains the dual communications of e. This suggests how we can encode a sendW expression with communications inside the argument in our system. In order to maintain a sort of symmetry, we also have this restriction in the typing of sendC. Note that this problem only affects communications in the current sessions, and has no effect on new sessions opened in e: in fact, the typing allows e to contain session requests.

Figure 11 defines well-formed class tables. Rule Sess-WF says that a session declaration in a class C is well typed if its body has the declared return type and session type by assuming that this is of type C. Note 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 cont-free in rule Sess-WF, which implies that well-typed session bodies do not contain free occurrences of $\mathtt{cont}^{\mathtt{T}}$.

We conclude this section by observing that the system presented in Figure 9, given a typable expression e and the related class table, actually *infers* the session type of e. In fact, that system is itself an inference algorithm of the session type of an expression, which expects session types of sessions to be declared in the class table.

It is easy to prove the following proposition by induction on typing rules.

Proposition 5.1. If $\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}$ and $\Gamma \vdash \mathtt{e} : \mathtt{T}' \fatsemi \mathtt{t}'$, then $\mathtt{t} = \mathtt{t}'$.

The unicity of session types follows from the fact that receiving actions are modelled through expressions in which the classes of received objects are explicitly declared. This is a characteristic feature of our approach to session types compared with standard systems (Yoshida and Vasconcelos 2007).

In Section 7, we will present an inference algorithm that reconstructs the session types of session declarations given a class table where these session types are omitted.

5.2. *Typing of run-time expressions*

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

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

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

A session environment only maps a finite set of channels to session types different from ε, and all the rest to ε. We can then represent one session environment with an infinite number of finite sets that 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 12 gives the typing rules for run-time expressions, which differ from those for channel-free expressions by having session environments instead of a unique session type. For this reason, we extend the *concatenation* of session types to *session environments* as follows:

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

In rule NewC-RT, the expressions for field initialisation can be partially evaluated, and for this reason, they can contain channel names. For example,

$$\mathtt{new}\ \mathtt{C}(\mathtt{o.s}\{\mathtt{sendC}(5)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}\})$$

evaluates to

$$\mathtt{new}\ \mathtt{C}(\mathtt{k.sendC}(5)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \wr \mathtt{k}\int \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2 \wr \mathtt{k}\int\}) \;||\; \tilde{\mathtt{k}}.\mathtt{recvC}(\mathtt{x})\{\mathtt{C}'_1 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_1 \wr \tilde{\mathtt{k}}\int \,[]\, \mathtt{C}'_2 \Rightarrow [\mathtt{o}/\mathtt{this}]\mathtt{e}'_2 \wr \tilde{\mathtt{k}}\int\}$$

if

$$\mathtt{recvC}(\mathtt{x})\{\mathtt{C}'_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{e}'_2\}$$

is the body of session $\mathtt{s}$ in the class of object $\mathtt{o}$.

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

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

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

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

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

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

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

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

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

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

$$\textsc{SendW-RT}\quad \frac{\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \varnothing \qquad \Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\} \qquad \mathtt{T} <: \mathtt{T}' \qquad \forall \mathtt{T}' \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2) \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}}$$

$$\textsc{ReceiveW-RT}\quad \frac{\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\} \qquad \mathtt{T} <: \mathtt{T}' \qquad \forall \mathtt{T}' \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2) \qquad \alpha \text{ fresh in } \mathtt{t}_1, \mathtt{t}_2}{\Gamma \vdash_{\mathtt{r}} \mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}}$$

Fig. 12. Typing rules for run-time expressions.

In rule SessReq-RT, we make use of the judgement $\Gamma \vdash \mathtt{e}' : \mathtt{T} \fatsemi \mathtt{t}'$, where the expression $\mathtt{e}'$ does not contain channels, but may contain object identifiers. For example, by reducing

$$\mathtt{k.recvC(x)}\{\mathtt{C} \Rightarrow \mathtt{o.s}\,\{\mathtt{sendC(x)}\{\_\}\} \,[]\, \_\}, h$$

where $h(\mathtt{k}) = \mathtt{o}'$ and $h(\mathtt{o}') = (\mathtt{C}, \_)$, we get $\mathtt{o.s}\,\{\mathtt{sendC}(\mathtt{o}')\{\_\}\}$. This justifies our use of channel-free expressions rather than user expressions in the typing rules of Section 5.1.

Note that in the typing of communications expressions, the expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ in the two branches only contain the current channel $\mathtt{k}$ as subject since channels are only created at run time, and these expressions will never be reduced before the selection has been done. In rule SendC-RT, we know that the session environment Σ, which is obtained by typing the expression $\mathtt{e}$, cannot contain occurrences of the channel $\mathtt{k}$ since $\mathtt{e}$ is obtained by reducing a channel-free expression with session type ε, as prescribed by rule SendC-T. In rule SendW-RT, we can assume the emtpy session environment for typing the expression

`e` since the evaluation of `e` cannot start before the `sendW` expression has been unfolded to a `sendC`.

The following lemma gives the weakening property for term environments.

Lemma 5.1 (weakening). Let $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$. Then:

(1) If $\mathtt{x} \notin \mathsf{dom}(\Gamma)$, then $\Gamma, \mathtt{x} : \mathtt{T}' \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$.
(2) If $\mathtt{o} \notin \mathsf{dom}(\Gamma)$, then $\Gamma, \mathtt{o} : \mathtt{C} \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$. □

The typing rules for run-time expressions only differ from those for user expressions in assigning the session type to explicit channels, and not in the type `T`.

The relation between the two systems is clarified by the following proposition, which will be useful in showing the subject reduction property.

Proposition 5.2. $\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}$ implies $\Gamma \vdash_{\mathtt{r}} \mathtt{e} \wr \mathtt{k} \int : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}\}$.

Note that $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \varnothing$ is equivalent to $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \{\mathtt{k} : \varepsilon\}$ by our convention on session environments. Analogously, $\Sigma = \Sigma.\varnothing = \varnothing.\Sigma$ for any Σ.

As a final remark, note that we do not provide an explicit rule for typing parallel threads. Typing rules give type to single (run-time) expressions only, while expressions can also reduce to parallel threads by reduction rules. Indeed, in this case, we only use the notion of well-typedness: a parallel composition of expressions is considered to be well typed (in the environment Γ) if each single expression is typed (in Γ). We will take this point into account when formulating the subject reduction property in Section 6, where we prove that the semantics preserves typing.

6. Properties

In this section we prove type safety, which is the fundamental property ensuring that our system is well founded.

A program consists of a set of declarations and a main expression to be evaluated. So a well-typed executable program means that the induced class table is well formed and the main expression is typed, using that class table, according to the rules of Figure 9. We require the main expression to be a typable closed user expression; furthermore, all communication and delegation expressions must occur inside session co-bodies. It is easy to verify that this is equivalent to requiring typability in the system of Section 5.1 from the empty term environment with an empty session type using a well-formed class table. We introduce the notion of initial expression as follows.

Definition 6.1. An *initial* expression `e` is an expression satisfying $\varnothing \vdash \mathtt{e} : \mathtt{T} \fatsemi \varepsilon$ for some `T`.

Proposition 6.1. An *initial* expression `e` is a closed and channel-complete user expression.

For example, the expressions $\mathtt{sendC(o)}\{\ldots\}$ and $\mathtt{o} \bullet \mathtt{s}\,\{\}$ are not initial expressions since if they are typed, their term environments and session types are not empty.

Proposition 5.2 guarantees that initial expressions are also given the same type, with no assumption about communications, using the typing for run-time expressions.

The type safety property ensures that the evaluation of an initial expression cannot get stuck. The proof is carried out in two steps. First we state the subject reduction property, that is, we prove that not only are types preserved, but the heap also evolves during the evaluation in a way that is consistent with the term and session environments. Then we prove type safety, dealing with the crucial case of communication expressions to show that they cannot get stuck on a communication deadlock.

6.1. *Subject reduction*

We begin with some preliminary definitions and lemmas required for the proof of the Subject Reduction Theorem.

The first definition formalises the evolution of session types and session environments.

Definition 6.2.

(1) A session type $\mathtt{t}'$ is at a *later stage* than another session type $\mathtt{t}$, written $\mathtt{t} \sqsubseteq \mathtt{t}'$, if it is deducible from the following rules:

$$\textsc{Later-0}\ \frac{}{\mathtt{t} \sqsubseteq \varepsilon} \qquad \textsc{Later-1}\ \frac{}{\mathtt{t} \sqsubseteq \mathtt{t}} \qquad \textsc{Later-2}\ \frac{\mathtt{t} \sqsubseteq \mathtt{t}'' \quad \mathtt{t}'' \sqsubseteq \mathtt{t}'}{\mathtt{t} \sqsubseteq \mathtt{t}'}$$

$$\textsc{Later-3}\ \frac{\mathtt{t} \sqsubseteq \mathtt{t}'}{\mathtt{t}.\mathtt{t}'' \sqsubseteq \mathtt{t}'.\mathtt{t}''} \qquad \textsc{Later-4}\ \frac{}{\dagger\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\} \sqsubseteq \mathtt{t}_i}$$

(2) A session environment Σ' is at a *later stage* than another session environment Σ, $\Sigma \sqsubseteq \Sigma'$, if $\mathtt{k} : \mathtt{t} \in \Sigma$ and $\mathtt{t} \neq \varepsilon$ imply $\mathtt{k} : \mathtt{t}' \in \Sigma'$ and $\mathtt{t} \sqsubseteq \mathtt{t}'$.

The evolution of session environments also takes into account the fact that new channels can be created by session calls, so, for example, assuming that $\mathtt{t}_3$ and $\mathtt{t}_4$ are dual, we have

$$\{\mathtt{k} : !\{\mathtt{Shape} \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{String} \Rightarrow \mathtt{t}_2\}\} \sqsubseteq \{\mathtt{k} : \mathtt{t}_1, \mathtt{k}_1 : \mathtt{t}_3, \tilde{\mathtt{k}}_1 : \mathtt{t}_4\}.$$

The next definition gives standard conditions on heap well-formedness and agreement between heaps and term environments.

Definition 6.3 (well-formed heap and agreement). A term environment Γ and a heap h *agree*, written $ag(\Gamma; h)$, if both:

(1) h is well formed, that is

$$\begin{array}{rcl} h(\mathtt{o}) & = & (\mathtt{C}, \overline{\mathtt{f} = \mathtt{o}}) \\ \mathit{ftype}_r(\mathtt{C}, \mathtt{f}_i) = \mathtt{T} & \Rightarrow & h(\mathtt{o})(\mathtt{f}_i) = (\mathtt{C}', \_) \\ \mathtt{C}' & <: \mathtt{T}. & \end{array}$$

(2) The classes of objects in h are the classes associated with them by Γ, that is,

$$\forall \mathtt{o} \in \mathsf{dom}(\Gamma),\ h(\mathtt{o}) = (\Gamma(\mathtt{o}), \_).$$

In part (1) of the above definition, recall that $\mathit{ftype}_r(\mathtt{C}, \mathtt{f}_i) = \mathit{ftype}_w(\mathtt{C}, \mathtt{f}_i)$, where $\mathtt{C}$ is a class.

The following lemma states the obvious property that in any type derivation ending with rule Sub-RT, there is a subderivation giving a subtype to the same expression such that its final rule is different from Sub-RT.

Lemma 6.1. In any derivation of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma$, there is a subderivation of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$, with $\mathtt{T} <: \mathtt{T}'$, where the last applied rule is different from Sub-RT.

Proof. The proof is by a straightforward induction on the derivation of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$. □

Lemma 6.1 means that in the following proofs we can assume without loss of generality that the given typing derivations end with a rule different from Sub-RT.

In order to simplify the proof of Subject Reduction, we will first show the preservation of typing under the substitution of subexpressions. In our calculus, the difficulty is that we must deal carefully with session environments in substitutions.

Lemma 6.2 uses evaluation contexts as defined in Section 4. Note that this lemma does not require that the expression in the hole of the context be a redex.

Lemma 6.2 (evaluation context substitution). In any derivation of $\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}] : \mathtt{T} \fatsemi \Sigma$, there exist Σ_1, Σ_2, $\mathtt{T}'$ such that:

(1) There is a subderivation of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma_1$ and $\Sigma = \Sigma_1.\Sigma_2$.
(2) $\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}'] : \mathtt{T} \fatsemi \Sigma'_1.\Sigma_2$, for any $\mathtt{e}'$ such that $\Gamma \vdash_{\mathtt{r}} \mathtt{e}' : \mathtt{T}' \fatsemi \Sigma'_1$ with $\Sigma_1 \sqsubseteq \Sigma'_1$.

Proof. The proof is by induction on the definition of $\mathscr{E}$. The base case is when $\mathscr{E}$ is the empty context and is trivial. In the induction step, each case proceeds by analysing the final rule used in the derivation of $\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}] : \mathtt{T} \fatsemi \Sigma$, which is assumed to be different from Sub-RT by Lemma 6.1.

— $\mathscr{E}[\mathtt{e}] = \mathscr{E}'[\mathtt{e}]; \mathtt{e}''$:
This case is immediate from rule Seq-RTand the induction hypothesis.

— $\mathscr{E}[\mathtt{e}] = \mathscr{E}'[\mathtt{e}].\mathtt{s}\,\{\mathtt{e}''\}$:
The final rule is SessReq-RT, which implies that

$$\Sigma = \Sigma'.\Sigma''$$

and there is a subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma'.$$

Hence, by the induction hypothesis, we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma_1$$

where $\Sigma' = \Sigma_1.\Sigma_2'$, that is, $\Sigma = \Sigma_1.\Sigma_2$ by taking $\Sigma_2 = \Sigma_2'.\Sigma''$. Moreover, by the induction hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma'$$

implies

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}'] : \mathtt{T}_1 \fatsemi \Sigma_1'.\Sigma_2'.$$

Hence, we can substitute a derivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}'] : \mathtt{T}_1 \fatsemi \Sigma_1'.\Sigma_2'$$

for the subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma_1.\Sigma_2',$$

in the derivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}] : \mathtt{T} \fatsemi \Sigma_1.\Sigma_2.$$

Rule SessReq-RT still applies since the other premises stay the same, so we obtain

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}'] : \mathtt{T} \fatsemi \Sigma_1'.\Sigma_2.$$

— $\mathscr{E}[\mathtt{e}] = \mathscr{E}'[\mathtt{e}] \bullet \mathtt{s}\,\{\mathtt{k}\}$:
The final rule is SessDel-RT, which implies that there is a subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma'$$

such that

$$\begin{aligned} \Sigma &= \Sigma'.\{\mathtt{k} : \mathtt{t}\} \\ \mathit{stype}(\mathtt{s}, \mathtt{T}_1) &= \{\mathtt{t}\} \\ \mathit{rtype}(\mathtt{s}, \mathtt{T}_1) &= \mathtt{T}. \end{aligned}$$

Hence, by the induction hypothesis, we have a subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma_1$$

where $\Sigma' = \Sigma_1.\Sigma_2'$ for some Σ_2'. So we have $\Sigma = \Sigma_1.\Sigma_2$ by defining $\Sigma_2 = \Sigma_2'.\{\mathtt{k} : \mathtt{t}\}$. Moreover, by the induction hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma'$$

implies

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}'] : \mathtt{T}_1 \fatsemi \Sigma_1'.\Sigma_2',$$

so we can replace

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{T}_1 \fatsemi \Sigma_1.\Sigma_2'$$

with

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}'] : \mathtt{T}_1 \fatsemi \Sigma_1'.\Sigma_2'$$

in the derivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}] : \mathtt{T} \fatsemi \Sigma_1.\Sigma_2$$

and rule SessDel-RT still applies. So we obtain

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{e}'] : \mathtt{T} \fatsemi \Sigma'_1.\Sigma_2.$$

— $\mathscr{E}[\mathtt{e}] = \mathtt{k.sendC}(\mathscr{E}'[\mathtt{e}])\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}$:
The final rule is SendC-RT, which implies that there is a subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \Sigma',$$

such that

$$\Sigma = \Sigma', \{\mathtt{k} :! \{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}$$
$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\}.$$

Hence, by the induction hypothesis, there is a subderivation of

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T}' \fatsemi \Sigma_1$$

where

$$\Sigma' = \Sigma_1.\Sigma'',$$

for some Σ'', that is, we have $\Sigma = \Sigma_1.\Sigma_2$ by taking

$$\Sigma_2 = \Sigma'', \{\mathtt{k} :! \{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}.$$

Moreover, the induction hypothesis on

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}] : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \Sigma_1.\Sigma''$$

tells us that

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}'[\mathtt{e}'] : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \Sigma'_1.\Sigma''.$$

Hence, since

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i\},$$

we obtain

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendC}(\mathscr{E}'[\mathtt{e}'])\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \Sigma'_1.\Sigma_2,$$

by rule SendC-RT.

The remaining cases follow straightforwardly by using the same proof pattern as in the above cases. □

Lemma 6.3 (term substitution).

(1) If

$$\Gamma, \mathtt{z} : \mathtt{C} \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$$
$$\Gamma \vdash_{\mathtt{r}} \mathtt{o} : \mathtt{C} \fatsemi \varnothing,$$

then

$$\Gamma \vdash_{\mathtt{r}} [\mathtt{o}/\mathtt{z}]\mathtt{e} : \mathtt{T} \fatsemi \Sigma.$$

(2) If

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\},$$

then, for $i \in \{1, 2\}$,

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i\lfloor \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}\rfloor : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i'\}$$

where

$$\mathtt{t}_i' = [\mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}/\alpha]\mathtt{t}_i.$$

(3) If

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\},$$

then, for $i \in \{1, 2\}$,

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i\lfloor \mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}\rfloor : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i'\}$$

where

$$\mathtt{t}_i' = [\mu\alpha.?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}/\alpha]\mathtt{t}_i.$$

Proof.

(1) This part is immediate by substituting

$$\Gamma \vdash_{\mathtt{r}} \mathtt{o} : \mathtt{C} \fatsemi \varnothing$$

for

$$\Gamma, \mathtt{z} : \mathtt{C} \vdash_{\mathtt{r}} \mathtt{z} : \mathtt{C} \fatsemi \varnothing,$$

in any derivation of

$$\Gamma, \mathtt{z} : \mathtt{C} \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma.$$

(2) By rule SendW-RT, we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}$$

implies

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}_i \fatsemi \{\mathtt{k} : \mathtt{t}_i''\},$$

for $\mathtt{t}_i'' = [\circledcirc/\alpha]\mathtt{t}_i$ and $\mathtt{T} <: \mathtt{T}'$ for all $\mathtt{T}' \in tc(\mathtt{e}_1) \cup tc(e_2)$. The last condition and the definition of tc ensure that if $\mathtt{cont}^{\mathtt{T}'}$ occurs free in $\mathtt{e}_1$ or $\mathtt{e}_2$, then $\mathtt{T} <: \mathtt{T}'$, so any free occurrence of $\mathtt{cont}^{\mathtt{T}'}$ in $\mathtt{e}_i$ is given type by

$$\Gamma \vdash_{\mathtt{r}} \mathtt{cont}^{\mathtt{T}'} : \mathtt{T}' \fatsemi \{\mathtt{k} : \circledcirc\}.$$

From

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\},$$

we derive

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}' \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}$$

by rule SUB-RT. Observe that the only constraint satisfied by $\circledcirc$ that appears in the premises of the typing rules is $\circledcirc \neq \varepsilon$ since no session type is the dual of $\circledcirc$. Thus, if we replace

$$\Gamma \vdash_{\mathtt{r}} \mathtt{cont}^{\mathtt{T}'} : \mathtt{T}' \fatsemi \{\mathtt{k} : \circledcirc\}$$

by

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}' \fatsemi \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\},$$

inside the derivation of

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}_i \fatsemi \{\mathtt{k} : \mathtt{t}_i''\},$$

we obtain a derivation of

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i[\mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}] : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}_i'\}.$$

(3) This part is similar to part (1), but using RECEIVEW-RT in place of SENDW-RT. □

Lemma 6.4 (typing of session bodies). If

$$\begin{aligned} \mathit{sbody}(\mathtt{s}, \mathtt{C}) &= \mathtt{e} \\ \mathit{stype}(\mathtt{s}, \mathtt{C}) &= \mathtt{t} \\ \mathit{rtype}(\mathtt{s}, \mathtt{C}) &= \mathtt{T}, \end{aligned}$$

then

$$\{\mathtt{this} : \mathtt{C}\} \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}.$$

Proof. The proof is by induction on the definition of $\mathit{sbody}(\mathtt{s}, \mathtt{C})$, using the definitions of *stype* and *rtype*. In the base case, s is defined in C and the proof then follows from rule SESS-WF. The induction step is straightforward. □

We can now prove the Subject Reduction Theorem. We will only type single expressions, though they can result in parallel threads. Since we do not have a typing for parallel threads, we require each single expression to be well typed. Moreover, we want to get our property in the most general form, allowing the property to hold for all well-typed expressions, which can sometimes only be generated by initial expressions in parallel with other expressions. For example, no initial expression can reduce to the expression

$$\mathtt{e} = \mathtt{o.s}\,\{\mathtt{sendC(5)}\{\mathtt{e}_1\}\};\mathtt{k.sendC(3)}\{\mathtt{e}_2 \wr \mathtt{k}\int\},$$

but

$$\mathtt{e}_0 = \mathtt{o}'.\mathtt{s}'\,\{\mathtt{o.s}\,\{\mathtt{sendC(5)}\{\mathtt{e}_1\}\};\mathtt{sendC(3)}\{\mathtt{e}_2\}\}$$

reduces to

$$\mathtt{e} \parallel \tilde{\mathtt{k}}.\mathtt{recvC(x)}\{[\mathtt{o}'/\mathtt{this}]\mathtt{e}' \wr \tilde{\mathtt{k}}\int\}$$

if $\mathtt{recvC(x)}\{\mathtt{e}'\}$ is the body of session $\mathtt{s}'$ in the class of the object $\mathtt{o}'$.

Note also that receive expressions can never get objects of the wrong types. For example, the execution of

$$\mathtt{k.recvC(x)}\{\mathtt{Bool} \Rightarrow \neg\mathtt{x} \,[\!]\, \mathtt{Int} \Rightarrow -\mathtt{x}\}$$

if $h(\mathtt{k})=\mathtt{"a"}$ is simply stopped, that is, it does not produce a run-time error. In fact, the reduction rule ReceiveCase-R requires the class of the object in the heap to be a subclass of at least one of the classes declared in the `recvC` expression. Note that such a configuration cannot be generated starting from an initial expression. For this reason, in contrast to the calculus of Coppo *et al.* (2007), we do not need to require agreement between the objects in the queues associated with channels by the heap and the session types of the same channels in the session environment.

Theorem 6.1 (subject reduction). If $ag(\Gamma;h)$ and $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$, then:

(1) $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$ implies that there exist Σ', Γ' such that

$$\Gamma \subseteq \Gamma'$$
$$\Sigma \sqsubseteq \Sigma'$$
$$ag(\Gamma';h')$$
$$\Gamma' \vdash_{\mathtt{r}} \mathtt{e}' : \mathtt{T} \fatsemi \Sigma'.$$

(2) $\mathtt{e}, h \longrightarrow \mathtt{e}_1 \parallel \mathtt{e}_2, h'$ implies that

$$h' = h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]$$

for some fresh $\mathtt{k}$, and $ag(\Gamma;h')$, and that there exist $\mathtt{T}', \mathtt{t}, \mathtt{t}'$ such that

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_1 : \mathtt{T} \fatsemi \Sigma \cup \{\mathtt{k} : \mathtt{t}\}$$
$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_2 : \mathtt{T}' \fatsemi \{\tilde{\mathtt{k}} : \mathtt{t}'\},$$

and $\mathtt{t} \bowtie \mathtt{t}'$.

Proof. The proof is by induction on the definition of $\longrightarrow$. We proceed by case analysis on the final rule applied (by Lemma 6.1, we only need to consider cases in typing derivations of $\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{T} \fatsemi \Sigma$ where the last rule applied is different from Sub-RT):

— SessReq-R:

$$\frac{h(\mathtt{o}) = (\mathtt{C}, \_) \quad sbody(\mathtt{s}, \mathtt{C}) = \mathtt{e}' \quad \mathtt{k}, \tilde{\mathtt{k}} \notin h}{\mathscr{E}[\mathtt{o.s}\,\{\mathtt{e}\}], h \longrightarrow \mathscr{E}[\mathtt{e} \wr \mathtt{k} \int] \parallel [\mathtt{o}/\mathtt{this}]\mathtt{e}' \wr \tilde{\mathtt{k}} \int, h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]}.$$

By $h(\mathtt{o}) = (\mathtt{C}, \_)$ and $ag(\Gamma;h)$, we get

$$\Gamma \vdash_{\mathtt{r}} \mathtt{o} : \mathtt{C} \fatsemi \varnothing$$

using Axiom-RT.
By hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{o.s}\,\{\mathtt{e}\}] : \mathtt{T} \fatsemi \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{o.s}\,\{\mathtt{e}\} : \mathtt{T}' \fatsemi \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule SessReq-RT, we have

$$\begin{array}{c} \Sigma_1 = \varnothing \\ \Sigma_2 = \Sigma \\ \Gamma \vdash \mathtt{e} : \mathtt{T}' \fatsemi \mathtt{t}' \\ \mathit{stype}(\mathtt{s}, \mathtt{C}) = \mathtt{t} \\ \mathtt{t} \bowtie \mathtt{t}'. \end{array}$$

By Proposition 5.2, we get

$$\Gamma \models_{\mathtt{r}} \mathtt{e} \wr \mathtt{k} \int : \mathtt{T}' \fatsemi \{\mathtt{k} : \mathtt{t}'\}.$$

By Lemma 6.2 (2), we have

$$\Gamma \models_{\mathtt{r}} \mathscr{E}[\mathtt{e} \wr \mathtt{k} \int] : \mathtt{T} \fatsemi \{\mathtt{k} : \mathtt{t}'\}.\Sigma.$$

Let $\mathit{rtype}(\mathtt{s}, \mathtt{C}) = \mathtt{T}_0$. So

$$\mathtt{this} : \mathtt{C} \vdash \mathtt{e}' : \mathtt{T}_0 \fatsemi \mathtt{t}$$

by Lemma 6.4, which implies

$$\mathtt{this} : \mathtt{C} \models_{\mathtt{r}} \mathtt{e}' \wr \tilde{\mathtt{k}} \int : \mathtt{T}_0 \fatsemi \{\tilde{\mathtt{k}} : \mathtt{t}\}$$

by Proposition 5.2. Therefore, by Lemmas 5.1 and 6.3 (1), we can conclude that

$$\Gamma \models_{\mathtt{r}} [\mathtt{o}/\mathtt{this}]\mathtt{e}' \wr \tilde{\mathtt{k}} \int : \mathtt{T}_0 \fatsemi \{\tilde{\mathtt{k}} : \mathtt{t}\}.$$

Note that the new heap $h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]$ still agrees with Γ since the only changes are about channels.

— SessDel-R:

$$\frac{h(\mathtt{o}) = (\mathtt{C}, \_) \quad \mathit{sbody}(\mathtt{s}, \mathtt{C}) = \mathtt{e}}{\mathscr{E}[\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}], h \longrightarrow \mathscr{E}[[\mathtt{o}/\mathtt{this}]\mathtt{e} \wr \mathtt{k} \int], h}.$$

By $h(\mathtt{o}) = (\mathtt{C}, \_)$ and $ag(\Gamma; h)$, we get

$$\Gamma \models_{\mathtt{r}} \mathtt{o} : \mathtt{C} \fatsemi \varnothing$$

using Axiom-RT.
By hypothesis,

$$\Gamma \models_{\mathtt{r}} \mathscr{E}[\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}] : \mathtt{T} \fatsemi \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \models_{\mathtt{r}} \mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\} : \mathtt{T}' \fatsemi \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule SessDel-RT, we have

$$\begin{array}{c} \Sigma_1 = \{\mathtt{k} : \mathtt{t}\} \\ \mathit{stype}(\mathtt{s}, \mathtt{C}) = \mathtt{t} \\ \mathit{rtype}(\mathtt{s}, \mathtt{C}) = \mathtt{T}'. \end{array}$$

By Lemma 6.4,

$$\texttt{this} : \texttt{C} \vdash \texttt{e} : \texttt{T}' \fatsemi \texttt{t},$$

so by Proposition 5.2,

$$\texttt{this} : \texttt{C} \models_{\texttt{r}} \texttt{e} \wr \texttt{k} \int : \texttt{T}' \fatsemi \{\texttt{k} : \texttt{t}\}.$$

Therefore, by Lemmas 5.1 and 6.3 (1), we have

$$\Gamma \models_{\texttt{r}} [\texttt{o}/\texttt{this}]\texttt{e} \wr \texttt{k} \int : \texttt{T}' \fatsemi \{\texttt{k} : \texttt{t}\},$$

and by Lemma 6.2 (2), we can conclude that

$$\Gamma \models_{\texttt{r}} \mathscr{E}[[\texttt{o}/\texttt{this}]\texttt{e} \wr \texttt{k} \int] : \texttt{T} \fatsemi \Sigma.$$

— SendCase-R:

$$\frac{h(\tilde{\texttt{k}}) = \overline{\texttt{o}} \quad h(\texttt{o}) = (\texttt{C}, \_) \quad \texttt{C} \Downarrow \{\texttt{C}_1, \texttt{C}_2\} = \texttt{C}_i}{\mathscr{E}[\texttt{k.sendC(o)}\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\}], h \longrightarrow \mathscr{E}[\texttt{e}_i], h[\tilde{\texttt{k}} \mapsto \overline{\texttt{o}} :: \texttt{o}]}.$$

By $h(\texttt{o}) = (\texttt{C}, \_)$ and $ag(\Gamma; h)$, we get

$$\Gamma \models_{\texttt{r}} \texttt{o} : \texttt{C} \fatsemi \varnothing$$

using Axiom-RT. By hypothesis,

$$\Gamma \models_{\texttt{r}} \mathscr{E}[\texttt{k.sendC(o)}\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\}] : \texttt{T} \fatsemi \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \models_{\texttt{r}} \texttt{k.sendC(o)}\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\} : \texttt{T}' \fatsemi \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule SendC-RT, we have

$$\Sigma_1 = \{\texttt{k} :! \{\texttt{C}_1 \Rightarrow \texttt{t}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{t}_2\}\}$$
$$\Gamma \models_{\texttt{r}} \texttt{e}_i : \texttt{T}' \fatsemi \{\texttt{k} : \texttt{t}_i\}.$$

By Lemma 6.2 (2), we have

$$\Gamma \models_{\texttt{r}} \mathscr{E}[\texttt{e}_i] : \texttt{T} \fatsemi \Sigma',$$

where

$$\Sigma' = \{\texttt{k} : \texttt{t}_i\}.\Sigma_2.$$

From Definition 6.2 (Later-3 and Later-4), we can then conclude that $\Sigma \sqsubseteq \Sigma'$. Note that the new heap $h[\tilde{\texttt{k}} \mapsto \overline{\texttt{o}} :: \texttt{o}]$ still agrees with Γ since the only changes are about channels.

— ReceiveCase-R:

$$\frac{h(\texttt{k}) = \texttt{o} :: \overline{\texttt{o}} \quad h(\texttt{o}) = (\texttt{C}, \_) \quad \texttt{C} \Downarrow \{\texttt{C}_1, \texttt{C}_2\} = \texttt{C}_i}{\mathscr{E}[\texttt{k.recvC(x)}\{\texttt{C}_1 \Rightarrow \texttt{e}_1 \,[]\, \texttt{C}_2 \Rightarrow \texttt{e}_2\}], h \longrightarrow \mathscr{E}[[\texttt{o}/\texttt{x}]\texttt{e}_i], h[\texttt{k} \mapsto \overline{\texttt{o}}]}.$$

By $h(\texttt{o}) = (\texttt{C}, \_)$ and $ag(\Gamma; h)$, we get

$$\Gamma \models_{\texttt{r}} \texttt{o} : \texttt{C} \fatsemi \varnothing$$

using Axiom-RT. Applying rule Sub-RT, we get

$$\Gamma \vdash_{\mathtt{r}} \mathtt{o} : \mathtt{C}_i \,;\, \varnothing.$$

By hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}] : \mathtt{T} \,;\, \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}' \,;\, \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule ReceiveC-RT, we have

$$\Sigma_1 = \{\mathtt{k} :?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}$$
$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}' \,;\, \{\mathtt{k} : \mathtt{t}_i\}.$$

By Lemma 6.3 (1), we have

$$\Gamma \vdash_{\mathtt{r}} [\mathtt{o}/\mathtt{x}]\mathtt{e}_i : \mathtt{T}' \,;\, \{\mathtt{k} : \mathtt{t}_i\}.$$

By Lemma 6.2 (2), we have

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[[\mathtt{o}/\mathtt{x}]\mathtt{e}_i] : \mathtt{T} \,;\, \Sigma',$$

where

$$\Sigma' = \{\mathtt{k} : \mathtt{t}_i\}.\Sigma_2.$$

From Definition 6.2 (Later-3 and Later-4), we can then conclude that $\Sigma \sqsubseteq \Sigma'$.
Note that the new heap $h[\mathtt{k} \mapsto \overline{\mathtt{o}}]$ still agrees with Γ since the only changes are about channels.

— SendWhile-R:

$$\mathscr{E}[\mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}], h \longrightarrow \mathscr{E}[\mathtt{k.sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}], h$$

where

$$\mathtt{e}'_i = \mathtt{e}_i\lfloor \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}\rfloor.$$

By hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}] : \mathtt{T} \,;\, \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}' \,;\, \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule SendW-RT, we have

$$env S_1 = \{\mathtt{k} : \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}$$
$$\Gamma \vdash_{\mathtt{r}} \mathtt{e} : \mathtt{C}_1 \vee \mathtt{C}_2 \,;\, \varnothing$$
$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}' \,;\, \{\mathtt{k} : \mathtt{t}_i\}$$

and α fresh in $\mathtt{t}_1, \mathtt{t}_2$ and $\mathtt{T}' <: \mathtt{T}''$ for all $\mathtt{T}'' \in tc(\mathtt{e}_1) \cup tc(\mathtt{e}_2)$.
Let

$$\mathtt{t}'_i = [(\mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\})/\circledcirc]\mathtt{t}_i.$$

By Lemma 6.3 (2), we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{e}'_i : \mathtt{T}' \fatsemi \{\mathtt{k} : \mathtt{t}'_i\}.$$

By rule SendC-RT, we get

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\} : \mathtt{T}' \fatsemi \mathtt{k} :!\{\mathtt{C}_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}'_2\},$$

and by Lemma 6.2 (2), we can conclude that

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{k.sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}] : \mathtt{T} \fatsemi \Sigma',$$

where

$$\Sigma' = \{\mathtt{k} :!\{\mathtt{C}_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}'_2\}\}.\Sigma_2$$

and $\Sigma \sqsubseteq \Sigma'$ by Definition 6.2 (Later-1 and Later-3), since we consider recursive types modulo fold/unfold.

— ReceiveWhile-R:

$$\mathscr{E}[\mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}], h \longrightarrow \mathscr{E}[\mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}], h$$

where

$$\mathtt{e}'_i = \mathtt{e}_i\lfloor \mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}/\mathtt{cont}\rfloor.$$

By hypothesis,

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\}] : \mathtt{T} \fatsemi \Sigma,$$

so by Lemma 6.2 (1), we have

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}' \fatsemi \Sigma_1$$

and $\Sigma = \Sigma_1.\Sigma_2$. From rule ReceiveW-RT, we have

$$\Sigma_1 = \{\mathtt{k} : \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\}\}$$
$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}' \fatsemi \{\mathtt{k} : \mathtt{t}_i\},$$

and α fresh in $\mathtt{t}_1, \mathtt{t}_2$ and $\mathtt{T}' <: \mathtt{T}''$ for all $\mathtt{T}'' \in tc(\mathtt{e}_1) \cup tc(\mathtt{e}_2)$.
Let

$$\mathtt{t}'_i = [(\mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\mathtt{t}_2\})/\circledcirc]\mathtt{t}_i.$$

By Lemma 6.3 (3), we have

$$\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathtt{r}} \mathtt{e}'_i : \mathtt{T}' \fatsemi \{\mathtt{k} : \mathtt{t}'_i\}.$$

By rule ReceiveC-RT, we get

$$\Gamma \vdash_{\mathtt{r}} \mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\} : \mathtt{T}' \fatsemi \mathtt{k} :?\{\mathtt{C}_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}'_2\}.$$

By Lemma 6.2 (2), we can then conclude that

$$\Gamma \vdash_{\mathtt{r}} \mathscr{E}[\mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}] : \mathtt{T} \fatsemi \Sigma',$$

where

$$\Sigma' = \{\mathtt{k} :?\{\mathtt{C}_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}'_2\}\}.\Sigma_2$$

and $\Sigma \sqsubseteq \Sigma'$ by Definition 6.2 (LATER-1 and LATER-3), since we consider recursive types modulo fold/unfold.

The remaining cases follow easily from the induction hypothesis. □

Using the Subject Reduction Theorem, we can show that expressions, which are obtained by reducing initial expressions, are typed from environments that agree with the current heap.

Corollary 6.1. If $\mathtt{e}$ is an initial expression and $\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}' \mid\mid P, h$, then $\Gamma \vdash_{\mathtt{T}} \mathtt{e}' : \mathtt{T} \fatsemi \Sigma$ for some Γ, $\mathtt{T}$, Σ such that $ag(\Gamma, h)$.

Proof. The proof is by induction on $\longrightarrow^*$. The base case is immediate by the definition of initial expression. In the induction case, by definition,

$$\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}' \mid\mid P, h$$

means

$$\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}_1 \mid\mid \mathtt{e}_2 \mid\mid \ldots \mid\mid \mathtt{e}_n, h'$$

and either

$$\mathtt{e}_1, h' \longrightarrow \mathtt{e}', h \quad \text{and} \quad P \equiv \mathtt{e}_2 \mid\mid \ldots \mid\mid \mathtt{e}_n$$

or

$$\mathtt{e}_1, h' \longrightarrow \mathtt{e}' \mid\mid \mathtt{e}'', h \quad \text{and} \quad P \equiv \mathtt{e}'' \mid\mid \mathtt{e}_2 \mid\mid \ldots \mid\mid \mathtt{e}_n.$$

By the induction hypothesis, $\mathtt{e}_1$ is well typed from a term environment that agrees with h'. Therefore, $\mathtt{e}'$ is well typed from a term environment that agrees with h by Theorem 6.1. □

6.2. *Type safety*

The run-time errors our type system has to prevent 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.

In particular, for the second point, we want to show that the communications of well-typed sessions cannot become stuck. To this end, we have to study the global properties of type preservation during the reduction of parallel threads: specifically, we need to take into account the objects in the queues associated with channels and their relations with the session types of the channels themselves.

In the following definition we extend the notion of duality between session types to take account also of the objects already sent by a thread and waiting to be read by the thread that has the dual channel.

Definition 6.4. Let h be a heap, $\overline{\mathtt{o}}$ be a queue of objects in h and $\mathtt{t}, \mathtt{t}'$ be two session types. The relation $\mathtt{t} \ltimes^{\overline{\mathtt{o}}}_{h} \mathtt{t}'$ is defined by:

(1) $\mathtt{t} \ltimes^{()}_{h} \mathtt{t}'$ if $\mathtt{t} \bowtie \mathtt{t}'$.

(2) $\mathtt{t}_i.\mathtt{t}' \bowtie_h^{\overline{\mathtt{o}}::\mathtt{o}} \mathtt{t}''$ for $i \in \{1,2\}$ if

$$!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}' \bowtie_h^{\overline{\mathtt{o}}} \mathtt{t}''$$
$$h(\mathtt{o}) = (\mathtt{C}, \_)$$

and

$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i.$$

Intuitively, this definition describes an agreement between the session type $\mathtt{t}$ of a channel $\mathtt{k}$ and the session type $\mathtt{t}'$ of $\tilde{\mathtt{k}}$ after the objects $\overline{\mathtt{o}}$ have been put in the queue associated with $\tilde{\mathtt{k}}$ in h (recall that communication is asynchronous and that only one of the queues $h(\mathtt{k})$ and $h(\tilde{\mathtt{k}})$ can be non-empty). Thus, when the queue is empty (case (1) of the definition), $\mathtt{t}'$ and $\mathtt{t}$ agree if they are dual. When the queue is $\overline{\mathtt{o}} :: \mathtt{o}_i$ (case (2)), if the session type $\mathtt{t}''$ agrees with

$$!\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$$

after the objects $\overline{\mathtt{o}}$ have been put in the queue, then it also agrees with the type $\mathtt{t}_i.\mathtt{t}'$, where $\mathtt{t}_i$ is the session type of the branch obtained after putting the object $\mathtt{o}_i$ in the queue. For instance, $\mathtt{t}''$ agrees with $\mathtt{t}_1.\mathtt{t}'$ through the queue `"a"::true::3` (written $\mathtt{t}_1.\mathtt{t}' \bowtie_h^{\texttt{"a"::true::3}} \mathtt{t}''$) if it agrees with

$$!\{\mathtt{Int} \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{Object} \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$$

through the queue `"a"::true`. Indeed, after branch selection (the sent value 3 is an `Int`), the continuation of

$$!\{\mathtt{Int} \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{Object} \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$$

is $\mathtt{t}_1.\mathtt{t}'$.

The main lemma concerning the above relation says that if the type $\mathtt{t}$ of a channel $\mathtt{k}$ agrees with the type $?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$ of $\tilde{\mathtt{k}}$ when h maps $\tilde{\mathtt{k}}$ to the queue $\mathtt{o} :: \overline{\mathtt{o}}$, and $\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$, where $i \in \{1,2\}$ and $\mathtt{C}$ is the class of $\mathtt{o}$ in h, then $\mathtt{t}$ agrees with $\mathtt{t}_i.\mathtt{t}'$ when h maps $\tilde{\mathtt{k}}$ to the queue $\overline{\mathtt{o}}$.

Lemma 6.5. If

$$\mathtt{t} \bowtie_h^{\mathtt{o}::\overline{\mathtt{o}}} ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$$

and

$$h(\mathtt{o}) = (\mathtt{C}, \_)$$
$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i,$$

then $\mathtt{t} \bowtie_h^{\overline{\mathtt{o}}} \mathtt{t}_i.\mathtt{t}'$.

Proof. The proof is by induction on the length of $\overline{\mathtt{o}}$.

In the base case $\overline{\mathtt{o}} = ( )$, the relation

$$\mathtt{t} \bowtie_h^{\mathtt{o}} ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'$$

can only have been obtained by Definition 6.4(2). So we have $\mathtt{t} = \mathtt{t}'_j.\mathtt{t}^*$ for some $\mathtt{t}'_j$ ($j \in \{1,2\}$) and $\mathtt{t}^*$, and

$$\mathtt{C} \Downarrow \{\mathtt{C}'_1, \mathtt{C}'_2\} = \mathtt{C}_j$$

and

$$!\{\mathtt{C}'_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{t}'_2\}.\mathtt{t}^* \leftthreetimes_h^{()} ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'.$$

By Definition 6.4(1), we get

$$!\{\mathtt{C}'_1 \Rightarrow \mathtt{t}'_1 \,[]\, \mathtt{C}'_2 \Rightarrow \mathtt{t}'_2\}.\mathtt{t}^* \bowtie ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'.$$

From

$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$$
$$\mathtt{C} \Downarrow \{\mathtt{C}'_1, \mathtt{C}'_2\} = \mathtt{C}_j,$$

we can derive either

$$\mathtt{C}_i \Downarrow \{\mathtt{C}'_1, \mathtt{C}'_2\} = \mathtt{C}_j$$

or

$$\mathtt{C}_j \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i,$$

which implies $\mathtt{t}_i \bowtie \mathtt{t}'_j$ and $\mathtt{t}^* \bowtie \mathtt{t}'$ by the definition of duality. Therefore, we can conclude $\mathtt{t}_i.\mathtt{t}^* \bowtie \mathtt{t}'_j.\mathtt{t}'$, which gives $\mathtt{t}_i.\mathtt{t}^* \leftthreetimes_h^{()} \mathtt{t}'_j.\mathtt{t}'$ by Definition 6.4(1).

For the induction case, we assume

$$\overline{\mathtt{o}} = \overline{\mathtt{o}'} :: \mathtt{o}^+.$$

So the hypothesis becomes

$$\mathtt{t} \leftthreetimes_h^{\mathtt{o}::\overline{\mathtt{o}'}::\mathtt{o}^+} ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'.$$

This relation can only have been obtained from Definition 6.4(2). So we have $\mathtt{t} = \mathtt{t}^+_j.\mathtt{t}''$ for some $\mathtt{t}^+_j$ ($j \in \{1,2\}$) and $\mathtt{t}''$, and

$$h(\mathtt{o}^+) = (\mathtt{C}^+, \_)$$
$$\mathtt{C}^+ \Downarrow \{\mathtt{C}^+_1, \mathtt{C}^+_2\} = \mathtt{C}^+_j$$

and

$$!\{\mathtt{C}^+_1 \Rightarrow \mathtt{C}^+_2 \,[]\, \mathtt{t}^+_1 \Rightarrow \mathtt{t}^+_2\}.\mathtt{t}'' \leftthreetimes_h^{\mathtt{o}::\overline{\mathtt{o}'}} ?\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}'.$$

By the induction hypothesis, we have

$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_i$$

and

$$!\{\mathtt{C}^+_1 \Rightarrow \mathtt{C}^+_2 \,[]\, \mathtt{t}^+_1 \Rightarrow \mathtt{t}^+_2\}.\mathtt{t}'' \leftthreetimes_h^{\overline{\mathtt{o}'}} \mathtt{t}_i.\mathtt{t}',$$

so applying Definition 6.4(2) again, we get the result. □

We will now extend the definition of *agreement* to session environments.

Definition 6.5.

(1) The predicate $ag(\Sigma; h)$ is defined by

$$ag(\Sigma; h) \quad \text{if} \quad \begin{cases} \mathtt{k} \in \mathsf{dom}(\Sigma) \;\Leftrightarrow\; \mathtt{k} \in \mathsf{dom}(h), \\ \forall \mathtt{k} \in \mathsf{dom}(\Sigma) : h(\mathtt{k}) = (\,) \;\Rightarrow\; \Sigma(\mathtt{k}) \ltimes_h^{h(\tilde{\mathtt{k}})} \Sigma(\tilde{\mathtt{k}}). \end{cases}$$

(2) $ag(\Gamma; \Sigma; h)$ if $ag(\Gamma; h)$ and $ag(\Sigma; h)$.

We then say a session environment and a heap agree if:

— the same set of channels occurs in the session environment and in the heap;
— when the queue of a channel $\mathtt{k}$ is empty, the queue of $\tilde{\mathtt{k}}$ relates the session type of $\mathtt{k}$ to the session type of $\tilde{\mathtt{k}}$.

A term environment, a session environment and a heap agree if both the heap with the standard environment and the heap with the session environment agree.

The following key lemma generalises Theorem 6.1 by asserting that the above agreement is preserved under reduction of well-typed parallel threads.

Lemma 6.6 (subject reduction generalisation). We let $\Gamma \vdash_{\mathtt{r}} \mathtt{e}_i : \mathtt{T}_i \fatsemi \Sigma_i$, $(1 \leqslant i \leqslant n)$ and assume $ag(\Gamma; \Sigma; h)$ where $\Sigma = \bigcup_{1 \leqslant i \leqslant n} \Sigma_i$. Then, if

$$\mathtt{e}_1 \parallel \ldots \parallel \mathtt{e}_n, h \longrightarrow \mathtt{e}'_1 \parallel \ldots \parallel \mathtt{e}'_{n'}, h' \quad \text{where} \quad 1 \leqslant n \leqslant n',$$

there exist Γ' and Σ'_i such that

$$\Gamma' \vdash_{\mathtt{r}} \mathtt{e}'_i : \mathtt{T}_i \fatsemi \Sigma'_i \quad (1 \leqslant i \leqslant n')$$

and

$$ag(\Gamma'; \Sigma'; h'),$$

where

$$\Sigma' = \bigcup_{1 \leqslant i \leqslant n'} \Sigma'_i.$$

Proof. We have that for some i $(1 \leqslant i \leqslant n)$, either $\mathtt{e}_i, h \longrightarrow \mathtt{e}'_i \parallel \mathtt{e}''_i, h'$ by an application of rule SessReq-R or $\mathtt{e}_i, h \longrightarrow \mathtt{e}'_i, h'$ by the application of any one of the other reduction rules. In the first case, the proof follows immediately from Theorem 6.1 (2) and Definition 6.5.

So we let $\mathtt{e}_i, h \longrightarrow \mathtt{e}'_i, h'$. If this reduction has not been obtained by a communication rule, the proof is trivial by Theorem 6.1 (1). The interesting cases are when the reduction $\mathtt{e}_i, h \longrightarrow \mathtt{e}'_i, h'$ is obtained by a communication rule. By Theorem 6.1, we immediately obtain $\Gamma' \vdash_{\mathtt{r}} \mathtt{e}'_i : \mathtt{T}_i \fatsemi \Sigma'_i$ and $ag(\Gamma'; h')$, so we only have to show $ag(\Sigma'; h')$, which implies $ag(\Gamma'; \Sigma'; h')$.

— SendCase-R:
Assume

$$\mathtt{e}_i = \mathscr{E}[\mathtt{k.sendC(o)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}''_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{e}''_2\}].$$

We have

$$\mathscr{E}[\mathtt{k.sendC(o)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}''_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{e}''_2\}],\ h \quad \longrightarrow \quad \mathscr{E}[\mathtt{e}''_j],\ h' \quad \text{with } j \in \{1, 2\}$$

where

$$h(\tilde{\mathtt{k}}) = \overline{\mathtt{o}}$$
$$h(\mathtt{o}) = (\mathtt{C}, \_)$$
$$h' = h[\tilde{\mathtt{k}} \mapsto \overline{\mathtt{o}} :: \mathtt{o}]$$
$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_j.$$

Since

$$\Gamma \vdash_{\mathtt{T}} \mathtt{e}_i : \mathtt{T}_i \, ; \, \Sigma_i,$$

by the proof of the same case in Theorem 6.1, we get

$$\Sigma_i = \{\mathtt{k} :! \{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}.\Sigma_i''$$

and

$$\Gamma \vdash_{\mathtt{T}} \mathscr{E}[\mathtt{e}_j''] : \mathtt{T}_{\S} \, \Sigma_i'$$

where

$$\Sigma_i' = \{\mathtt{k} : \mathtt{t}_j\}.\Sigma_i'' \quad \text{for} \quad j \in \{1, 2\}.$$

So we can derive

$$\Sigma'(\mathtt{k}) \ltimes_{h'}^{h'(\tilde{\mathtt{k}})} \Sigma'(\tilde{\mathtt{k}})$$

from

$$\Sigma(\mathtt{k}) \ltimes_{h}^{h(\tilde{\mathtt{k}})} \Sigma(\tilde{\mathtt{k}})$$

by Definition 6.4 (2)), and then conclude $ag(\Sigma'; h')$.

— ReceiveCase-R:

Assume

$$\mathtt{e}_i = \mathscr{E}[\mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1'' \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2''\}].$$

We have

$$\mathscr{E}[\mathtt{k.recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1'' \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2''\}], \; h \quad \longrightarrow \quad \mathscr{E}[[\mathtt{o}/\mathtt{x}]\mathtt{e}_j''], \; h' \quad \text{with } j \in \{1, 2\}$$

where

$$h(\mathtt{k}) = \mathtt{o} :: \overline{\mathtt{o}}$$
$$h(\mathtt{o}) = (\mathtt{C}, \_)$$
$$h' = h[\mathtt{k} \mapsto \overline{\mathtt{o}}]$$
$$\mathtt{C} \Downarrow \{\mathtt{C}_1, \mathtt{C}_2\} = \mathtt{C}_j.$$

Since $\Gamma \vdash_{\mathtt{T}} \mathtt{e}_i : \mathtt{T}_i \, ; \, \Sigma_i$, by the proof of the same case in Theorem 6.1, we get

$$\Sigma_i = \{\mathtt{k} :? \{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}\}.\Sigma_i''$$

and

$$\Gamma \vdash_{\mathtt{T}} \mathscr{E}[[\mathtt{o}/\mathtt{x}]\mathtt{e}_j''] : \mathtt{T}_{\S} \, \Sigma_i'$$

where

$$\Sigma_i' = \{\mathtt{k} : \mathtt{t}_j\}.\Sigma_i'' \quad \text{for} \quad j \in \{1, 2\}.$$

So we can derive

$$\Sigma'(\tilde{\mathtt{k}}) \ltimes_{h'}^{h'(\mathtt{k})} \Sigma'(\mathtt{k})$$

from

$$\Sigma(\tilde{\mathtt{k}}) \ltimes_{h}^{h(\mathtt{k})} \Sigma(\mathtt{k})$$

by Lemma 6.5, and then conclude $ag(\Sigma'; h')$. □

It is convenient to take into account the order in which communication and delegation redexes (see Section 4) occurring in the same expression are reduced. To this end, we introduce the 'follows' relation between redexes.

Definition 6.6. Let $\mathtt{e}$ be an expression and $\mathtt{r}_1, \mathtt{r}_2$ be two different occurrences of communication or delegation redexes in $\mathtt{e}$. We say that $\mathtt{r}_2$ *follows* $\mathtt{r}_1$ in $\mathtt{e}$ if there is a subexpression $\mathtt{e}'$ of $\mathtt{e}$ such that $\mathtt{e}' = \mathscr{E}[\mathtt{r}_1]$ and $\mathtt{r}_2$ occurs in $\mathtt{e}'$.

Note that by the definition of the evaluation context, $\mathtt{r}_1$ cannot be a subexpression of $\mathtt{r}_2$, but $\mathtt{r}_2$ can be a subexpression of $\mathtt{r}_1$.

It is easy to check that, if $\mathtt{r}_1$ and $\mathtt{r}_2$ are as in the previous definition, then $\mathtt{r}_1$ needs to be reduced before $\mathtt{r}_2$, since $\mathtt{r}_1$ occurs in the hole of an evaluation context $\mathscr{E}$, while $\mathtt{r}_2$ occurs elsewhere in the same expression.

By convention, we assume that all fresh channels created when reducing parallel threads take successive indexes according to the order of creation, that is, they are named $\mathtt{k}_0$, $\mathtt{k}_1, \ldots$. This means that if

$$P, h \longrightarrow^* Q, h' \longrightarrow^* Q', h''$$

and $\mathtt{k}_i$ is a channel created in the reduction $P, h \longrightarrow^* Q, h'$, and $\mathtt{k}_j$ is a channel created in the reduction $Q, h' \longrightarrow^* Q', h''$, then $i < j$.

The *subject* of a communication or delegation redex is the channel specified in its syntax on which the communication takes place. The *index of a communication or delegation redex* is the index of its subject.

The following crucial lemma states that a channel and its dual cannot occur in the same thread. Moreover, it states that the order of the indexes of the communication and delegation redexes agrees with the 'follows' relation between redexes.

Lemma 6.7. Let $\mathtt{e}$ be an initial expression and $\mathtt{e}, [\,] \longrightarrow^* \mathtt{e}_1 \parallel \ldots \parallel \mathtt{e}_n, h$. Then:

(1) No expression $\mathtt{e}_i$ can contain occurrences of both $\mathtt{k}$ and $\tilde{\mathtt{k}}$ for some channel $\mathtt{k}$.
(2) If $\mathtt{r}_1, \mathtt{r}_2$ are two different occurrences of communication or delegation redexes in $\mathtt{e}_i$ ($i \in \{1, \ldots, n\}$) and $\mathtt{r}_2$ follows $\mathtt{r}_1$, then the index of $\mathtt{r}_1$ is greater than or equal to the index of $\mathtt{r}_2$.

Proof.

(1) This part follows straightforwardly by noting that the channels $\mathtt{k}$ and $\tilde{\mathtt{k}}$ are introduced by the rule SessReq-R in two different parallel threads.
(2) $\varnothing \vdash \mathtt{e} : \mathtt{T} \fatsemi \varepsilon$ implies that no channel occurs in $\mathtt{e}$, so the property holds trivially. We now prove that the reduction preserves the property: that is, if all the channels in the subexpressions of an expression are indexed in a non-increasing order in the sense of

Definition 6.6, starting from the redex to all the following redexes, then after one step of reduction, we get expressions that have the same property. The proof is by case analysis on the definition of $\longrightarrow$:

— SessReq-R:
We have

$$\frac{h(\mathtt{o}) = (\mathtt{C}, \_) \qquad sbody(\mathtt{s}, \mathtt{C}) = \mathtt{e}'' \qquad \mathtt{k}, \tilde{\mathtt{k}} \notin h}{\mathscr{E}[\mathtt{o.s}\,\{\mathtt{e}'\}], h \longrightarrow \mathscr{E}[\mathtt{e}' \wr \mathtt{k} \int] \parallel [\mathtt{o}/\mathtt{this}]\mathtt{e}'' \wr \tilde{\mathtt{k}} \int, h[\mathtt{k}, \tilde{\mathtt{k}} \mapsto ()]}\,.$$

Let $\mathscr{E}[\mathtt{o.s}\,\{\mathtt{e}'\}]$ be an expression in which the desired property holds. After one step of reduction, the new channel $\mathtt{k}$ in the expression $\mathtt{e}' \wr \mathtt{k} \int$ is the one with the highest index and no other channel occurs in it. Moreover, all communication and delegation redexes occurring in $\mathscr{E}$ follow all communication and delegation redexes in $\mathtt{e}' \wr \mathtt{k} \int$. Finally, note that by the induction hypothesis, the desired property holds for all communication and delegation redexes occurring in $\mathscr{E}$.
In parallel, we have the expression $[\mathtt{o}/\mathtt{this}]\mathtt{e}'' \wr \tilde{\mathtt{k}} \int$, where $\mathtt{e}''$ is a session body, so the only channel in this expression is $\tilde{\mathtt{k}}$. Hence, this reduction rule preserves the property.

— SessDel-R:
We have

$$\frac{h(\mathtt{o}) = (\mathtt{C}, \_) \qquad sbody(\mathtt{s}, \mathtt{C}) = \mathtt{e}'}{\mathscr{E}[\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}], h \longrightarrow \mathscr{E}[[\mathtt{o}/\mathtt{this}]\mathtt{e}' \wr \mathtt{k} \int], h}\,.$$

Let $\mathscr{E}[\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}]$ be an expression in which the desired property holds. Since $\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}$ is the redex, $\mathtt{k}$ is the channel with the highest index. After one step of reduction, the next expression to be reduced is $[\mathtt{o}/\mathtt{this}]\mathtt{e}' \wr \mathtt{k} \int$, and $\mathtt{k}$ is still the only channel that occurs in it.

— SendCase-R:
We have

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

If the expression

$$\mathscr{E}[\mathtt{k.sendC(o)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}'_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}'_2\}]$$

is an expression in which the desired property holds, then $\mathtt{k}$ is the channel with the highest index. The channel $\mathtt{k}$ is the only channel occurring in the expressions $\mathtt{e}'_1, \mathtt{e}'_2$. So, after one step of reduction, the expression $\mathtt{e}'_j$ can either contain only the channel $\mathtt{k}$, which is the one with the highest index, or it can contain no channel, so the property still holds.

— ReceiveCase-R, SendWhile-R and ReceiveWhile-R:
The proof in these cases is similar to the previous one.

In all the remaining cases, no channel is introduced or modified, so the property is trivially preserved. □

The above lemma is a technical step in proving the *deadlock freedom* property for communication expressions. Indeed, it is easy to verify that well-typed sending redexes always reduce, as well as while-receiving redexes. So the crucial case is when we obtain a parallel composition of case-receiving redexes, and in the following lemma, we prove that these receiving actions are not stuck since their expectations match the values on the channel queue.

Lemma 6.8 (deadlock freedom). We let $\mathtt{e}$ be an initial expression and assume

$$\mathtt{e}, [\,] \longrightarrow^{*} \mathtt{o}_1 \parallel \ldots \parallel \mathtt{o}_m \parallel \mathtt{e}_1 \parallel \ldots \parallel \mathtt{e}_n, h,$$

such that $m \geqslant 0$ and that for all i $(1 \leqslant i \leqslant n)$, we have $\mathtt{e}_i = \mathscr{E}_i[\mathtt{r}_i]$, where $\mathscr{E}_i$ is an evaluation context and $\mathtt{r}_i$ is a case-receiving redex. Then there is i $(1 \leqslant i \leqslant n)$ such that $\mathtt{e}_i, h \longrightarrow P, h'$ for some P, h'.

Proof. By Corollary 6.1, each $\mathtt{e}_i$ is well typed from a term environment Γ that agrees with h.

Let j be the highest of the indexes of the channels occurring in $\mathtt{e}_1 \mid \ldots \mid \mathtt{e}_n$.

If both $\mathtt{k}_j$ and $\tilde{\mathtt{k}}_j$ occur in $\mathtt{e}_1 \mid \ldots \mid \mathtt{e}_n$, then, by Lemma 6.7 (1), they occur in two different expressions: let them be $\mathtt{e}_p$ and $\mathtt{e}_q$ with $1 \leqslant p \neq q \leqslant n$. By Lemma 6.7 (2), the subjects of the two redexes $\mathtt{r}_p$ and $\mathtt{r}_q$ are the channels $\mathtt{k}_j$ and $\tilde{\mathtt{k}}_j$. Moreover, we must have that $\Sigma_p(\mathtt{k}_j)$, $\Sigma_q(\tilde{\mathtt{k}}_j)$ are of the forms

$$?\{\mathtt{D}_1 \Rightarrow \mathtt{t}_1 \,[\!]\, \mathtt{D}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t},$$
$$?\{\mathtt{D}'_1 \Rightarrow \mathtt{t}'_1 \,[\!]\, \mathtt{D}'_2 \Rightarrow \mathtt{t}'_2\}.\mathtt{t}'$$

since $\mathtt{r}_p$ and $\mathtt{r}_q$ are case-receiving redexes. If $h(\mathtt{k}_j)$ is not empty, we let $h(\mathtt{k}_j) = \mathtt{o} :: \overline{\mathtt{o}'}$. By Lemma 6.5, $h(\mathtt{o}) = (\mathtt{C}, \_)$ and $\mathtt{C} \Downarrow \{\mathtt{D}_1, \mathtt{D}_2\}$ is defined, so $\mathtt{r}_p$ can perform a ReceiveCase-R step, which is contrary to the hypothesis. We can reason similarly if $h(\tilde{\mathtt{k}}_j)$ is not empty. Otherwise, if both $h(\mathtt{k}_j)$ and $h(\tilde{\mathtt{k}}_j)$ are empty, then by Lemma 6.6, we get $ag(\Sigma; h)$, where Σ is the session environment of $\mathtt{e}_1 \mid \ldots \mid \mathtt{e}_n$. This implies $\Sigma(\mathtt{k}) \ltimes_h^{()} \Sigma(\tilde{\mathtt{k}})$ by Definition 6.5 and thus $\Sigma_q(\tilde{\mathtt{k}}_j) \bowtie \Sigma_p(\mathtt{k}_j)$ by Definition 6.4 (1). But this is impossible since $\Sigma_p(\mathtt{k}_j)$ and $\Sigma_q(\tilde{\mathtt{k}}_j)$ have the forms

$$?\{\mathtt{D}_1 \Rightarrow \mathtt{t}_1 \,[\!]\, \mathtt{D}_2 \Rightarrow \mathtt{t}_2\}.\mathtt{t}$$
$$?\{\mathtt{D}'_1 \Rightarrow \mathtt{t}'_1 \,[\!]\, \mathtt{D}'_2 \Rightarrow \mathtt{t}'_2\}.\mathtt{t}'.$$

If only $\mathtt{k}_j$ occurs in $\mathtt{e}_1 \mid \ldots \mid \mathtt{e}_n$, we must have

$$\Sigma(\mathtt{k}_j) \neq \varepsilon$$
$$\Sigma(\tilde{\mathtt{k}}_j) = \varepsilon.$$

From $ag(\Sigma; h)$, by Definition 6.5, we get that $\Sigma(\tilde{\mathtt{k}}_j) = \varepsilon$ implies $h(\tilde{\mathtt{k}}_j) = (\,)$, and thus $\varepsilon \ltimes_h^{h(\mathtt{k}_j)} \Sigma(\mathtt{k}_j)$. We conclude that $h(\mathtt{k}_j)$ is not empty, so we can proceed as before. □

Theorem 6.2 (type safety). If $\mathtt{e}$ is an initial expression and $\mathtt{e}, [] \longrightarrow^* \mathtt{e}_1 \parallel \ldots \parallel \mathtt{e}_n, h$, then one of the following conditions holds:

— There is i $(1 \leqslant i \leqslant n)$, such that $\mathtt{e}_i, h \longrightarrow P, h'$ for some P, h'.
— $\mathtt{e}_i$ is an object for all i $(1 \leqslant i \leqslant n)$.

Proof. By Proposition 6.1, $\mathtt{e}$ is closed and channel-complete, so by Proposition 4.2, each $\mathtt{e}_i$ is closed and channel-complete. Therefore, by Proposition 4.1, either $\mathtt{e}_i$ is an object identifier or $\mathtt{e}_i = \mathscr{E}_i[\mathtt{r}_i]$ for some evaluation context $\mathscr{E}_i$ and some redex $\mathtt{r}_i$. If $\mathtt{e}_i = \mathscr{E}_i[\mathtt{r}_i]$, then, by Corollary 6.1, $\mathtt{e}_i$ can be typed from a term environment Γ that agrees with h, so, by Lemma 6.2 (1), we have $\mathtt{r}_i$ can be typed from Γ too.

If some $\mathtt{r}_i$ is of one of the shapes

$$\mathtt{o}\,;\mathtt{e}'$$
$$\mathtt{new}\ \mathtt{C}(\overline{\mathtt{o}})$$
$$\mathtt{k.sendC(o)}\{\mathtt{C} \Rightarrow \mathtt{e} \,[]\, \mathtt{C} \Rightarrow \mathtt{e}\}$$
$$\mathtt{k.sendW(e)}\{\mathtt{C} \Rightarrow \mathtt{e} \,[]\, \mathtt{C} \Rightarrow \mathtt{e}\}$$
$$\mathtt{k.recvW(x)}\{\mathtt{C} \Rightarrow \mathtt{e} \,[]\, \mathtt{C} \Rightarrow \mathtt{e}\},$$

we can immediately verify that $\mathtt{e}_i$ reduces.

Otherwise, let some $\mathtt{r}_i$ be of one of the shapes

$$\mathtt{o.f}$$
$$\mathtt{o.f} := \mathtt{o}'$$
$$\mathtt{o.s}\,\{\mathtt{e}'\}$$
$$\mathtt{o} \bullet \mathtt{s}\,\{\mathtt{k}\}.$$

Since an object identifier cannot occur in an initial expression, the run-time expression $\mathtt{o}$ has been obtained by reducing $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{o}})$ for some $\mathtt{C}$, $\overline{\mathtt{f} : \mathtt{o}}$, which implies $h(\mathtt{o}) = (\mathtt{C}, \overline{\mathtt{f} : \mathtt{o}})$ by rule (NewC-R). By Definition 6.3 (2), this implies $\Gamma(\mathtt{o}) = \mathtt{C}$. If $\mathtt{r}_i = \mathtt{o.f}$, rule (FLD-RT) has been applied with a premise $\Gamma \vdash_{\overline{\mathtt{r}}} \mathtt{o} : \mathtt{T} \fatsemi \varnothing$ for some $\mathtt{T}$ such that $\mathtt{C} <: \mathtt{T}$ and $\mathtt{f} \in \mathit{fields}(\mathtt{T})$, so $\mathtt{f} \in \mathit{fields}(\mathtt{C})$. If $\mathtt{r}_i = \mathtt{o.s}\{\ldots\}$, rule (SESSREQ-RT) has been applied with a premise $\Gamma \vdash_{\overline{\mathtt{r}}} \mathtt{o} : \mathtt{T} \fatsemi \varnothing$ for some $\mathtt{T}$ such that $\mathtt{C} <: \mathtt{T}$ and $\mathit{stype}(\mathtt{s}, \mathtt{T})$ is defined. Therefore, $\mathit{stype}(\mathtt{s}, \mathtt{C})$ is also defined, so $\mathit{sbody}(\mathtt{s}, \mathtt{C})$ is defined. We can similarly show that $\mathit{sbody}(\mathtt{s}, \mathtt{C})$ is defined when $\mathtt{r}_i = \mathtt{o} \bullet \mathtt{s}\{\ldots\}$. Hence, we can conclude that $\mathtt{e}_i$ reduces in all the above cases.

The only remaining alternative, which is when all $\mathtt{r}_i$ are case-receiving redexes, follows from Lemma 6.8. □

7. Session type reconstruction

The type system presented in Figure 9 derives session types for expressions assuming that all session declarations are decorated with explicit session types. Moreover, because of the subsumption rule, expressions can have many types. In this section we present an inference algorithm (Figure 13) that:

(i) gives an expression its minimal type;

$$\textsc{Axiom-T-I}\quad \Gamma \vdash_{\mathrm{I}} \mathtt{z} : \Gamma(\mathtt{z}) \,;\, \varepsilon \rhd \varnothing, \varnothing \qquad \textsc{Cont-T-I}\quad \Gamma \vdash_{\mathrm{I}} \mathtt{cont}^{\mathtt{T}} : \mathtt{T} \,;\, \circledcirc \rhd \varnothing, \varnothing$$

$$\textsc{NewC-T-I}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\,\mathtt{f}} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}'_i \,;\, \varepsilon \rhd \mathscr{C}_i, \mathscr{D}_i \qquad \mathtt{T}'_i <: \mathtt{T}_i}{\Gamma \vdash_{\mathrm{I}} \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C} \,;\, \varepsilon \rhd \bigcup_i \mathscr{C}_i, \bigcup_i \mathscr{D}_i} \qquad \textsc{Fld-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \,;\, \theta \rhd \mathscr{C}, \mathscr{D}}{\Gamma \vdash_{\mathrm{I}} \mathtt{e.f} : \mathit{ftype}_r(\mathtt{f}, \mathtt{T}) \,;\, \theta \rhd \mathscr{C}, \mathscr{D}}$$

$$\textsc{Seq-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \,;\, \theta \rhd \mathscr{C}, \mathscr{D} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}' : \mathtt{T}' \,;\, \theta' \rhd \mathscr{C}', \mathscr{D}'}{\Gamma \vdash_{\mathrm{I}} \mathtt{e;e}' : \mathtt{T}' \,;\, \theta.\theta' \rhd \mathscr{C} \cup \mathscr{C}', \mathscr{D} \cup \mathscr{D}'}$$

$$\textsc{FldAss-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \,;\, \theta \rhd \mathscr{C}, \mathscr{D} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}' : \mathtt{T}' \,;\, \theta' \rhd \mathscr{C}', \mathscr{D}' \qquad \mathtt{T}' <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T})}{\Gamma \vdash_{\mathrm{I}} \mathtt{e.f} := \mathtt{e}' : \mathit{ftype}_r(\mathtt{f}, \mathtt{T}) \,;\, \theta.\theta' \rhd \mathscr{C} \cup \mathscr{C}', \mathscr{D} \cup \mathscr{D}'}$$

$$\textsc{SessReq-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{C}_1 \vee \cdots \vee \mathtt{C}_n \,;\, \theta \rhd \mathscr{C}, \mathscr{D} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}' : \mathtt{T}' \,;\, \theta' \rhd \mathscr{C}', \mathscr{D}' \qquad \mathscr{D}'' = \mathscr{D} \cup \mathscr{D}' \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_i} \bowtie \theta' | i \in \{1, \ldots, n\}\}}{\Gamma \vdash_{\mathrm{I}} \mathtt{e.s}\{\mathtt{e}'\} : \mathtt{T}' \,;\, \theta \rhd \mathscr{C} \cup \mathscr{C}', \mathscr{D}''}$$

$$\textsc{SessDel-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{C}_1 \vee \cdots \vee \mathtt{C}_n \,;\, \theta \rhd \mathscr{C}, \mathscr{D} \qquad \mathit{rtype}(\mathtt{s}, \mathtt{C}_1 \vee \cdots \vee \mathtt{C}_n) = \mathtt{T} \qquad \mathscr{C}' = \mathscr{C} \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_1} \neq \varepsilon\} \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_i} = \chi^{\mathtt{s}}_{\mathtt{C}_j} | i \neq j \in \{1, \ldots, n\}\}}{\Gamma \vdash_{\mathrm{I}} \mathtt{e} \bullet \mathtt{s}\,\{\} : \mathtt{T} \,;\, \theta.\chi^{\mathtt{s}}_{\mathtt{C}_1} \rhd \mathscr{C}', \mathscr{D}}$$

$$\textsc{SendC-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \,;\, \varepsilon \rhd \mathscr{C}, \mathscr{D} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}_i \,;\, \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i \qquad \mathtt{T} <: \mathtt{C}_1 \vee \mathtt{C}_2}{\Gamma \vdash_{\mathrm{I}} \mathtt{sendC(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \,;\, !\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\} \rhd \mathscr{C} \cup \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D} \cup \mathscr{D}_1 \cup \mathscr{D}_2}$$

$$\textsc{ReceiveC-T-I}\quad \frac{\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}_i \,;\, \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i}{\Gamma \vdash_{\mathrm{I}} \mathtt{recvC(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \,;\, ?\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\} \rhd \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D}_1 \cup \mathscr{D}_2}$$

$$\textsc{SendW-T-I}\quad \frac{\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \,;\, \varepsilon \rhd \mathscr{C}, \mathscr{D} \qquad \Gamma \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}_i \,;\, \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i \qquad \alpha \text{ fresh in } \theta_1, \theta_2 \qquad \mathtt{T}_1 \vee \mathtt{T}_2 <: \mathtt{T}' \qquad \forall \mathtt{T}' \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2) \qquad \mathtt{T} <: \mathtt{C}_1 \vee \mathtt{C}_2}{\Gamma \vdash_{\mathrm{I}} \mathtt{sendW(e)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \,;\, \mu\alpha.!\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\theta_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\theta_2\} \rhd \mathscr{C} \cup \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D} \cup \mathscr{D}_1 \cup \mathscr{D}_2}$$

$$\textsc{ReceiveW-T-I}\quad \frac{\Gamma, \mathtt{x} : \mathtt{C}_i \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}_i \,;\, \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i \qquad \alpha \text{ fresh in } \theta_1, \theta_2 \qquad \mathtt{T}_1 \vee \mathtt{T}_2 <: \mathtt{T} \qquad \forall \mathtt{T} \in \mathit{tc}(\mathtt{e}_1) \cup \mathit{tc}(\mathtt{e}_2)}{\Gamma \vdash_{\mathrm{I}} \mathtt{recvW(x)}\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \,;\, \mu\alpha.?\{\mathtt{C}_1 \Rightarrow [\alpha/\circledcirc]\theta_1 \,[]\, \mathtt{C}_2 \Rightarrow [\alpha/\circledcirc]\theta_2\} \rhd \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D}_1 \cup \mathscr{D}_2}$$

Fig. 13. Constraint-based typing rules for channel-free expressions.

(ii) calculates the constraints that must be satisfied in order to reconstruct the related session type (which is unique, as stated in Proposition 5.1).

This means that programmers need no longer be responsible for declaring the session types.

We define an *inference class table ICT* as a class table in which each session declaration $\mathtt{s}$, in each class $\mathtt{C}$, is decorated by the *session-in-class variable* $\chi^{\mathtt{s}}_{\mathtt{C}}$ representing the session type that will be inferred by the algorithm.

Then we extend the syntax of session types to *session type schemes* in order to include session-in-class variables:

$$\theta ::= \mathtt{t} \mid \chi^{\mathtt{s}}_{\mathtt{C}} \mid \theta.\theta \mid \dagger\{\mathtt{C}_1 \Rightarrow \theta \,[]\, \mathtt{C}_2 \Rightarrow \theta\} \;\mid\; \mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \theta \,[]\, \mathtt{C}_2 \Rightarrow \theta\}\,.$$

$$\textsc{Sess-WF-I}\quad \frac{\{\texttt{this} : \texttt{C}\} \vdash_{\mathrm{I}} \texttt{e} : \texttt{T} \fatsemi \theta \rhd \mathscr{C}', \mathscr{D} \qquad \mathscr{C} = \mathscr{C}' \cup \{\chi^{\texttt{s}}_{\texttt{C}} = \theta\} \qquad \circledcirc \text{ does not appear in } \theta}{\texttt{T}\,\chi^{\texttt{s}}_{\texttt{C}}\,\texttt{s}\,\{\ \texttt{e}\ \}\ \texttt{ok in C with}\ \mathscr{C}, \mathscr{D}}$$

$$\textsc{Class-WF-I}\quad \frac{\texttt{D ok} \qquad \texttt{S}_i\ \texttt{ok in C with}\ \mathscr{C}_i, \mathscr{D}_i}{\texttt{class C} \lhd \texttt{D}\ \{\ \overline{\texttt{Tf}};\ \overline{\texttt{S}}\ \}\ \texttt{ok with}\ \bigcup_i \mathscr{C}_i, \bigcup_i \mathscr{D}_i}$$

Fig. 14. Well-formed inference class tables.

If CT is a class table, we use CT^{-} to denote the inference class table obtained by replacing in CT the declared session type of any session $\texttt{s}$ in any class $\texttt{C}$ by $\chi^{\texttt{s}}_{\texttt{C}}$.

In order to reconstruct the session types of session declarations, we use two kinds of constraints:

— A set of *equality* (and *disequality*) constraints, denoted by $\mathscr{C}$, will collect assertions of the form $\chi^{\texttt{s}}_{\texttt{C}} = \theta$ and $\chi^{\texttt{s}}_{\texttt{C}} \neq \varepsilon$.

— A set of *duality* constraints, denoted by $\mathscr{D}$, will collect assertions of the form $\chi^{\texttt{s}}_{\texttt{C}} \bowtie \theta$.

The constraint-based type inference system is presented in Figure 13. Note that if a session-in-class variable $\chi^{\texttt{s}}_{\texttt{C}}$ occurs in a session type that is inferred for an expression, then $\chi^{\texttt{s}}_{\texttt{C}}$ has been introduced by rule SessDel-T-I, so the related set of constraints must contain $\chi^{\texttt{s}}_{\texttt{C}} \neq \varepsilon$. This means that no derived session type can be equated to ε by a substitution that satisfies the set of constraints. For this reason, when required, we explicitly write ε in the antecedents of the inference rules.

The resulting minimal type in the communication rules is the union of the types of the two branches, that is, their least supertype.

The rules for well-formedness of session and class declarations are given in Figure 14. The declaration of a session $\texttt{s}$ in a class $\texttt{C}$ is well formed under the constraints $\mathscr{C}$ and $\mathscr{D}$ if the body $\texttt{e}$ is well typed under constraints $\mathscr{C}'$ and $\mathscr{D}$. The set $\mathscr{C}$ includes the constraints collected typing the body $\texttt{e}$ ($\mathscr{C}'$) and the equation $\chi^{\texttt{s}}_{\texttt{C}} = \theta$ that assigns to the session variable $\chi^{\texttt{s}}_{\texttt{C}}$ the session type scheme θ representing the communications performed in the body $\texttt{e}$. The condition in rule Sess-WF that $\circledcirc$ does not appear in θ is justified by the fact that $\circledcirc$ has no dual type, so sessions whose bodies would be typed with types containing $\circledcirc$ would be useless.

The well-formedness of a class declaration is checked under the union of the constraints collected checking the well-formedness of session declarations in $\texttt{C}$.

We define the *set of constraints of an inference class table ICT* as the pair $\langle \bigcup_{i\in I} \mathscr{C}_i ; \bigcup_{i\in I} \mathscr{D}_i \rangle$, where $\texttt{C}_i$ for $i \in I$ is the set of classes defined in ICT and $\texttt{class}\ \texttt{C}_i\ \ldots\ \texttt{ok with}\ \mathscr{D}_i, \mathscr{C}_i$.

8. Properties of the constraint-based typing

In this section we prove that the constraint typing rules of Figure 13 are sound and complete with respect to the typing rules of Figure 9.

Indeed, given an inference class table ICT that is well formed under constraints $\langle \mathscr{C}; \mathscr{D} \rangle$, if σ is a substitution that satisfies $\mathscr{C}$ and $\mathscr{D}$, then $\sigma(ICT)$ gives a well-formed class table according to the type derivation $\vdash$ (Soundness).

Conversely, for any well-formed class table CT, the corresponding inference class table CT^{-} will be well formed under constraints $\langle \mathscr{C}; \mathscr{D} \rangle$ such that there is a unique substitution σ that satisfies $\mathscr{C}$ and $\mathscr{D}$ (Completeness). Furthermore, we prove that $\sigma(CT^{-}) = CT$.

Definition 8.1 (type substitution). A type substitution σ is a finite mapping from session type variables to session types. The application of a substitution to a session type scheme is defined as follows:

$$\sigma(\mathtt{t}) = \mathtt{t}$$

$$\sigma(\chi^{\mathtt{s}}_{\mathtt{C}}) = \begin{cases} \mathtt{t} & \text{if } (\chi^{\mathtt{s}}_{\mathtt{C}} \mapsto \mathtt{t}) \in \sigma \\ \chi^{\mathtt{s}}_{\mathtt{C}} & \text{if } \chi^{\mathtt{s}}_{\mathtt{C}} \notin \mathsf{dom}(\sigma) \end{cases}$$

$$\sigma(\theta.\theta') = \sigma(\theta).\sigma(\theta')$$

$$\sigma(\dagger\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\}) = \dagger\{\mathtt{C}_1 \Rightarrow \sigma(\theta_1) \,[]\, \mathtt{C}_2 \Rightarrow \sigma(\theta_2)\}$$

$$\sigma(\mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\}) = \mu\alpha.\dagger\{\mathtt{C}_1 \Rightarrow \sigma(\theta_1) \,[]\, \mathtt{C}_2 \Rightarrow \sigma(\theta_2)\}.$$

Substitutions on inference class tables are defined as expected.

In the soundness property formulation, it is enough to consider expressions that occur in class tables.

Theorem 8.1 (soundness). Let ICT be an inference class table with set of constraints $\langle \mathscr{C}'; \mathscr{D}' \rangle$. If $\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}$ using ICT is such that $\mathscr{C} \subseteq \mathscr{C}'$ and $\mathscr{D} \subseteq \mathscr{D}'$ and σ is a substitution that satisfies $\mathscr{C}'$ and $\mathscr{D}'$, then $\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \sigma(\theta)$ using the class table $\sigma(ICT)$.

Proof. The proof is by induction on the type derivation of $\Gamma \vdash_{\mathrm{I}} \mathtt{e} : \mathtt{T} \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}$, with a case analysis on the final rule. We will only consider the most interesting cases.

Note that σ satisfies $\mathscr{C}'$ and $\mathscr{D}'$, so $\sigma(ICT)$ is a class table and $\sigma(\theta)$ is a session type.

— FldAss-T-I:

We have

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2 : \mathtt{T} \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}.$$

From rule FldAss-T-I, we have

$$\mathtt{T} = \mathit{ftype}_r(\mathtt{f}, \mathtt{T}_1)$$
$$\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$$
$$\mathscr{D} = \mathscr{D}_1 \cup \mathscr{D}_2$$
$$\theta = \theta_1.\theta_2$$

and

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \theta_1 \rhd \mathscr{C}_1, \mathscr{D}_1$$
$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \theta_2 \rhd \mathscr{C}_2, \mathscr{D}_2$$

for some $\mathscr{C}_1, \mathscr{C}_2, \mathscr{D}_1, \mathscr{D}_2, \mathtt{T}_1$ and some $\mathtt{T}_2$ such that

$$\mathtt{T}_2 <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1).$$

Since

$$\mathscr{C}_1 \subseteq \mathscr{C} \subseteq \mathscr{C}'$$
$$\mathscr{C}_2 \subseteq \mathscr{C} \subseteq \mathscr{C}'$$
$$\mathscr{D}_1 \subseteq \mathscr{D} \subseteq \mathscr{D}'$$
$$\mathscr{D}_2 \subseteq \mathscr{D} \subseteq \mathscr{D}',$$

by the induction hypothesis,

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \sigma(\theta_1)$$
$$\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \sigma(\theta_2)$$

using the class table $\sigma(\mathit{ICT})$. By applying rules Sub-T (since $\mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1) <: \mathit{ftype}_r(\mathtt{f}, \mathtt{T}_1)$ by definition) and FldAss-T, we get the result

$$\Gamma \vdash \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2 : \mathtt{T} \fatsemi \sigma(\theta)$$

using the class table $\sigma(\mathit{ICT})$.

— SessReq-T-I:
We have

$$\Gamma \vdash_1 \mathtt{e}_1.\mathtt{s}\,\{\mathtt{e}_2\} : \mathtt{T} \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}.$$

From rule SessReq-T-I, we have

$$\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$$
$$\mathscr{D} = \mathscr{D}_1 \cup \mathscr{D}_2 \cup \{\chi^{\mathtt{s}}_{\mathtt{C}} \bowtie \theta' | \mathtt{C} \in \mathtt{T}'\}$$

and

$$\Gamma \vdash_1 \mathtt{e}_1 : \mathtt{T}' \fatsemi \theta \rhd \mathscr{C}_1, \mathscr{D}_1$$
$$\Gamma \vdash_1 \mathtt{e}_2 : \mathtt{T} \fatsemi \theta' \rhd \mathscr{C}_2, \mathscr{D}_2$$

for some $\mathscr{C}_1, \mathscr{C}_2, \mathscr{D}_1, \mathscr{D}_2, \mathtt{T}'$. Since

$$\mathscr{C}_1 \subseteq \mathscr{C} \subseteq \mathscr{C}'$$
$$\mathscr{C}_2 \subseteq \mathscr{C} \subseteq \mathscr{C}'$$
$$\mathscr{D}_1 \subseteq \mathscr{D} \subseteq \mathscr{D}'$$
$$\mathscr{D}_2 \subseteq \mathscr{D} \subseteq \mathscr{D}',$$

by the induction hypothesis,

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}' \fatsemi \sigma(\theta)$$
$$\Gamma \vdash \mathtt{e}_2 : \mathtt{T} \fatsemi \sigma(\theta').$$

Moreover, the fact that σ satisfies $\mathscr{D}$, implies that $\sigma(\chi^{\mathtt{s}}_{\mathtt{C}}) \bowtie \sigma(\theta')$ for all $\mathtt{C} \in \mathtt{T}'$, and

$$\{\sigma(\chi^{\mathtt{s}}_{\mathtt{C}}) \,|\, \mathtt{C} \in \mathtt{T}'\} = \mathit{stype}(\mathtt{s}, \mathtt{T}')$$

using the class table $\sigma(ICT)$. Applying rule SessReq-T, we then get the result

$$\Gamma \vdash \mathtt{e}_1.\mathtt{s}\,\{\mathtt{e}_2\} : \mathtt{T} \fatsemi \sigma(\theta)$$

using the class table $\sigma(ICT)$.

— SessDel-T-I:
We have

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_0 \bullet \mathtt{s}\,\{\} : \mathtt{T} \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}.$$

From rule SessDel-T-I, we have

$$\mathscr{C} = \mathscr{C}_1 \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_1} \neq \varepsilon\} \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_i} = \chi^{\mathtt{s}}_{\mathtt{C}_j} | i \neq j \in \{1, \ldots, n\}\},$$

and

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_0 : \mathtt{T}' \fatsemi \theta \rhd \mathscr{C}_1, \mathscr{D}$$

and

$$rtype(\mathtt{s}, \mathtt{T}') = \mathtt{T}$$
$$\mathtt{T}' = \mathtt{C}_1 \vee \cdots \vee \mathtt{C}_n$$

for some $\mathscr{C}_1, \mathtt{T}', \mathtt{C}_1, \cdots, \mathtt{C}_n$. Since

$$\mathscr{C}_1 \subseteq \mathscr{C} \subseteq \mathscr{C}',$$

by the induction hypothesis,

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{T}' \fatsemi \sigma(\theta)$$

and

$$stype(\mathtt{s}, \mathtt{T}') = \{\sigma(\chi^{\mathtt{s}}_{\mathtt{C}_1})\},$$

with $\sigma(\chi^{\mathtt{s}}_{\mathtt{C}_1}) \neq \varepsilon$, using the class table $\sigma(ICT)$. Applying rule SessDel-T, we then get the result

$$\Gamma \vdash \mathtt{e}_0 \bullet \mathtt{s}\,\{\} : \mathtt{T} \fatsemi \sigma(\theta)$$

using the class table $\sigma(ICT)$.

— SendC-T-I:
We have

$$\Gamma \vdash_{\mathrm{I}} \mathtt{sendC}(\mathtt{e}_0)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}.$$

From rule SendC-T-I, we have

$$\theta = !\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\}$$
$$\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3$$
$$\mathscr{D} = \mathscr{D}_1 \cup \mathscr{D}_2 \cup \mathscr{D}_3$$

and

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_i : \mathtt{T}_i \fatsemi \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i \quad (i \in \{1, 2\})$$

for some $\theta_1, \theta_2, \mathscr{C}_1, \mathscr{C}_2, \mathscr{C}_3, \mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3$. Moreover,

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_0 : \mathtt{T}' \fatsemi \varepsilon \rhd \mathscr{C}_3, \mathscr{D}_3$$

for some

$$\mathtt{T}' <: \mathtt{C}_1 \vee \mathtt{C}_2.$$

Since

$$\mathscr{C}_j \subseteq \mathscr{C} \subseteq \mathscr{C}'$$
$$\mathscr{D}_j \subseteq \mathscr{D} \subseteq \mathscr{D}'$$

for $j \in \{1,2,3\}$, by the induction hypothesis,

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{T}' \fatsemi \varepsilon$$
$$\Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \fatsemi \sigma(\theta_i)$$

using the class table $\sigma(ICT)$. Applying rule Sub-T, we then get

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \varepsilon$$
$$\Gamma \vdash \mathtt{e}_i : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \sigma(\theta_i).$$

So SendC-T applies, and we obtain

$$\Gamma \vdash \mathtt{sendC}(\mathtt{e}_0)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[\!]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi \sigma(\theta)$$

using the class table $\sigma(ICT)$.

— ReceiveC-T-I, SendW-T-I, ReceiveW-T-I:
All these cases are similar to the above. □

Theorem 8.2 (completeness). Let CT be a well-formed class table and σ be a substitution such that $\{\sigma(\chi^{\mathtt{s}}_{\mathtt{C}})\} = \mathit{stype}(\mathtt{s},\mathtt{C})$, for any $\mathtt{s},\mathtt{C} \in CT$.

Then, for any expression e, if $\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}$ using CT, we have for some $\mathscr{C}, \mathscr{D}, \mathtt{T}'$:

(i) $\Gamma \vdash_1 \mathtt{e} : \mathtt{T}' \fatsemi \theta \rhd \mathscr{C}, \mathscr{D}$ using CT^{-}.
(ii) $\mathtt{T}' <: \mathtt{T}$.
(iii) σ satisfies $\mathscr{C}$ and $\mathscr{D}$.
(iv) $\sigma(\theta) = \mathtt{t}$.

Proof. The proof is by induction on the type derivation of $\Gamma \vdash \mathtt{e} : \mathtt{T} \fatsemi \mathtt{t}$, with a case analysis on the final rule. We will only consider the most interesting cases:

— FldAss-T:
We have

$$\Gamma \vdash \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2 : \mathtt{T} \fatsemi \mathtt{t}.$$

From rule FldAss-T we have

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

and

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \mathtt{t}_1$$
$$\Gamma \vdash \mathtt{e}_2 : \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1) \fatsemi \mathtt{t}_2$$

for some $\mathtt{T}_1$, $\mathtt{t}_1$, $\mathtt{t}_2$.
By the induction hypothesis, we have for some $\mathtt{T}'_1 <: \mathtt{T}_1$, $\mathtt{T}_2 <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1)$, θ_1, θ_2, $\mathscr{C}_1$, $\mathscr{C}_2$,$\mathscr{D}_1$ and $\mathscr{D}_2$:
(1) $\Gamma \vdash_\iota \mathtt{e}_1 : \mathtt{T}'_1 \fatsemi \theta_1 \rhd \mathscr{C}_1, \mathscr{D}_1$ and $\Gamma \vdash_\iota \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \theta_2 \rhd \mathscr{C}_2, \mathscr{D}_2$.
(2) σ satisfies $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{D}_1$, $\mathscr{D}_2$, and $\sigma(\theta_1) = \mathtt{t}_1$ and $\sigma(\theta_2) = \mathtt{t}_2$.
The condition

$$\mathtt{T}_2 <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}'_1)$$

holds by

$$\mathtt{T}_2 <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1)$$

since $\mathtt{T}'_1 <: \mathtt{T}_1$ implies

$$\mathit{ftype}_w(\mathtt{f}, \mathtt{T}_1) <: \mathit{ftype}_w(\mathtt{f}, \mathtt{T}'_1)$$

by the definition of ftype_w. So we can apply rule FLDASS-T-I to (1) to give

$$\Gamma \vdash_\iota \mathtt{e}_1.\mathtt{f} := \mathtt{e}_2 : \mathit{ftype}_r(\mathtt{f}, \mathtt{T}'_1) \fatsemi \theta_1.\theta_2 \rhd \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D}_1 \cup \mathscr{D}_2.$$

Note that $\mathtt{T}'_1 <: \mathtt{T}_1$ implies

$$\mathit{ftype}_r(\mathtt{f}, \mathtt{T}'_1) <: \mathit{ftype}_r(\mathtt{f}, \mathtt{T}_1)$$

by the definition of ftype_r. From (2), we can then conclude that σ satisfies

$$\mathscr{C}_1 \cup \mathscr{C}_2$$
$$\mathscr{D}_1 \cup \mathscr{D}_2,$$

and that

$$\sigma(\theta_1.\theta_2) = \mathtt{t}.$$

— SESSREQ-T:
We have

$$\Gamma \vdash \mathtt{e}_1.\mathtt{s}\{\mathtt{e}_2\} : \mathtt{T} \fatsemi \mathtt{t}.$$

From rule SESSREQ-T, we have

$$\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \fatsemi \mathtt{t}$$
$$\Gamma \vdash \mathtt{e}_2 : \mathtt{T} \fatsemi \mathtt{t}_2$$

and $\mathtt{t}_2 \bowtie \mathtt{t}'$ for some $\mathtt{T}_1$, $\mathtt{t}_2$ and for all $\mathtt{t}'$ such that $\mathtt{t}' \in \mathit{stype}(\mathtt{s}, \mathtt{T}_1)$.
By the induction hypothesis, we have for some $\mathtt{T}'_1 <: \mathtt{T}_1$, $\mathtt{T}_2 <: \mathtt{T}$, θ_1, θ_2, $\mathscr{C}_2$, $\mathscr{C}_2$, $\mathscr{D}_1$, $\mathscr{D}_2$:
(1) $\Gamma \vdash_\iota \mathtt{e}_1 : \mathtt{T}'_1 \fatsemi \theta_1 \rhd \mathscr{C}_1, \mathscr{D}_1$ and $\Gamma \vdash_\iota \mathtt{e}_2 : \mathtt{T}_2 \fatsemi \theta_2 \rhd \mathscr{C}_2, \mathscr{D}_2$.
(2) σ satisfies $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{D}_1$, $\mathscr{D}_2$, and $\sigma(\theta_1) = \mathtt{t}$ and $\sigma(\theta_2) = \mathtt{t}_2$.
Let $\mathtt{T}_1 = \mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$. From rule SESSREQ-T-I and (1), we have

$$\Gamma \vdash_\iota \mathtt{e}_1.\mathtt{s}\{\mathtt{e}_2\} : \mathtt{T}_2 \fatsemi \theta_1 \rhd \mathscr{C}, \mathscr{D},$$

where

$$\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$$
$$\mathscr{D} = \mathscr{D}_1 \cup \mathscr{D}_2 \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_i} \bowtie \theta_2 | i \in \{1, \ldots, n\}\}.$$

Since, by hypothesis,

$$\{\sigma(\chi^{\mathtt{s}}_{\mathtt{C}})\} = stype(\mathtt{s}, \mathtt{C})$$

for all $\mathtt{s}, \mathtt{C} \in CT$, and by definition,

$$stype(\mathtt{s}, \mathtt{T}_1) = \bigcup_{i \in \{1,\ldots,n\}} stype(\mathtt{s}, \mathtt{C}_i),$$

by (2), the condition $\mathtt{t}_2 \bowtie \mathtt{t}'$ for all $\mathtt{t}' \in stype(\mathtt{s}, \mathtt{T}_1)$ implies

$$\sigma(\chi^{\mathtt{s}}_{\mathtt{C}_i}) \bowtie \sigma(\theta_2)$$

for all $i \in \{1, \ldots, n\}$. We can then conclude from (2) that σ satisfies $\mathscr{C}$ and $\mathscr{D}$.

— SessDel-T:
We have

$$\Gamma \vdash \mathtt{e}_0 \bullet \mathtt{s}\,\{\} : \mathtt{T} \fatsemi \mathtt{t}.$$

From rule SessDel-T, we have

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \fatsemi \mathtt{t}_0,$$

and

$$\begin{aligned} stype(\mathtt{s}, \mathtt{T}_0) &= \{\mathtt{t}'\} \\ \mathtt{t}' &\neq \varepsilon \\ \mathtt{t} &= \mathtt{t}_0.\mathtt{t}' \\ rtype(\mathtt{s}, \mathtt{T}_0) &= \mathtt{T}. \end{aligned}$$

By the induction hypothesis, we have for some $\mathtt{T}'_0 <: \mathtt{T}_0$, θ_0, $\mathscr{C}_0$, $\mathscr{D}$:

(1) $\Gamma \vdash_{\mathrm{I}} \mathtt{e}_0 : \mathtt{T}'_0 \fatsemi \theta_0 \rhd \mathscr{C}_0, \mathscr{D}$.

(2) σ satisfies $\mathscr{C}_0$ and $\mathscr{D}$ and $\sigma(\theta_0) = \mathtt{t}_0$.

Let $\mathtt{T}_0 = \mathtt{C}_1 \vee \ldots \vee \mathtt{C}_n$. From rule SessDel-T-I and (1), we have

$$\Gamma \vdash_{\mathrm{I}} \mathtt{e}_0 \bullet \mathtt{s}\,\{\} : rtype(\mathtt{s}, \mathtt{T}'_0) \fatsemi \theta_0.\chi^{\mathtt{s}}_{\mathtt{C}_1} \rhd \mathscr{C}, \mathscr{D},$$

where

$$\mathscr{C} = \mathscr{C}_0 \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_1} \neq \varepsilon\} \cup \{\chi^{\mathtt{s}}_{\mathtt{C}_i} = \chi^{\mathtt{s}}_{\mathtt{C}_j} | i \neq j \in \{1, \ldots, n\}\}.$$

Since $stype(\mathtt{s}, \mathtt{T}_0) = \{\mathtt{t}'\}$ implies $stype(\mathtt{s}, \mathtt{C}_i) = \{\mathtt{t}'\}$ for $i \in \{1, \ldots, n\}$ and, by hypothesis,

$$\{\sigma(\chi^{\mathtt{s}}_{\mathtt{C}_i})\} = stype(\mathtt{s}, \mathtt{C}_i),$$

we get

$$\sigma(\chi^{\mathtt{s}}_{\mathtt{C}_i}) = \mathtt{t}'.$$

So

$$\sigma(\theta.\chi^{\mathtt{s}}_{\mathtt{C}_1}) = \mathtt{t}_0.\mathtt{t}'$$

and σ satisfies $\mathscr{C}$.

From $\mathtt{T}'_0 <: \mathtt{T}_0$, we have

$$rtype(\mathtt{s}, \mathtt{T}'_0) <: rtype(\mathtt{s}, \mathtt{T}_0),$$

so we obtain the result.

— SendC-T:
We have

$$\Gamma \vdash \mathtt{sendC}(\mathtt{e}_0)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T} \fatsemi \mathtt{t}.$$

From rule SendC-T, we have

$$\mathtt{t} = !\{\mathtt{C}_1 \Rightarrow \mathtt{t}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{t}_2\}$$

and

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_1 \vee \mathtt{C}_2 \fatsemi \varepsilon$$
$$\Gamma \vdash \mathtt{e}_i : \mathtt{T} \fatsemi \mathtt{t}_i \qquad \text{for } i \in \{1,2\}.$$

By the induction hypothesis, for some $\mathtt{T}' <: \mathtt{C}_1 \vee \mathtt{C}_2$, $\mathscr{C}$, $\mathscr{D}$, $\mathtt{T}_i <: \mathtt{T}$, θ_i, $\mathscr{C}_i$, and $\mathscr{D}_i$ with $i \in \{1,2\}$:
(1) $\Gamma \vdash_1 \mathtt{e}_0 : \mathtt{T}' \fatsemi \varepsilon \rhd \mathscr{C}, \mathscr{D}$.
(2) $\Gamma \vdash_1 \mathtt{e}_i : \mathtt{T}_i \fatsemi \theta_i \rhd \mathscr{C}_i, \mathscr{D}_i$.
(3) σ satisfies $\mathscr{C}$, $\mathscr{D}$, $\mathscr{C}_i$, $\mathscr{D}_i$ and $\sigma(\theta_i) = \mathtt{t}_i$.
We can now apply rule SendC-T-I to obtain

$$\Gamma \vdash_1 \mathtt{sendC}(\mathtt{e}_0)\{\mathtt{C}_1 \Rightarrow \mathtt{e}_1 \,[]\, \mathtt{C}_2 \Rightarrow \mathtt{e}_2\} : \mathtt{T}_1 \vee \mathtt{T}_2 \fatsemi !\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\} \rhd \mathscr{C} \cup \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{D} \cup \mathscr{D}_1 \cup \mathscr{D}_2.$$

Since $\mathtt{T}_1 <: \mathtt{T}$ and $\mathtt{T}_2 <: \mathtt{T}$, we get $\mathtt{T}_1 \vee \mathtt{T}_2 <: \mathtt{T}$. We can now conclude from (3) that

$$\sigma(!\{\mathtt{C}_1 \Rightarrow \theta_1 \,[]\, \mathtt{C}_2 \Rightarrow \theta_2\}) = \mathtt{t}$$

and σ satisfies

$$\mathscr{D} \cup \mathscr{D}_1 \cup \mathscr{D}_2$$
$$\mathscr{C} \cup \mathscr{C}_1 \cup \mathscr{C}_2.$$

— ReceiveC-T-I, SendW-T-I, ReceiveW-T-I:
These are all similar to the above cases. □

It is interesting to note that we do not need to consider principal solutions as in the standard approach ((Pierce 2002), Chapter 22). The reason is that the classes of exchanged objects are explicit in communication expressions.

Corollary 8.1 (uniqueness of the solution). Let CT be a class table and CT^- be the corresponding inference class table with constraints ($\mathscr{C}$,$\mathscr{D}$). Let $\sigma(CT^-) = CT$. For any substitution σ' that satisfies ($\mathscr{C}$,$\mathscr{D}$) and such that $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$, we get $\sigma' = \sigma$.

Proof. Let us suppose *ad absurdum* that $\sigma' \neq \sigma$, that is, $\sigma'(CT^-) = CT' \neq CT$. The only difference between CT' and CT is in the session types declared in the sessions definitions. This contradicts Proposition 5.1 and rule Sess-WF. □

Summarising, when read from bottom to top, the constraint typing rules determine an algorithm that calculates the constraints that must be satisfied in order for a class table ICT to be well formed. If a solution exists, it is the (unique) substitution σ that verifies both $\mathscr{C}$ and $\mathscr{D}$. The procedure for finding this substitution σ consists of two steps. First we apply a standard unification algorithm on first-order type expressions (see Pierce (2002, Chapter 22)) to solve the equality constraints in $\mathscr{C}$. Then we verify that σ satisfies all the duality constraints in $\mathscr{D}$. If this procedure succeeds, then σ gives the session types that decorate all the session declarations in such a way that $\sigma(ICT)$ is well formed.

9. Related work

Union types have been shown to be useful for enhancing the flexibility of subtyping in the following settings: functional languages (Barbanera *et al.* 1995; Frisch *et al.* 2008); object-oriented languages (Igarashi and Nagira 2007); languages manipulating semi-structured data (Gapeyev and Pierce 2003); and the π-calculus (Castagna *et al.* 2008; Castagna *et al.* 2009).

It is interesting to compare $\mathscr{F}\mathsf{SAM}^{\vee}$ with FJ $\vee$, which is an extension of FJ with union types proposed in Igarashi and Nagira (2007). They define union types as in the current paper: the essential difference is that they have traditional methods rather than sessions. The method signatures are of the form $\overline{\mathtt{T}} \to \mathtt{T}$. The method type lookup function applied to a method name `m` and a type `T` gives a set of method signatures, that is, all the signatures of `m` in the classes that build `T`. This is similar to our *stype* function, which returns a set of session types. The method-call rule checks that the types of the parameters agree with all the signatures found by the method type lookup function for the type of the object. Our SessReq-T rule also requires the session type of the co-body to 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 2 extends without any changes to methods with union types.

Session types were first introduced into model communication protocols between π-calculus processes (Honda 1993; Takeuchi *et al.* 1994; Honda *et al.* 1998). They have been made more expressive by enriching them with: correspondence assertions (Bonelli *et al.* 2005); subtyping (Gay and Hole 2005); bounded polymorphism (Gay 2008); higher-order processes (Mostrous and Yoshida 2007; Mostrous and Yoshida 2009); exceptions (Carbone *et al.* 2008b); and concurrent constraints (Coppo and Dezani-Ciancaglini 2009). They have also been made safer by assuring deadlock-freedom (Dezani-Ciancaglini *et al.* 2008; Bettini *et al.* 2008b). Session types have also been extended to include: multi-party communications (Bonelli and Compagnoni 2008; Carbone *et al.* 2008a); action permutations (Honda *et al.* 2009); design by contracts (Bocchi *et al.* 2010); dependent types for parametricity (Yoshida *et al.* 2010); upper bounds on buffer sizes (Deniélou and Yoshida 2010); and access/information flow control (Capecchi *et al.* 2010a; Capecchi *et al.* 2011). Session types have also been developed for: CORBA (Vallecillo *et al.* 2002); functional languages (Gay *et al.* 2003; Vasconcelos *et al.* 2006; Bhargavan *et al.* 2009); boxed ambients (Garralda *et al.* 2006); the W3C standard description language for Web

Services CDL (Carbone *et al.* 2007; Web Services Choreography Working Group 2002; Sparkes 2006; Honda *et al.* 2007); and object-oriented programming languages.

The rest of this section is devoted to the literature on session types in the object-oriented paradigm.

The earlier papers Dezani-Ciancaglini *et al.* (2005), Dezani-Ciancaglini *et al.* (2006), Coppo *et al.* (2007), Dezani-Ciancaglini *et al.* (2007) and Dezani-Ciancaglini *et al.* (2009) discuss a multi-threaded object-oriented calculus augmented with session primitives that supports session names as parameters of methods, spawning, iterative sessions and delegation.

The language Sing# (Fähndrich *et al.* 2006) is a variant of C# that combines session types with ownership types (Clarke *et al.* 2001), supports message-based communication through a designed heap area (shared memory) and allows interfaces between OS-modules to be described as message-passing conversations. CoreSing# (Bono *et al.* 2011) is a core calculus inspired by the main features of Sing#. It is equipped with a type system that uses session types and a novel form of ownership types to ensure the absence of communication errors, memory faults and memory leaks in a communications model based on copyless message passing.

SJ (Hu *et al.* 2008) is an extension of Java with syntax for session types and structured communication operations. The main features of SJ are asynchronous message passing, delegation, session subtyping, interleaving, class downloading and failure handling. Hu *et al.* (2010) presents an extension of SJ that allows type-safe event-driven session programming.

Gay *et al.* (2010) formalises a core distributed class-based object-oriented language with a static type system that combines session-typed channels and a form of typestates. Each class definition has a session type that specifies the possible sequences of method calls. Channels can be stored in object fields, and separated methods implement parts of sessions. The availability of methods depends on the state of objects.

The amalgamation of the notion of session-based communication with object-oriented programming was first developed in Drossopoulou *et al.* (2007). A characteristic of this design is that channel names are only generated at run time, and, as a consequence, only delegation of a session to another session within the same thread is expressible. Since the delegating and delegated sessions can have different objects as receivers, this delegation is in this respect related to the delegation of method execution in object-based calculi (Lieberman 1986). $\mathscr{F}\mathsf{SAM}^{\vee}$ extends the calculus of Drossopoulou *et al.* (2007) with union types and a cleaner and simpler typing and operational semantics, since delegation in Drossopoulou *et al.* (2007) requires *ad hoc* run-time constructors. Capecchi *et al.* (2009) added generic types to a language/calculus based on the approach of Drossopoulou *et al.* (2007), but we claim that union types are a better fit than generic types for our communication primitives based on classes of exchanged objects. We think that the present type reconstruction cannot be adapted easily to the session types of Capecchi *et al.* (2009) because of intrinsic difficulties in performing type inference with generic types. Giachino (2009) presents an extension of $\mathscr{F}\mathsf{SAM}^{\vee}$ with intersection and negation types that allows a service-oriented interpretation of session overloading.

10. Conclusions

The core language $\mathscr{F}\mathsf{SAM}^{\vee}$, which was first presented in Bettini *et al.* (2008a), showed how the addition of union types to an object-oriented language with session types enhances flexibility.

In the current paper, we have presented a full formalisation of the language and proved that the language is type safe. Moreover, we have presented an inference algorithm that gives an expression its minimal type and calculates the constraints that must be satisfied in order to reconstruct the related (unique) session type for each session declaration.

The language $\mathscr{F}\mathsf{SAM}^{\vee}$can be also viewed as a kernel proposal for generalising the standard notion of sessionless methods in the object-oriented framework, where method call interactions between two objects is limited to the initial sending of argument values for parameters. Once the syntax of the expressions is extended to send/receive operations, a method definition can include a sequence of interactions. The typechecking will be responsible for deriving session types for methods, thereby determining the appropriate evaluation rule to be used for method invocation: an empty session type will cause a standard semantics, but a non-empty one will use the evaluation rules defined in the current paper.

The amalgamation of communication-centred and object-oriented programming, as developed in Drossopoulou *et al.* (2007), Capecchi *et al.* (2009) and the current paper, does not allow some common patterns of concurrent programming to be expressed naturally. Session nesting is a strong limitation in the programming design: for example, the only way of implementing a 'forwarder' is to create a new session for each forwarded message. Our restricted delegation does not allow us to write a server that does load-balancing by delegation to worker threads in a straightforward way. We plan to remove these drawbacks (presumably by adding explicit channels) and to extend our approach in various directions. In particular, we plan to integrate this approach with multi-party session communication (Carbone *et al.* 2008a; Bettini *et al.* 2008b), access and information flow control (Capecchi *et al.* 2010a; Capecchi *et al.* 2011) and exception handling (Capecchi *et al.* 2010b).

We plan to develop a prototype implementation of a language based on the approach presented in this paper. A possible tool for the implementation of the run-time system of our language is IMC[†], which is a Java framework for implementing network applications that provides reusable mechanisms for dealing with the implementation of communication protocols. Indeed, IMC has already been used for implementing the run-time system of calculi with session-based communication primitives (Bettini *et al.* 2008c). This would also allow us to embed our type system for session types into a distributed setting; we do not see any crucial issues arising from transposing our session type setting to a distributed context since our approach, as stated in the Introduction, is agnostic with respect to other aspects of the language. Of course, our session types do not deal with network failures since they are only concerned with the correctness of the communication protocols.

[†] See `http://imc-fi.sourceforge.net`.

Acknowledgements

We are grateful to the anonymous referees for their comments and suggestions, which contributed significantly to improving an earlier draft.

References

Barbanera, F., Dezani-Ciancaglini, M. and de'Liguoro, U. (1995) Intersection and Union Types: Syntax and Semantics. *Information and Computation* **119** 202–230.

Bettini, L., Capecchi, S., Dezani-Ciancaglini, M., Giachino, E. and Venneri, B. (2008a) Session and Union Types for Object Oriented Programming. In: Concurrency, Graphs and Models. *Springer-Verlag Lecture Notes in Computer Science* **5065** 659–680.

Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M. and Yoshida, N. (2008b) Global Progress in Dynamically Interleaved Multiparty Sessions. In: CONCUR'08. *Springer-Verlag Lecture Notes in Computer Science* **5201** 418–433.

Bettini, L., De Nicola, R. and Loreti, M. (2008c) Implementing Session Centered Calculi. In: COORDINATION'08. *Springer-Verlag Lecture Notes in Computer Science* **5052** 17–32.

Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C. and Leifer, J. J. (2009) Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In: *Proceedings 22nd IEEE Computer Security Foundations Symposium, 2009 (CSF '09)* 124–140.

Bocchi, L., Honda, K., Tuosto, E. and Yoshida, N. (2010) A Theory of Design-by-Contract for Distributed Multiparty Interactions. In: CONCUR'10. *Springer-Verlag Lecture Notes in Computer Science* **6269** 162–176.

Bonelli, E. and Compagnoni, A. (2008) Multipoint Session Types for a Distributed Calculus. In: TGC'07. *Springer-Verlag Lecture Notes in Computer Science* **4912** 240–256.

Bonelli, E., Compagnoni, A. and Gunter, E. (2005) Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming* **15** (2) 219–248.

Bono, V., Messa, C. and Padovani, L. (2011) Typing Copyless Message Passing. In: ESOP'11. *Springer-Verlag Lecture Notes in Computer Science* **6602** 57–76.

Capecchi, S., Castellani, I. and Dezani-Ciancaglini, M. (2011) Information Flow Safety in Multiparty Sessions. In: EXPRESS'11. *Electronic Proceedings in Theoretical Computer Science* **64** 16–31.

Capecchi, S., Castellani, I., Dezani-Ciancaglini, M. and Rezk, T. (2010a) Session Types for Access and Information Flow Control. In: CONCUR'10. *Springer-Verlag Lecture Notes in Computer Science* **6269** 237–252.

Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S. and Giachino, E. (2009) Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science* **410** 142–167.

Capecchi, S., Giachino, E. and Yoshida, N. (2010b) Global Escape in Multiparty Sessions. In: FSTTCS'10. *LIPIcs* **8**, Schloss Dagstuhl–Leibniz-Zentrum für Informatik 338–351.

Carbone, M., Honda, K. and Yoshida, N. (2007) Structured Communication-Centred Programming for Web Services. In: ESOP'07. *Springer-Verlag Lecture Notes in Computer Science* **4421** 2–17.

Carbone, M., Honda, K. and Yoshida, N. (2008a) Multiparty Asynchronous Session Types. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'08)* 273–284.

Carbone, M., Honda, K. and Yoshida, N. (2008b) Structured Interactional Exceptions for Session Types. In: CONCUR'08. *Springer-Verlag Lecture Notes in Computer Science* **5201** 402–417.

Castagna, G., De Nicola, R. and Varacca, D. (2008) Semantic Subtyping for the π-calculus. *Theoretical Computer Science* **398** (1–3) 217–242.

Castagna, G., Dezani-Ciancaglini, M., Giachino, E. and Padovani, L. (2009) Foundations of Session Types. In: *PPDP '09: Proceedings of the 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* 219–230.

Clarke, D., Noble, J. and Potter, J. (2001) Simple Ownership Types for Object Containment. In: ECOOP'01. *Springer-Verlag Lecture Notes in Computer Science* **2072** 53–76.

Coppo, M. and Dezani-Ciancaglini, M. (2009) Structured Communications with Concurrent Constraints. In: TGC'08. *Springer-Verlag Lecture Notes in Computer Science* **5474** 104–125.

Coppo, M., Dezani-Ciancaglini, M. and Yoshida, N. (2007) Asynchronous Session Types and Progress for Object-Oriented Languages. In: FMOODS'07. *Springer-Verlag Lecture Notes in Computer Science* **4468** 1–31.

Deniélou, P.-M. and Yoshida, N. (2010) Buffered Communication Analysis in Distributed Multiparty Sessions. In: CONCUR'10. *Springer-Verlag Lecture Notes in Computer Science* **6269** 343–357.

Dezani-Ciancaglini, M., de' Liguoro, U. and Yoshida, N. (2008) On Progress for Structured Communications. In: TGC'07. *Springer-Verlag Lecture Notes in Computer Science* **4912** 257–275.

Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E. and Yoshida, N. (2007) Bounded Session Types for Object-Oriented Languages. In: FMCO'06. *Springer-Verlag Lecture Notes in Computer Science* **4709** 207–245.

Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D. and Yoshida, N. (2009) Session Types for Object-Oriented Languages. *Information and Computation* **207** (5) 595–641.

Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N. and Drossopoulou, S. (2006) Session Types for Object-Oriented Languages. In: ECOOP'06. *Springer-Verlag Lecture Notes in Computer Science* **4067** 328–352.

Dezani-Ciancaglini, M., Yoshida, N., Ahern, A. and Drossopoulou, S. (2005) A Distributed Object Oriented Language with Session Types. In: TGC'05. *Springer-Verlag Lecture Notes in Computer Science* **3705** 299–318.

Drossopoulou, S., Dezani-Ciancaglini, M. and Coppo, M. (2007) Amalgamating the Session Types and the Object Oriented Programming Paradigms. Presented at MPOOL'07.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R. and Levi, S. (2006) Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys2006)* 177–190.

Frisch, A., Castagna, G. and Benzaken, V. (2008) Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *Journal of the ACM* **55** (4) 1–64.

Gapeyev, V. and Pierce, B. C. (2003) Regular Object Types. In: ECOOP'03. *Springer-Verlag Lecture Notes in Computer Science* **2743** 151–175.

Garralda, P., Compagnoni, A. and Dezani-Ciancaglini, M. (2006) BASS: Boxed Ambients with Safe Sessions. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming* 61–72.

Gay, S. (2008) Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science* **18** (5) 895–930.

Gay, S. and Hole, M. (2005) Subtyping for Session Types in the Pi-Calculus. *Acta Informatica* **42** (2/3) 191–225.

Gay, S., Vasconcelos, V. T. and Ravara, A. (2003) Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow.

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. and Caldeira, A. Z. (2010) Modular Session Types for Distributed Object-oriented Programming. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 299–312.

Giachino, E. (2009) *Session Types: Semantic Foundations and Object-Oriented Applications*, Ph.D. thesis, Università degli Studi di Torino and Université Paris 7.

Honda, K. (1993) Types for Dyadic Interaction. In: CONCUR'93. *Springer-Verlag Lecture Notes in Computer Science* **715** 509–523.

Honda, K., Mostrous, D. and Yoshida, N. (2009) Global Principal Typing in Partially Commutative Asynchronous Sessions. In: ESOP'09. *Springer-Verlag Lecture Notes in Computer Science* **5502** 316–332.

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998) Language Primitives and Type Disciplines for Structured Communication-based Programming. In: ESOP'98. *Springer-Verlag Lecture Notes in Computer Science* **1381** 22–138.

Honda, K., Yoshida, N. and Carbone, M. (2007) Web Services, Mobile Processes and Types. *EATCS Bulletin* **91** 160–188.

Hu, R., Kouzapas, D., Pernet, O., Yoshida, N. and Honda, K. (2010) Type-Safe Eventful Sessions in Java. In: ECOOP'10. *Springer-Verlag Lecture Notes in Computer Science* **6183** 329–353.

Hu, R., Yoshida, N. and Honda, K. (2008) Session-Based Distributed Programming in Java. In: ECOOP'08. *Springer-Verlag Lecture Notes in Computer Science* **5142** 516–541.

Igarashi, A. and Nagira, H. (2007) Union Types for Object Oriented Programming. *Journal of Object Technology* **6** (2) 31–52.

Igarashi, A., Pierce, B. C. and Wadler, P. (2001) Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM TOPLAS* **23** (3) 396–450.

Lieberman, H. (1986) Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In: Proceedings of the 1986 conference on Object-Oriented Programming Languages, Systems and Applications: OOPLSA '86. *SIGPLAN Notices* **21** (11) 214–223.

Mostrous, D. and Yoshida, N. (2007) Two Session Typing Systems for Higher-Order Mobile Processes. In: TLCA'07. *Springer-Verlag Lecture Notes in Computer Science* **4583** 321–335.

Mostrous, D. and Yoshida, N. (2009) Session-Based Communication Optimisation for Higher-Order Mobile Processes. In: TLCA'09. *Springer-Verlag Lecture Notes in Computer Science* **5608** 203–218.

Pierce, B. C. (2002) *Types and Programming Languages*, MIT Press.

Sparkes, S. (2006) Conversation with Steve Ross-Talbot. *ACM Queue* **4** (2) 14–23.

Takeuchi, K., Honda, K. and Kubo, M. (1994) An Interaction-based Language and its Typing System. In: PARLE'94. *Springer-Verlag Lecture Notes in Computer Science* **817** 398–413.

Vallecillo, A., Vasconcelos, V. T. and Ravara, A. (2002) Typing the Behavior of Objects and Components using Session Types. In: FOCLASA'02. *Electronic Notes in Theoretical Computer Science* **68** (3) 439–456.

Vasconcelos, V. T., Gay, S. and Ravara, A. (2006) Typechecking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science* **368** (1–2) 64–87.

Web Services Choreography Working Group (2002) Web Services Choreography Description Language. (Available at `http://www.w3.org/2002/ws/chor/`.)

Yoshida, N., Deniélou, P.-M., Bejleri, A. and Hu, R. (2010) Parameterised Multiparty Session Types. In: FOSSACS'10. *Springer-Verlag Lecture Notes in Computer Science* **6014** 128–145.

Yoshida, N. and Vasconcelos, V. T. (2007) Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. In: SecRet'06. *Electronic Notes in Theoretical Computer Science* **171** (4) 73–93.