Sessions, from types to programming languages

Vasco T. Vasconcelos
LaSIGE, Faculty of Sciences, University of Lisbon

Abstract

We discuss session types independently of any programming language. We then embody the notion in languages from three different paradigms: the pi calculus, a functional language, and an object-oriented language.

1 Introduction

Session types allow a concise description of a continuum of interactions among different partners. The notion was originally introduced with the aim of specifying and disciplining interactions between two partners running in parallel and communicating via message passing [8, 12]; the setting was then a mild variation of the pi-calculus [10]. Since then, session types were incorporated in different paradigms, including functional languages, object-oriented languages and service-oriented computing, to name a few. Reference [2] presents a recent overview of the field. What once looked like a notion of types tuned for a particular paradigm of computation, turned out to be a quite rich, language-independent idea.

Traditionally, session types have been used to describe linear interactions only, that is, interactions between exactly two partners (or two threads). But linearity alone is not enough to express the rich computational structures one might be interested in. As such, (linear) session types are often complemented with some other sort of (shared) types, resulting in two disjoint categories for types incorporating a certain dose of redundancy. Inspired on a formulation of linear types for functional programming [15], where types (pre-types, more precisely) are annotated with qualifiers, we have formulated an elegant theory of session types for the pi-calculus [14]. A pre-type equipped with a linear qualifier plays the role of a traditional session type; when annotated with an unrestricted qualifier, the type describes an entity shared by multiple partners.

In the vast majority of applications, sessions types are associated to message passing, describing the messages flowing on communication channels, regardless of the host programming language (pi calculus, functional, or object-oriented). But this need not be the case. Session types have been used to describe the behavior of objects in component models [13], as well as attached to class definitions where they specify possible sequences of method calls [5].

This paper presents a version of session types equipped with lin/un qualifiers [14], meant as descriptions for *communication media* in general. It then incorporates the notion into three different programming paradigms: a purely concurrent message passing system (embodied by the pi calculus), a multithreaded functional language equipped with message passing, and a multithreaded object-based language. The communication medium is instantiated as communication channels in the first two languages; and as object references in the last. The tone is left informal, references point to detailed descriptions.

2 Session Types

This section introduces the notion of lin/un qualified session types as well as our running example.

The running example. Our running example is that of an *online petition service*. Petition writers start by providing the title of the petition, a piece of text describing the situation and what is needed, and the deadline for signature collection. In order to make life easier to petition writers, the service allows this information to be added with no particular order; writers can even resubmit information if needed.

Once writers are happy with the provided petition details, time comes for submission. This is the point where writers commit to the uploaded data and seek approval for starting the petition. If accepted, meaning for example that the deadline for signature collection is not in the past or that there is no other running petition under the same name, then writers may start promoting the petition. Promoting a petition means two things: signing and disseminating. The petition writer may sign the petition, and must get people to sign it, by letting them know of the newly created petition. Hopefully the writer's acquaintances will further sign and disseminate the petition, thus contributing to the success of the campaign.

We formalise the protocol that runs amongst the petition service, the petition writer, and the signatories. The protocol can be divided in two phases—*setup* and *promotion*—separated by a submit operation. During the setup phase, if an unbounded number of data (titles, descriptions, and dates) is reaching the server by an arbitrary order, then it must be tagged, for otherwise servers would face difficulties in distinguishing, say, two consecutive titles from a description followed by a title. Therefore writers start by selecting one particular operation:

setTitle for setting the title, *setDate* for setting the deadline, *submit* for moving to the promotion phase, or even setting the description of the situation (omitted henceforth). *Selecting* one particular operation is described by a type of the form $\oplus\{\mathit{setTitle}: ..., \mathit{setDate}: ..., \mathit{submit}: ...\}$, where the dots replace the protocol description after the selection operation.

Following a *setTitle* operation, writers are supposed to send the actual title (a string); similarly a date is supposed to follow a *setDate* selection. *Output* of a string is denoted by $!\mathsf{string}$, so that our protocol now looks like this: $\oplus\{\mathit{setTitle}: !\mathsf{string}..., \mathit{setDate}: !\mathsf{date}..., \mathit{submit}: ...\}$. After successful uploading the title or the date, writers are given the chance to revise their data or to submit the proposal, so that the protocol returns to the starting point. This requires a recursive type, which we write in the form of an equation.[1]

$$\mathsf{Petition} = \oplus\{\mathit{setTitle}: !\mathsf{string}.\mathsf{Petition}, \mathit{setDate}: !\mathsf{date}.\mathsf{Petition}, \mathit{submit}: ...\}$$

As discussed above, a submission operation may result in acceptance or rejection. We could model such an outcome with a boolean value, but the ensuing protocol crucially depends on this outcome: move to the promotion phase or halt. As such the writer must expect the selection of an operation—*accepted* or *denied*—coming from the server. Dually, the writer must *offer* the server a menu composed of the two operations; we write such a type as $\&\{\mathit{accepted}: ..., \mathit{denied}: ...\}$.

If denied then the writer receives a reason in the form of a string and the protocol terminates (the writer may try again, but on a different run of the protocol). If accepted then the protocol moves to the promotion phase. *Input* is denoted by ?, so that receiving a string becomes $?\mathsf{string}$; we represent protocol termination by **end**. Here is our type so far:

$$\begin{aligned}\mathsf{Petition} = \oplus\{&\mathit{setTitle}: !\mathsf{string}.\mathsf{Petition}, \mathit{setDate}: !\mathsf{date}.\mathsf{Petition},\\ &\mathit{submit}: \&\{\mathit{accepted}: \mathsf{Promotion}, \mathit{denied}: ?\mathsf{string}.\mathbf{end}\}\}\end{aligned}$$

During the promotion phase all one can do is to sign the petition by sending a signature (in the form of a string), so that we have

$$\mathsf{Promotion} = !\mathsf{string}.\mathsf{Promotion}$$

We have hinted above that the protocol is to be run amongst the petition service, the petition writer, and the signatories. If we assume that no two consecutive operations (input, output, branch, or select) in a protocol are atomic by default, then some care must be exerted on how many partners may know the protocol medium at each time. If this medium is disseminated in the setup phase, then

[1]The attentive reader certainly noticed that writers may submit without first uploading the petition details. We assume that there is a default title ("Save me") and a default deadline (1/1/1970). We leave it as an exercise to refine the type in such a way that a) the *submit* button is pressed only after the *setTitle* and the *setDate* have been pressed at least once each, and b) each of these two last buttons may be pressed more than once.

race conditions may arise when, e.g., the server receives a commit operation from one partner, followed by a *setDate* from a different, unsynchronized partner. On the other hand, during the promotion phase, the success of the petition crucially depends on dissemination, so that we want the protocol medium shared by an unbounded number of potential promoters (signatories and disseminators).

Towards this end we qualify each operation in a type with one of two qualifiers: **lin** denotes a linear operation; **un** denotes an unrestricted, or shared, operation. When the protocol is in a **lin** state then the programming language must guarantee that exactly two partners (server and writer) know the protocol medium; when in an **un** state then an unbounded number of partners (potentially zero) may have access to the medium. Our fully qualified type is thus:

$$\begin{aligned}\mathsf{Petition} &= \mathbf{lin}\oplus\{\mathit{setTitle}\colon \mathbf{lin}!\mathsf{string}.\mathsf{Petition},\ \mathit{setDate}\colon \mathbf{lin}!\mathsf{date}.\mathsf{Petition},\\ &\qquad \mathit{submit}\colon \mathbf{lin}\&\{\mathit{accepted}\colon \mathsf{Promotion},\ \mathit{denied}\colon \mathbf{lin}?\mathsf{string}.\mathbf{lin}\ \mathbf{end}\}\}\\ \mathsf{Promotion} &= \mathbf{un}!\mathsf{string}.\mathsf{Promotion}\end{aligned}$$

There is one last question that we must answer. How do petition writers and petition servers initiate a particular run of the protocol? Petition servers are usually installed on well-known names. It is on one such name that writers and servers agree on initiating a new run of the petition protocol. The protocol is itself started from a small bootstrap protocol, where the server provides the writer with a fresh $\mathsf{Petition}$. If the only thing the server does is to start new $\mathsf{Petition}$ protocols, then its type is $\mathsf{Server} = \mathbf{un}!\mathsf{Petition}.\mathsf{Server}$ which we abbreviate to $*!\,\mathsf{Petition}$.

So far we have been looking at the protocol from the point of view of writers and promoters. How do things look like when seen from the server side? In order to comply with the writer's expectations, servers must start by offering a menu composed of operations *setTitle*, *setDate*, and *submit*, which we write as $\mathbf{lin}\&\{\mathit{setTitle}\colon ...,\ \mathit{setDate}\colon ...,\ \mathit{submit}\colon ...\}$. After a *setTitle* operation the server must input a string ($?\mathsf{string}$); after *setDate*, it is time to input a date ($?\mathsf{date}$). After *submit*, the server must select one of the two operations—*accepted* or *denied*—on the client, which we write as $\mathbf{lin}\oplus\{\mathit{accepted}\colon ...,\ \mathit{denied}\colon ...\}$. When *denied*, then the service must output a string and terminate with **lin end**; the server and the client terminate the protocol together. It should by now be clear that, in order for communication to run smoothly among the various partners involved, when one says output ($!$), the other says input ($?$), when one says select ($\oplus$), the other says branch ($\&$), and when one says terminate (**end**) so does the other. The un/lin qualifiers must match in each case. The types constructed in this way are said to be *dual*.

Session types. The types that describe our protocols are generated by the grammar in Figure 1, where we use letter p to denote an unqualified (or pre-) type, and letter T to describe a type. Recursive types are required to be contractive, that is,

$$\begin{array}{llr@{\qquad}llr}
q & ::= & \textit{Qualifiers:} & & !T.T & \text{send} \\
 & \mathsf{lin} & \text{linear} & & \oplus\{l_i\colon T_i\}_{i\in I} & \text{select} \\
 & \mathsf{un} & \text{unrestricted} & & \&\{l_i\colon T_i\}_{i\in I} & \text{branch} \\
p & ::= & \textit{Pretypes:} & T & ::= & \textit{Types:} \\
 & \mathsf{unit} & \text{unit} & & q\,p & \text{qualified pretype} \\
 & \mathsf{end} & \text{termination} & & a & \text{type variable} \\
 & ?T.T & \text{receive} & & \mu a.T & \text{recursive type}
\end{array}$$

Figure 1: The syntax of types

$$\overline{q\,?T.U} = q\,!T.\overline{U} \qquad \overline{q\,!T.U} = q\,?T.\overline{U} \qquad \overline{q\,\mathsf{end}} = q\,\mathsf{end}$$
$$\overline{q\,\oplus\{l_i\colon T_i\}_{i\in I}} = q\,\&\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{q\,\&\{l_i\colon T_i\}_{i\in I}} = q\,\oplus\{l_i\colon \overline{T_i}\}_{i\in I}$$
$$\overline{\mu a.T} = \mu a.\overline{T} \qquad \overline{a} = a$$

Figure 2: The dual function on types

containing no subexpression of the form $\mu\, a_1 \dots \mu\, a_n.a_1$. The equations introduced above are transformed into recursive types in the standard way:

Petition = **rec** a.**lin**$\oplus${*setTitle*: !string.a, *setDate*: !date.a, *submit*: ...}

In the presence of recursive types, we define *type equality* as the equality of the regular infinite trees obtained by the infinite unfolding of recursive types. The formal definition, which we omit, is co-inductive. In this way we can use types **un**!string.**rec** a.**un**!string.a and **rec** a.**un**!string.**un**!string.a interchangeably, in any mathematical context. This allows us never to consider a type $\mu a.T$ explicitly (or a for that matter). Instead, we pick another type in the same equivalence class, namely the type obtained by replacing in T occurrences of type variable a by type $\mu a.T$, usually written $T[\mu a.T/a]$. If the result of the process turns out to start with μ, we repeat the procedure. Contractiveness ensures the termination of the unfolding process. In other words, we take an *equi-recursive* view of types [11].

Rather than providing a co-inductive definition of duality, we start by defining a function from types to types as in Figure 2. Then, to check that a given type T_1 is dual of another type T_2, we first build the dual of T_1 and then check that the thus obtained type is equivalent to T_2. For example, to show that type **rec** a.**un**!string.**un**!string.a is dual to **rec** b.**un**?string.b, we first build type **rec** a.**un**?string.**un**?string.a, dual of the former and then check that it is equivalent

Context split

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma}{\Gamma, x\colon \mathsf{un}\, p = (\Gamma_1, x\colon \mathsf{un}\, p) \circ (\Gamma_2, x\colon \mathsf{un}\, p)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = (\Gamma_1, x\colon \mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x\colon \mathsf{lin}\, p)}$$

Context update

$$\Gamma = \Gamma + \emptyset \quad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x\colon T = \Gamma_1 + (\Gamma_2, x\colon T)} \quad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x\colon \mathsf{un} p = (\Gamma_1, x\colon \mathsf{un} p) + (\Gamma_2, x\colon \mathsf{un}\, p)}$$

Figure 3: Context split and context update

to the latter. Duality is defined on session types only; it does not apply to the unit type. Would we require boolean or function types, say, duality would not be defined on them either.

Programs are usually typed against a context describing the types for the free identifiers. *Typing contexts* are finite maps Γ from identifiers (or variables, denoted by x) to types. Symbol $\emptyset$ indicates an empty map. Given an arbitrary map Γ and a variable not in the domain of Γ, we denote by $\Gamma, x\colon T$ the map equal to Γ everywhere except at x where $\Gamma(x) = T$. We maintain the linearity invariant through the standard linear context splitting operation. When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in two and passing a different part to each process. Figure 3 defines the context splitting relation $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that in the third rule, x is not in Γ_2 otherwise it would be in $\Gamma = \Gamma_1 \circ \Gamma_2$ and the result $\Gamma, x\colon \mathsf{lin} p$ would not be defined, and similarly for the last rule and Γ_1.

Unlike conventional linear values that are consumed once they become used, values that describe the medium on which protocols run are consumed piecewise: an input on a medium of type $q?T_1.T_2$ renders the *same* medium at type T_2. We introduce a *context update* operation for the effect (Figure 3). If q is lin then, by virtue of context splitting, reading the medium removes its type $q?T_1.T_2$ from the context, while context update adds the continuation type T_2 (second rule). If, on the other hand, q is un then the type remains in the context, and we must add a type T_2 equivalent to $q?T_1.T_2$, according to the definition of context update (third rule). Since we want to add the continuation T_2, it must be the case that T_2 is equivalent to $\mathsf{un}?T_1.T_2$, which can happen if, e.g., T_2 is of the form $\mu a.\mathsf{un}?T_1.a$. This form of type is so common that we introduce an abbreviation for it, $*?T_1$, as we have seen.

Linear type systems follow an invariant whereby unrestricted data structures may not include linear data structures. This is usually accomplished by defining two predicates un and lin that operate both on types and on contexts. The rules state that linear data structures can hold objects of a linear or unrestricted nature, but that unrestricted data structures can only contain unrestricted values. Making $q \sqsubseteq q'$ the smallest reflexive such that $\mathsf{lin} \sqsubseteq \mathsf{un}$ we define [15]:

- $q(T)$ when $T = q'p$ and $q \sqsubseteq q'$
- $q(\Gamma)$ when $x\colon T \in \Gamma$ implies $q(T)$

Notice that in particular $\mathsf{lin}(T)$ is true for any T, and similarly for contexts.

3 Session types in the pi-calculus

We now embody the types as described in the previous section in a message passing concurrent language, the pi-calculus [10]. Our medium of communication (as we put it in the previous section) is channels where messages flow. A channel at a linear type is held by exactly two threads; a channel at an unrestricted type is held by zero or more threads.

The running example in the pi calculus. We assume a petition server installed at the well-known channel ps. Petition writers read from this channel a petition channel p (line 3, Figure 4). Our writer starts by setting the deadline, then the title[2], and finally decides to adjust the deadline (lines 4–6). Once happy with the information provided, the writer submits the request, and waits for an answer (lines 7–8). If accepted, then the writer distributes the petition channel to its two acquaintances (Signatory1 and Signatory2), and signs the petition himself (lines 9–11); if denied then the writer receives the reason for denial (in the form of a string), closes the channel end point and terminates (lines 13–14).

The three acquaintances have different behaviours. The first gets channel p and signs the petition (lines 17–18); the second further sends the petition to another potential signatory (Signatory3) and signs (line 21). Finally, Signatory3 gets the channel but decides neither to sign nor to further disseminate (lines 23–24).

Let us have a look at the server side. Our implementation is divided into three components: the Server itself that creates and sends new channels on ps; the Setup that gathers the information on a particular petition, and the Promotion that collects the various signatures. Petition servers must provide (on well-known name ps) fresh linear petition channels. This is accomplished in the pi-calculus with a **new**-constructor that creates a fresh channel, different from all others (line 3). One end

[2]"Save the Iberian wolf", *Canis lupus signatus*.

```
1  SaveTheWolf :: ∗?Petition
2  SaveTheWolf ps =
3    ps?p.
4      p ◁ setDate. p!(31,12,2010).
5      p ◁ setTitle. p!"Save the Wolf".
6      p ◁ setDate. p!(31,12,2100).
7      p ◁ submit.
8      p ▷ {accepted:
9            Signatory1 p |
10           Signatory2 p |
11           p!"me"
12         denied:
13           p?x.
14           close p
15         }
16 Signatory1 :: ∗! string
17 Signatory1 p =
18   p!"signatory1"
19 Signatory2 :: ∗! string
20 Signatory2 p =
21   Signatory3 p | p!"signatory2"
22 Signatory3 :: ∗! string
23 Signatory3 p =
24   inaction
```

```
1  Server :: ∗! Petition
2  Server ps =
3      (new p1 p2)
4      ps!p2.(
5        Setup (p1,(1,1,1970),"Save me") |
6        Server ps)
7  Setup :: Petition ∗ date ∗ string
8  Setup (p, d, t) =
9    p ▷ {setDate: p?d'.Setup (p, d', t),
10       setTitle: p?t'.Setup (p, d, t'),
11       submit: p ◁ accepted.
12               Promotion (p, [])
13       }
14 Promotion :: ∗?string ∗ stringList
15 Promotion (p, l) =
16     p?s.Promotion (p, s :: l)
```

```
1  Main =
2    (new ps1 ps2)
3    Server ps1 |
4    SaveTheWolf ps2
```

Figure 4: Petition example in the pi-calculus

of this channel, denoted by p2, is passed to potential writers (line 4); the other end, called p1, is passed to process Setup, together with the default deadline and title (line 5).

Process Setup receives the petition channel, the default deadline, and the default title, and interactively updates the last two (lines 9–10). Our simplistic server accepts each single petition (line 11). The protocol now passes to the promotion phase, by providing the Promotion process with the petition channel p and an empty list, where the signatory names are to be stored. In order to simplify the example, we use a data type for lists, where [] denotes the empty list and s :: l denotes a list composed of an element s at the head and a list l at the tail. Such a data type would have to be encoded in the base language [9, page 106]. Process Promotion receives a signature s on channel p, stores it in the list (s :: l) and recurs.[3]

Our Main process creates a channel and distributes one of its ends (ps1) to process Server and the other end (ps2) to the petition writer, SaveTheWolf. In

[3]Notice that, in the example, we use symbol :: both as list concatenation and to introduce types in processes.

Typing rules for values

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash ()\colon q\,\mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad (\text{T-Unit},\text{T-Var})$$

Typing rules for processes

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{inaction}} \qquad \frac{\Gamma \vdash x\colon \mathsf{lin}\,\mathsf{end}}{\Gamma \vdash \mathsf{close}\ x} \qquad \frac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \frac{\Gamma, x\colon T, y\colon \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}$$

$$(\text{T-Inact},\text{T-Close},\text{T-Par},\text{T-Res})$$

$$\frac{\Gamma_1 \vdash x\colon q\,!T_1.T_2 \qquad \Gamma_2 \vdash v\colon T_1 \qquad \Gamma_3 + x\colon T_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!v.P} \qquad (\text{T-Out})$$

$$\frac{\Gamma_1 \vdash x\colon q\,?T_1.T_2 \qquad (\Gamma_2, y\colon T_1) + x\colon T_2 \vdash P \qquad q(\Gamma_2)}{\Gamma_1 \circ \Gamma_2 \vdash q\,x?y.P} \qquad (\text{T-In})$$

Figure 5: Typing rules for the pi-calculus

the previous section we discussed the type of the well-known name where new petitions are to be requested: $*!$ Petition or $*?$Petition, depending on the point of view; these are the types of the two ends of the newly created channel, ps1 and ps2, respectively.

For the client, channel ps carries p typed at type Petition. The initial, linear, part of the channel is consumed in lines 3–8 of process SaveTheWales. When control reaches line 9, channel p is of type $*!$string, allowing it to be freely passed around and used for signing (lines 9-11, as well as processes Signatory1, Signatory2, and Signatory3). On the server side, the initial linear part of the type dual of to Petition is consumed in process Setup, whereas the unrestricted part ($*?$string) is used in process Promotion.

Typing pi processes. The typing rules for the pi-calculus are in Figure 5. We omit the grammar of values and processes, which can be easily inferred from the rules. Typing judgements for values are of the form $\Gamma \vdash v\colon T$, indicating that value v has type T under context Γ, as usual. Judgements for processes are of the form $\Gamma \vdash P$, testifying that process P is well typed under context Γ.

We briefly comment on the rules. The rules for values and for process inaction make sure that the unused typing context Γ is unrestricted, thus ensuring that linear values are completely consumed. The rule for closing channel ends, T-Close, requires a channel ready to be closed: linear (no other process may know it) and at end (the protocol on the channel is completed). The rule for parallel composition, T-Par, splits the incoming context in two and passes each part to a different process. The rule for name restriction, T-Res, simply adds two dual types to the context, each will type one end of the newly introduced channel. The rule for output processes of the form $x!v.P$, T-Out, splits the context in three parts, one to type each of the constituents of the process. The continuation P is typed at context Γ_3 updated with x at the continuation type T_2. This means that either $q = \mathsf{lin}$, and hence x is not in Γ_3, or $q = \mathsf{un}$ and in this case T_2 must be equal to $\mathsf{un}!T_1.T_2$.

The last rule in the figure, T-In, accounts for both simple (linear) or replicated (unrestricted) input. In the pi-calculus one traditionally indicates a perennial process by prefixing it with an exclamation mark. Since such a symbol was taken in our language to designate output, we then take the chance to align the syntax of the language with that of types. We write an ephemeral input process as $\mathsf{lin}x?y.P$ and a persistent (replicated) input process as $\mathsf{un}x?y.P$. The $q(\Gamma_2)$ proviso in the rule for input makes sure that the variables in the body of a replicated process are unrestricted themselves.

In the presence of free output it is well-known [16] that we must differentiate the two ends of a channel. Several techniques are known: annotating channel ends with distinct, $+, -$, polarities [6], using type constructors that can talk about the ends of a same channel [7], or work with a *double binder* that binds together distinct identifiers for the two ends of a same channel [14]. Here we follow the last method.

We have lived two decades without requiring a close operation for the pi-calculus. Why now? The truth is we do not strictly need it. Session types that start linearly may terminate as un end, but then the runtime has to run a potentially expensive garbage collection procedure in order to deallocate the data-structures necessary to implement channel operations. On the other hand, session types that terminate in lin end allow a close operation to explicitly deallocate the supporting data-structures. Channels of type un end, even though they cannot be used for communication, can still be freely passed around (and stored in data structures) but cannot be deallocated for there is an undetermined number of references to it.

We leave as an exercise the design of the rules for the branch and the selection processes (the interested reader may want to check a solution in reference [14]).

Back to the example. The code in Figure 4 omits the **lin** qualifier in input prefixes. The **un** input prefix qualifier cannot be found for we have used recursive definitions in place of replication. The encoding of recursion into replication is standard [9, page 94]. For example, the Promotion definition in lines 15–16 of the server is transformed into **un** promotion?(p,l). **lin** p?s.promotion!(p,s::l), whereas the call in line 12 becomes promotion!(p ,[]) . Finally, a (**new** promotion) binder as well as the replicated process itself must be placed at, say, the top level process, line 2 of Main.

In the example we use *polyadic* messages, as opposed to the *monadic* messages proposed in the typing rules where messages can carry exactly one value. This happens, for example, in the expansion of the Promotion process described above. Once again the polyadic-to-monadic encoding is well-known [9, page 93] (cf. [14]). To *atomically* send the two arguments, p and [], on channel promotion, we translate message promotion!(p ,[]) into a process that implements a simple protocol: (**new** c1,c2)promotion!c2.c1!p.c1![].**close** c1. Such a process creates a new channel; one end, c2, is sent to the client, the other, c1, is used to transmit the two arguments. On the other hand, the receiving side, **un** promotion?(p,l).P is translated into process **un** promotion?c.**lin** c?p.**lin** c?l.**close** c.P that receives a fresh *linear* channel on which the two parameters may be received without risk of interference.

The type of channel promotion can be precisely captured by our types. Because there is a replicated receptor installed at the channel, the type takes the form $*$?T, as seen from the point of view of the receiver. Type T describes the little (linear) protocol used to receive the two parameters, namely **lin**?($*$?string). **lin**? stringList . **lin** **end**.

4 Session types in a functional language

We now address the design of a call-by-value functional multi-threaded programming language. We add to a linear functional programming language [15] a notion of channels, akin to that described in the previous section for the pi-calculus. In addition to the lambda-calculus constructors—basic values, variables, abstraction, application and pairs—we rely on operations for channel creation, sending/receiving/selecting/branching on a channel, as well as for forking new threads. For the new constructs, we stick as much as possible to the syntax of the previous section.

The running example in a functional language. Because we use the same syntax for channel operations, the code for the client, Figure 6, should be easy to follow. The main difference with respect to the pi-calculus code is that, once (and if) the petition request is accepted by the server, the writer forks two threads, one for each signer (lines 9–10). Rather than sending p on a channel (known to the writer and to a signer), the channel is passed as a parameter to the function. To align our language with the expectations imposed by functional programming, we allow writing x? to receive a value on a channel x, without explicitly mentioning the variable that will hold the value, nor the term that constitutes the continuation. In our example, the petition writer simply discards the reason for denial (line 14).

```
saveTheWolf :: ∗?Petition → unit
saveTheWolf ps =
  let p = ps? in
  p ◁ setDate; p!(31,12,2010);
  p ◁ setTitle; p!"Save the Wolf";
  p ◁ setDate; p!(31,12,2100);
  p ◁ submit;
  p ▷ {accepted:
        fork (signer1 p);
        fork (signer2 p);
        p!"me"
      denied:
        p?;
        close p
      }
signer1 :: ∗!string → unit
signer1 p =
  p!"signer1"
signer2 :: ∗!string → unit
signer2 p =
  fork (signer3 p);
  p!"signer2"
signer3 :: ∗!string → unit
signer3 p = ()
```

```
petitionServer :: ∗!Petition → unit
petitionServer ps =
  split new Petition as p1, p2 in
  ps!p1;
  fork (setup p2 (1,1,1970) "Save me");
  petitionServer ps
setup :: dual(Petition) → date →
  string → unit
setut p d t =
  p ▷ {setDate: setup p (p?) t,
      setTitle: setup p d (p?),
      submit: p ◁ accepted;
              promotion p []
      }
promotion :: ∗?string →
  stringList → unit
promotion p l =
  promotion p ((p?)::l)
```

```
main :: unit → unit
main _ =
  split new ∗!Petition as ps1, ps2 in
  fork (petitionServer ps1);
  fork (saveTheWolf ps2)
```

Figure 6: Petition example in a functional language

Typing functional terms. We need one more type for functions; more precisely one pretype $p \to p$, which we add to those in Figure 1. As discussed in Section 2, duality is not defined on this type. Figure 7 presents the typing rules for the language. Once again, apologising for the inconvenience, rather than presenting the syntax we ask the reader to read it from the terms in the conclusion of the rules. Typing judgements are of the form $\Gamma_1 \vdash M\colon T; \Gamma_2$ conveying the idea that term M has type T under context Γ_1. The "continuation" context Γ_2 describes the residual types of the variables *used* in M for channel operations (input, output, receive, select). If T is a type of the form $\mathsf{lin?un\,unit}.T'$, we have:

$$x\colon T \vdash x?\colon \mathsf{un\,unit}; x\colon T'$$
$$x\colon T \nvdash ()\colon \mathsf{un\,unit}; x\colon T$$

The main challenge in the design of a type system for a functional language with session types is typing input and output operations without explicitly mentioning the continuation. In other words, we want to type terms $x?$ and $x!$ alone. In

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T; \emptyset} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash ()\colon q\,\mathsf{unit}; \emptyset} \qquad \frac{\mathsf{un}(\Gamma) \qquad q(T)}{\Gamma \vdash \mathsf{new}\,T\colon q(T,\overline{T}); \emptyset}$$

(T-Var,T-Unit,T-New)

$$\frac{\Gamma_1 \vdash x\colon q?T_1.T_2; \emptyset \qquad \mathsf{un}(\Gamma_2)}{\Gamma_1 \circ \Gamma_2 \vdash x?\,:\, T_1; \Gamma_2 + x\colon T_2} \qquad \frac{\Gamma_1 \vdash x\colon q!T_1.T_2; \emptyset \qquad \mathsf{un}(\Gamma_2)}{\Gamma_1 \circ \Gamma_2 \vdash x!\colon \mathsf{lin}(T_1 \to \mathsf{un}\,\mathsf{unit}); \Gamma_2 + x\colon T_2}$$

(T-In,T-Out)

$$\frac{\Gamma_1, x\colon T_1 \vdash M\colon T_2; \Gamma_2 \qquad q(\Gamma_1) \qquad \mathsf{un}(\Gamma_2)}{\Gamma_1 \vdash q\lambda x.M\colon q\,T_1 \to T_2; \emptyset}$$ (T-Abs)

$$\frac{\Gamma_1 \vdash M_1\colon q\,T_1 \to T_2; \Gamma_3 \qquad \Gamma_2 \vdash M_2\colon T_1; \Gamma_4}{\Gamma_1 \circ \Gamma_2 \vdash M_1 M_2\colon T_2; \Gamma_3 + \Gamma_4}$$ (T-App)

$$\frac{\Gamma_1 \vdash M_1\colon T_1; \Gamma_3 \qquad \Gamma_2 + \Gamma_3, x\colon T_1 \vdash M_2\colon T_2; \Gamma_4}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2\colon T_2; \Gamma_4} \qquad \frac{\Gamma_1 \vdash M\colon \mathsf{un}p; \Gamma_2 \qquad \mathsf{un}(\Gamma_2)}{\Gamma_1 \vdash \mathsf{fork}\,M\colon \mathsf{un}\,\mathsf{unit}; \emptyset}$$

(T-Let,T-Fork)

Figure 7: Typing rules for the functional language

our language, the continuation, if present at all, comes in the form of an abstraction or of the second component of a pair, and we would not like to mention them explicitly in the typing rules. We solve the problem by adding an extra context at the right hand side of judgements.

Equipped with such judgements, the rule for the input expression (T-In) acts effectively as an elimination rule for type $q?T_1.T_2$, where the type of expression $x?$ is T_1 and the context available to the continuation contains x at type T_2. Similarly, the rule for the output process (T-Out) works as an elimination rule for type $q!T_1.T_2$, where the type of expression $x!$ is understood as a (linear) function receiving T_1 (and delivering $\mathsf{un}\,\mathsf{unit}$) and where the continuation sees x at type T_2. In both rules the context available to the continuation is $\Gamma_2+x\colon T_2$, thus ensuring that, when in presence of a unrestricted T_2 (occurring both in Γ_1 and Γ_2), the type of x available to the continuation is equal to the initial type ($\mathsf{un}?T_1.T_2$ or $\mathsf{un}!T_1.T_2$), that is T_2 is, e.g., of the form $\mu a.\mathsf{un}?T_1.a$ (cf. rules T-In and T-Out in the type system for the pi-calculus, Figure 5).

In the rule for abstraction (T-Abs), if the qualifier of the function is un (meaning that the function can be used multiple times), then all free variables in the body of the function (hence in Γ_1) must equally be of unrestricted types (cf. the situation of the replicated input in Figure 5). In any case, functions must consume all their linear resources (unlike the system in reference [5]), as enforced by the proviso $\mathsf{un}(\Gamma_2)$. Being values, functions leave no linear values to be consumed by the continuation, as witnessed by the final empty context.[4] The rule for function application (T-App) splits the incoming context in two parts, Γ_1 and Γ_2, made available to its two parts M_1 and M_2. If function M_1 leaves linear values Γ_3 to be consumed by the continuation (as it happens in case $x!$), these are added to those of the argument, Γ_4, and made available to the continuation of the application.

What we do not know is how to type a general input or output expression, $M!$ or $M?$, for we need to get hold of the channel reference x in order to advance its type. For both cases we provide a let construct: rather than writing $M!$, we write $\mathsf{let}\ x = M\ \mathsf{in}\ x!$, and in place of $M?$, we write $\mathsf{let}\ x = M\ \mathsf{in}\ x?$. Very much like rule T-App, rule T-Let splits the incoming context in two parts, one for each subterm. The difference is that the linear values not completely consumed by M_1 (present in Γ_3) are added to Γ_2, in order to type the body M_2, whereas in the case of rule T-App they are added directly to the final context. The linear values not completely consumed by M_2 (in Γ_4) are those provided to the continuation of the let. In the same vein, we cannot directly type term $(x!)(x?)$ since we would be typing subterm $x!$ first where we expect subterm $x?$ to be evaluated first. Once again, the let-construct allows to make explicit the evaluation order, $\mathsf{let}\ y = x?\ \mathsf{in}\ (x!)y$.

The fork construct can be intuitively described by a type of the form $\mathsf{un}(\mathsf{un}\, p \to \mathsf{un}\,\mathsf{unit})$, more precisely by a type schema, ranging over all pre-types p. The $\mathsf{un}(\Gamma_2)$ proviso together with the $\mathsf{fork}\, M$ unrestricted type, $\mathsf{un}p$, makes sure that the forked expression M consumes all its linear resources.

Semicolon is as usual a derived construct. But we must use let, rather than abstraction. Expression $(); x?$ cannot be translated as $(p\lambda\_.x?)()$ for it would not be typable under context $x\colon \mathsf{lin?unit.lin}\, p$ (the function does not consume all its linear resources). Instead we use $\mathsf{let}\ \_ = ()\ \mathsf{in}\ x?$ which allows to leave x at type $\mathsf{lin}\, p$ to the continuation.

We have seen that functions must consume all their linear resources. This means that we cannot type a function of the form $\lambda x.x!5$ with x of the type $\mathsf{lin?unit.lin}\, p$. The alternative is for the function to return all its unused linear resources, as in $\lambda x.(x!5; x)$ which can be typed as $\mathsf{lin}(\mathsf{lin!nat.lin}p \to \mathsf{lin}p)$. Similarly, rather than writing $\lambda x.x?$, we must write $\lambda x.(x?, x)$ which can be typed as $\mathsf{lin}(\mathsf{lin?nat.lin}p \to (\mathsf{nat} * \mathsf{lin}p))$, following the approach described in reference [4].

We leave as an exercise deriving the typing rules for pair construction and deconstruction; they can be easily adapted from reference [15, page 10], as well as adapting the rule for closing channel ends. We equally leave as an exercise the rules for branch and selection. The reader may as well consider adding a fixed point operator to the language, without which the server in Figure 6 would not be typable.

[4]Context Γ_2 would as well do since $\Gamma_1 + \Gamma_2 = \Gamma_1$ when $\mathsf{un}(\Gamma_2)$ and $\Gamma_1 \circ \Gamma_2$ defined.

Back to the example. Following our convention, we have omitted all un qualifiers in code in Figure 6. The functions that compose the client are used only once, they could easily be typed at a lin type as well. Those for the server are recursive; they must be unrestricted. In order to comply to the restriction on function calls, whereby a channel cannot be used both in the function and in the argument, code must be adapted. For example, the function call on server, line 18, becomes **let** s = p? **in** promotion p (s :: l).

5 Session types in an object-based language

We now incorporate session types in a conventional class-based imperative programming language. The language features multithreaded concurrency, where different threads communicate solely by calling methods in remote objects. We thus see that the *communication medium*, identified in Section 2 and embodied as communication channels in the pi calculus and in our functional language (Sections 3 and 4), is instantiated here as *object references*. Whereas in channel-based languages, processes communicate by exchanging messages on (session governed) channels, in our object-oriented language threads communicate solely by calling methods on (session governed) object references. This option clearly contrasts with more conventional approaches that add communication channels to an object-oriented language (e.g., [3, 5]).

The running example in a language of objects. The code for our running example can be found in Figure 8. Composed of four classes: SaveTheWolf, PetitionServer, Signatory, and Main, we try to follow as close as possible the architecture of the solutions in previous sections.

While we could force, via a suitable encoding, the Petition session type as identified in Session 2 into our language, we seek a natural incorporation of the protocol into familiar OO concepts. With respect to session types, method calls bring some restrictions as well facilities. A *selection* operation (previously identified with a left triangle, $\triangleleft$) can be identified with a method call, while an *output* operation can only be identified with argument passing within a method call. An *input* operation can only occur as the result of a method call. What about *branching*? How can a target object force a branch on a client? For a simple binary branch, we could stipulate that boolean methods would force such a test, via conditional expressions. For more general branching structures we use conventional enumerations (**enum**) and a **switch** construct.

In object-oriented languages, method call and argument passing are usually interpreted as a single atomic operation, this means that, in our setting, selection followed by output is also atomic and we can take advantage of this situation to

```
enum Answer = {accepted, denied}
class SaveTheWolf {
  usage lin&{init: lin&{run: un end}};
  Petition  p;
  Signatory[Sign] signatory1;
  Signatory[Sign] signatory2;
  unit  init(PetitionServer s,
        Signatory[Sign] s1,
        Signatory[Sign] s2) {
    p = s.newPetition();
    signatory1 = s1;
    signatory2 = s2;
  }
  unit  run() {
    p.setDate
      (new Date(31, 12, 2010));
    p.setTitle("Save the Wolf");
    p.setDate
      (new Date(31, 12, 2100));
    switch (p.submit()) {
      case Answer.accepted:
        fork signatory1.signPlease(p);
        fork signatory2.signPlease(p);
        p.sign("me");
      case Answer.denied:
        free  p;
    }
  }
}
class Signatory {
  usage lin&{setName: Sign} where
  Sign = un&{signPlease: Sign};
  string  name;
  unit  setName(string n) {
    name = n;
  }
  unit  signPlease
    ( Petition [Promotion] p) {
    p.sign (name);
  }
}
```

```
class PetitionServer {
  Petition  newPetition()  {new Petition ();}
}
class Petition  {
  usage Setup where
  Setup = lin&{setTitle: Setup,
        setDate: Setup,
        submit: lin⊕{accepted: Promotion,
                     denied: lin end}}
  Promotion = un&{sign: Promotion,
                  howMany: Promotion};
  string   title   = "Save me";
  Date date = new Date(1,1,1970);
  List signatures = new List();
  unit  setTitle(string t) { title = t; }
  unit  setDate(string d) { date = d; }
  Answer submit() { Answer.accepted; }
  sync unit sign(string name) {
    signatures.add(name);
  }
  int  howMany() { signatures.length(); }
}
```

```
class Main {
  unit  main() {
    PetitionServer  server =
      new PetitionServer();
    Signatory s1 = new Signatory();
    s1.setName ("signatory1");
    Signatory s2 = new Signatory();
    s2.setName ("signatory2");
    SaveTheWolf wolf =
      new SaveTheWolf();
    wolf.init(server, s1, s2);
    fork  wolf.run();
  }
}
```

Figure 8: Petition example in an object-based language

simplify interaction in unrestricted mode (see below). Having mapped the four main operations of session types into OO concepts, it remains to discuss how to enforce protocols running on object references: we use for the effect a **usage** annotation in classes.

Let us then analyse the code. In order to model the possible outcome of a *submit* operation we setup an enumeration in line 1, Figure 8. This time we start by describing the server side. Class PetitionServer has one single purpose: to create Petition references, and it does this whenever invoked at method newPetition (line 2). All our classes are equipped with **usage** annotations, even if inserted by the compiler. In this case, method newPetition is repeatedly available, hence the (implicit) annotation is Init **where** Init = **un**&{newPetition: Init}, which we abbreviate to $*$&{newPetition}, following the schema introduced in Section 2.

Class Petition follows the protocol described by its **usage** clause, lines 5–11. With respect to type Petition, Section 2, we see that input (?) and output (!) are not explicitly present for they can be read from method signatures. We take the chance to make the protocol a little more realistic, by allowing, in the Promotion phase and in addition to operation *sign*, an operation *howMany* to obtain the number of signatures obtained so far. This is made possible by the fact that branch followed by input is a natural atomic operation in object-oriented languages. This is what we meant above by the extra flexibility provided by method calls.

The interesting part of the Petition class is exactly the (unrestricted) Promotion phase, involving methods *sign* and *howMany*, where the object may be held by multiple signatories. Being in an unrestricted phase, the object usage is of type $*$&{*sign*,*howMany*}, an abbreviation for the type in lines 10–11. But because the object may be held by multiple threads, it is up to the programmer to control concurrency, if so desired. In our case, we allow concurrency in the read method *howMany*, but prevent concurrent accesses to the write method *sign* by prefixing the method name with a **sync** qualifier.

We start the analysis of the client code by reading class Signatory. Method *setName* plays the role of a constructor, as witnessed by **usage lin**&{*setName*: Sign} in line 31. After initialization signatories can be shared while becoming ready to sign multiple petitions, via method *signPlease*. The petition to be signed is given as a parameter of type Petition [Promotion], an abbreviation for the type in lines 10-11, class Petition. This means that the reference received as parameter must have been subject to the Setup phase (lines 6–9, class Petition).

Class SaveTheWolf comprises two methods: *init* and *run*. They are supposed to be run in sequence; in fact *init* plays the role of an object constructor, a notion our language is not provided with. We decided that objects of this class must be initialised only once and *run* only once. This is enforced via the class annotation **usage lin**&{*init*: **lin**&{*run*: **un end**}}, where **end** can now be short for &{}, meaning that no further method is available to threads holding a reference to the object.

Rather than receiving the petition medium over a well-known channel (ps in the previous sections), method *init* asks the petitionServer for one such reference (line 10). The method also accepts two previously initialized signatories of type Signatory[Sign], defined in line 32. Method *run* conducts petition setup. Notice the presence of the **switch** construct to branch accordingly to the result of the *submit* operation (lines 20–26). The subsequent usage of reference p crucially depends on this test: if *accepted* then the protocol moves to the promotion phase, where the reference is distributed to two signatories and used to convey the signature of the petition creator itself (lines 22–24); if *denied* then reference p is at state **lin end** and we use operation **free** to release the memory allocated by the object, since no further operation is possible on the reference (witnessed by pre-type **end**) and p is the only reference to the object (described by the qualifier **lin**). Lines 22–23 fork two threads each running the code of method *signPlease* in class Signatory.

It is instructive to compare the "end" part of the session types for classes SaveTheWolf and Petition, unrestricted for the former, linear in the latter. The only reference to the only object of class SaveTheWolf is wolf in line 9, Main class. The reference is used to fork a thread (running the code of method *run*); if we assume that the fork operation succeeds immediately without waiting for the completion of method *run*, then method main cannot free the object, for its code might still be in use in the thus created thread. The case for reference p of type Petition is different: in line 26 we know that there is no thread running code of the object and may thus release the memory.

Finally, class Main creates a petition server, two signatories and forks a thread to run the SaveTheWolf petition client.

Typing classes. The type checker of our language makes sure that a) client code calls methods in the order specified in the class usage type; b) client code tests method results, if applicable, before proceeding to the next call; and c) references to linear objects are consumed to the end before being freed. Type-checking is modular and performed following a top-down strategy: program checking is conducted by checking each class separately, which in turn conducts the checking of each method within the class *in the order in which it appears in the classe's usage type*. The type system keeps track of the state of each field, each parameter and each local variable. When we call a method m on a field $o.f$, we check that the context associates to $o.f$ a type $q\&\{m\colon T, \dots\}$, and we update the type of the field to T. When switching on the result of a method call we check that the reference on which the method was called is of type $q \oplus \{c_1\colon T_1, \dots, c_n\colon T_n\}$, where $c_1, \dots, c_n$ are the constants in the enumerated type returned by the call. The context for branch c_i is then updated with type T_i. The technical details can be found in references [5, 1]; a prototype is available online.

Acknowledgements. This work was supported by FCT/MCTES via projects PTDC/EIA–CCO/105359/2008 and CMU–PT/NGN44–2009–12. The author would like to thank Joana Campos, Mariangiola Dezani-Ciancaglini, Simon Gay, and Marco Giunti for insightful comments.

References

[1] Joana Campos. Linear and shared objects in concurrent programming. Master's thesis, University of Lisbon, 2010.

[2] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: an overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer-Verlag, 2010.

[3] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. In *Proceeding of ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[4] Simon Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. Subsumes Technical Report 2007–251, University of Glasgow.

[5] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *ACM Symposium on Principles of Programming Languages*, pages 299–312. ACM Press, 2010.

[6] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[7] Marco Giunti and Vasco T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR'10*, volume 6269 of *LNCS*, pages 432–446. Springer, 2010.

[8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[9] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[12] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[13] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.

[14] Vasco T. Vasconcelos. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM 2009)*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer, 2009.

[15] David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.

[16] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *1st International Workshop on Security and Rewriting Techniques*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier, 2007.