Asynchronous Functional Session Types

Simon Gay and Vasco Vasconcelos

UNIVERSITY of GLASGOW

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland

TR-2007-251
May 2007

Asynchronous Functional Session Types

Simon Gay[1] and Vasco Vasconcelos[2]

[1] Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK.
Email: <simon@dcs.gla.ac.uk>
[2] Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa,
1749-016 Lisboa, Portugal. Email: <vv@di.fc.ul.pt>

May 30, 2007

Abstract

Session types support a type-theoretic formulation of structured patterns of communication, so that the communication behaviour of agents in a distributed system can be verified by static type checking. Applications include network protocols, business processes, and operating system services. In this paper we define a multithreaded functional language with session types, which unifies, simplifies and extends previous work. There are three main contributions. First: an operational semantics with buffered channels, instead of the synchronous communication of previous work. Second: session type manipulation by means of the standard structures of a linear type theory, rather than by means of new forms of typing judgement. Third: a notion of subtyping, including the standard subtyping relation for session types (imported into the functional setting) and a novel form of subtyping between standard and linear function types. Our new approach significantly simplifies session types in the functional setting, clarifies their essential features, and provides a secure foundation for language developments such as polymorphism and object-orientation, as well as further forms of static analysis including estimating the size of communication buffers.

Keywords: Session types, functional programming, typechecking, semantics, distributed programming, specification of communication protocols.

1 Introduction

The concept of *service-oriented computing* has transformed the design and implementation of large-scale distributed systems, including online consumer services such as e-commerce sites. It is now common practice to build a system by gluing together the online services of several providers: for example, online travel agents, centralised hotel reservation systems, and online shops. Such systems are characterised by detailed and complex protocols, separate development of components and re-use of existing components, and strict requirements for availability and correctness. In this setting, formal development methods and in particular static analysis are vitally important: for example, the implementor of an online travel agent cannot expect to test against the live booking systems of the airlines.

This paper concerns one approach to static analysis of the communication behaviour of agents in a distributed system: session types [13, 14, 17]. In this approach, communication protocols are expressed as types, so that static typechecking can be used to verify that agents observe the correct protocols. For example, the type

```
S = &⟨service: ?Int.!Int.S, quit:End⟩
```

describes the server's view of a protocol in which the server offers the options service and quit. If the client selects service then the server receives an integer, sends an integer in response, and the protocol repeats. If the client selects quit then the only remaining action is to close the connection. It is possible to statically typecheck a server implementation against the type S, to verify that the specified options are provided and are implemented correctly. Similarly, a client implementation can be typechecked against the dual type $\overline{S}$, in which input and output are interchanged.

Early work on session types used network protocols as a source of examples, but more recently the application domain has been extended to business protocols arising in web services [22] and operating system services [6]. By incorporating correspondence assertions, the behavioural guarantees offered by session types have been strengthened and applied to security analysis [2]. A theory of subtyping for session types has been developed [10] and adapted for specifying distributed software components [18]. Session types can now be regarded as an established concept with a wide range of applications.

The basic idea of session types is separate from the question of which programming language they should be embedded in, although of course a specific system incorporating session types must be based on a particular language. Much of the research has defined systems of session types for pi calculus and related process calculi, but recently there has been considerable interest in session types for more standard language paradigms. Our own previous work [19, 20] was the first proposal for a functional language with session types. Neubauer and Thiemann [15] took a different approach, embedding session types within the type system of Haskell. Session types are also of interest in object-oriented languages; this situation has been studied formally by Dezani-Ciancaglini *et al.* [4] and is included in the work of Fähndrich *et al.* [6].

In the present paper we define a multithreaded functional language with session types, unifying and simplifying several strands of previous work, and clarifying the relationship between session types and standard functional type theory. The contributions of the paper are as follows.

1. We formalize an operational semantics in which communication is buffered, instead of assuming synchronization between send and receive. This is similar to, but simpler

than, unpublished work by Neubauer and Thiemann [16]. Buffered communication is also used by Fähndrich *et al.* [6] but they have not published a formal semantics.

2. We work within the standard framework of a functional language with linear as well as unlimited types, treating session types as linear in order to guarantee that each channel endpoint is owned by a unique thread. For example,

$$\mathsf{receive} : ?T.S \to T \otimes S$$

so that the channel, with its new type, is returned with the received value.

3. We include two forms of subtyping: the standard subtyping relation for session types [10] and a novel form of subtyping between standard and linear function types [8].

The resulting system provides a clear and secure foundation for further developments such as polymorphism and object-orientation.

The outline of the rest of paper is as follows. Section 2 uses an example of a business process to present the language. Section 3 formally defines the syntax and the operational semantics. Section 4 defines the typing system and Section 5 gives the main results of the paper. Section 6 discusses related and future work.

2 Example: Business Protocol

We present a small example containing typical features of many web service business protocols [4, 22]. A mother and her young son are using an online book shop. The shop implements a simple protocol described by the session type

```
Shop = &⟨add: ?Book.Shop, checkout: ?Card.?Address.End⟩
```

The *branching* type constructor & indicates that the shop offers two options: add and checkout. After add, the shop receives (?) data of type Book, and then returns to the initial state. After checkout, the shop receives credit card details and an address for delivery, and that is the end of the interaction. Of course, a realistic shop would offer many more options.

To make the services of the shop available, the global environment should contain a name whose type is an access point for sessions of type Shop. We express this as shopAccess: [Shop]. A name such as shopAccess is analogous to a URL or an IP address, depending on the kind of service. The shop will contain an expression **accept** shopAccess and the shopper will contain an expression **request** shopAccess. At runtime these expressions interact to create a new private channel, which in the shop has type Shop and in the shopper has the dual type (Shopper = $\overline{\mathsf{Shop}}$; ! means send)

```
Shopper = ⊕⟨add:!Book.Shopper, checkout:!Card.!Address.End⟩
```

The shop is implemented as a function parameterised on its access point, using an auxiliary recursive function to handle the repetitive protocol. We do not show how the order is delivered, and assume the constructors emptyOrder and addBook.

```
shopLoop :: Shop → Order → Unit
shopLoop s order =
  case s of {
    add ⇒ λs. let (book,s) = receive s in
```

```
            shopLoop s (addBook book order)
    checkout ⇒ λs. let (card, s) = receive s in
                   let (address, s) = receive s in
                   close s }

shop :: [Shop] → Unit
shop shopAccess = shopLoop (accept shopAccess) emptyOrder
```

The **case** expression combines receiving an option and case-analysis of the option; the code includes a branch for each possibility.

The mother intends to choose a book for herself, then allow her son to choose a book. She does not want to give him free access to the channel which accesses the shop, so instead she gives him a function which allows him to choose exactly one book (of an appropriate kind) and then completes the transaction. This function plays the role of a gift voucher. Communication between mother and son is also described by a session type:

```
Son = ?(Book ⊸ Book).!Book.End
```

and the son has an access point of type [Son].

```
voucher :: Card → Address → Shop → Book ⊸ Book
voucher card address c book =
  let c = if (isChildrensBook book)
          then let c = select add c in
               send book c
          else c in
  let c = select checkout c in
  let c = send card c in
  let c = send address c in
  let _ = close c in
  book

mother :: Card → Address → [Shop] → [Son] → Book → Unit
mother card address shopAccess sonAccess book =
  let c = request shopAccess in
  let c = select add c in
  let c = send book c in
  let s = request sonAccess in
  let s = send (voucher card address c) s in
  let (sonBook, s) = receive s in
  close s

son :: [Son] → Book → Unit
son sonAccess book =
  let s = accept sonAccess in
  let (f, s) = receive s in
  let s = send (f book) s in
  close s
```

The complete system is a *configuration* of expressions in parallel, running as separate threads, typed in a suitable environment (which should also include the types of all of the functions, as well as mCard etc):

$$\begin{aligned}\mathsf{shopAccess}\colon [\mathsf{Shop}], \mathsf{sonAccess}\colon [\mathsf{Son}] \vdash \\ \langle \mathsf{shop}\ \mathsf{shopAccess}\rangle \parallel \langle \mathsf{son}\ \mathsf{sonAccess}\ \mathsf{sBook}\rangle \\ \parallel \langle \mathsf{mother}\ \mathsf{mCard}\ \mathsf{mAddress}\ \mathsf{shopAccess}\ \mathsf{sonAccess}\ \mathsf{mBook}\rangle\end{aligned}$$

The example illustrates the following general points about our language, its semantics and its type system; the details are presented in Sections 3 and 4.

- Channels, such as c in mother, are *linear* values; session types are linear types. The linear function type constructor $\multimap$ appears in the type of $\mathsf{voucher}$ because applying $\mathsf{voucher}$ to a channel of type Shop yields a function closure which contains a channel—hence this function closure must itself be treated as a linear value and given a linear type. Because of linearity, Son cannot duplicate the voucher and order more than one book.

- Operations on channels, such as **send** and **select**, return the channel after communicating on it. Our programming style is to repeatedly re-bind the channel name using **let**; each c is of course a fresh bound variable. The **receive** operation returns the value received and the channel, as a (linear) pair which is split by a **let** construct. In the static type system, the channel type returned by, for example, **send**, is not the same as the channel type given to it; this reflects the fact that part of the session types is consumed by a communication operation.

The *first novelty* of this work is a buffered, asynchronous, operational semantics. Previous work on functional session types [19] made the two threads involved in a session proceed in a lock-step fashion. The semantics presented here allows threads to proceed at their own pace, buffering the communications, thus allowing for efficient process communication, more in line with, say, socket implementations. For example, the mother may select add and send the book without blocking; the buffer (queue) for the end-channel c would contain both label add and the representation of book. This is a very important step, because synchronous communication is unrealistic; for example, previous work required a server to synchronise when sending a message to its client.

Observe that the type Shop allows an unbounded sequence of messages in the same direction, alternating between add labels and book details. The shop would therefore require a potentially unbounded buffer for incoming messages. However, Fähndrich *et al.* [6] have pointed out that if the session type does not allow unbounded sequences of messages in the same direction then it is possible to obtain a static upper bound on the size of the buffer. This is also true in our system. For example, the type S in Section 1 yields a bound of 2 because after sending $\mathsf{service}$ and an Int, the client must wait to receive an Int. A more realistic version of the shop example would require an acknowledgement when a book is added, and this would also lead to a bound on the buffer size.

The *second novelty* of the paper is the use of conventional functional types to describe session operations. As such, **send** is viewed as a function that accepts a value and a channel, and returns the same channel but in a state where the value has been sent. Similarly, **receive** accepts a channel and returns a pair whose first component is the value read from the channel and whose second component is the same channel in a state where a value has been read. The type system carefully distinguishes ordinary (unlimited) values from channel (linear) values, making sure that channel values are not duplicated. The type system supports programming with higher-order functions on channels in a very natural way, as illustrated by the function voucher in the example.

The *third novelty* of this paper is the subtyping relation introduced in Section 4. This combines two ideas. The first is that the standard function type is a subtype of the linear function type; this is useful for typechecking, as we will explain. The second is the inclusion of the standard subtyping relation for session types [10] in a functional language.

Variations of the example illustrate these and other features of our language.

Changing the function voucher. The mother decides that voucher should not order the book; she will complete the order herself. She defines

```
voucher book = book
```

which can have either of the types $\mathsf{Book} \rightarrow \mathsf{Book}$ and $\mathsf{Book} \multimap \mathsf{Book}$. We suggest that a type inference system should produce the type $\mathsf{Book} \rightarrow \mathsf{Book}$. Because we have $\mathsf{Book} \rightarrow \mathsf{Book} <: \mathsf{Book} \multimap \mathsf{Book}$ (Section 4), the expression **send** voucher b is still typable; there is no need to change the type Son.

Adding options to the session type Shop. The shop adds an option to remove a book from the order, changing the session type and its dual to

```
NewShop = &⟨add:  ?Book.NewShop,
            remove:  ?Book.NewShop,
            checkout:  ?Card.?Address.End⟩

NewShopper = ⊕⟨add:  !Book.NewShopper,
               remove:  !Book.NewShopper,
               checkout:  !Card.!Address.End⟩
```

We have $\mathsf{Shop} <: \mathsf{NewShop}$ and $\mathsf{NewShopper} <: \mathsf{Shopper}$. If the type of shopAccess in the global environment changes to [NewShop] then expression **request** shopAccess in mother returns a channel of type NewShopper. The subtyping relationship means that this channel can still be given to voucher as a parameter.

Using a third-party shipper. Like previous systems of session types, our type system allows channels to be sent on channels. For example, suppose that the shop uses a separate service, shipper, to arrange delivery of the order. When shop has received the customer's credit card details, it just passes the channel to shipper. When the customer sends her address, it goes directly to shipper. The session type used for communication between shop and shipper is as follows; note the occurrence of the session type ?Address.**End** as the type of the message.

```
Shipper = ?(?Address.End).End
```

The type Shop is not changed, and therefore mother is unaware of any change.

3 Syntax and Operational Semantics

Most of the syntax of our language was described in the previous section. We rely on a countable set of *term variables* x, and on a disjoint countable set of (runtime) *end-channels*

$$
\begin{aligned}
v &::= () \mid c \mid \lambda x.e \mid \mathsf{fix} \mid (v,v) \mid \\
&\qquad \mathsf{request} \mid \mathsf{accept} \mid \mathsf{send} \mid \mathsf{send}\ v \mid \mathsf{receive} \mid \mathsf{close} \\
e &::= v \mid x \mid ee \mid (e,e) \mid \mathsf{let}\ (x,x) = e\ \mathsf{in}\ e \mid \mathsf{fork}\ e\ e \mid \\
&\qquad \mathsf{select}\ l\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{l_i \colon e_i\}_{i \in I} \\
C &::= \langle e \rangle \mid C \parallel C \\
E &::= [\,] \mid Ee \mid vE \mid (E,e) \mid (v,E) \mid \mathsf{let}\ (x,y) = E\ \mathsf{in}\ e \mid \\
&\qquad \mathsf{select}\ l\ E \mid \mathsf{case}\ E\ \mathsf{of}\ \{l_i \colon e_i\}_{i \in I} \\
b &::= v \mid l \mid \mathsf{closed} \\
B &::= \emptyset \mid B, c \mapsto (c, \vec{b})
\end{aligned}
$$

Figure 1: Syntax.

$$(\lambda x.e)v \longrightarrow_{\mathsf{v}} e\{v/x\} \qquad \text{(R-App)}$$

$$\mathsf{let}\ (x,y) = (v,v')\ \mathsf{in}\ e \longrightarrow_{\mathsf{v}} e\{v/x\}\{v'/y\} \qquad \text{(R-Split)}$$

$$\mathsf{fix}\ (\lambda x.e) \longrightarrow_{\mathsf{v}} e\{(\mathsf{fix}\ (\lambda x.e))/x\} \qquad \text{(R-Fix)}$$

Figure 2: Basic reduction.

c, and define *values* v, *expressions* e and *configurations* C as in Figure 1. The operational semantics of the language is defined via the reduction relation in Figures 2, 3, and 4.

Figure 2 presents intra-thread reduction. Rule R-App describes the application of an argument to a function, by replacing the parameter x by the argument v by in the body of the function e (notation $e\{v/x\}$ denotes this operation). Rule R-Split opens a pair of values (v, v') allowing expression e to use each component via variables x and y. Finally, rule R-Fix expands a recursive function definition.

To simplify the presentation of inter-thread reduction, we rely on two notions: evaluation contexts (Figure 1) [23] and structural equivalence on configurations. An evaluation context is an expression with a hole, denoted [ ], where, intuitively, computation happens "next". Syntax $E[e]$ denotes the result of filling the hole of context E with expression e. Rules R-Thread and R-Fork in Figure 3 describe intra-thread reduction steps, by taking advantage of this notion. Structural equivalence, the smallest relation satisfying the two rules

$$C_1 \parallel C_2 \ \equiv C_2 \parallel C_1 \qquad \text{(E-Comm)}$$

$$C_1 \parallel (C_2 \parallel C_3) \ \equiv (C_1 \parallel C_2) \parallel C_3 \qquad \text{(E-Assoc)}$$

allows changing the syntactic order of the components in a configuration. Rules R-Par and R-Struct in Figure 3 isolate two threads that will engage in inter-thread communication, via the rules in Figure 4.

Except for the rules aforementioned, reduction relies on a global data structure, called a *buffer table* B, described in Figure 1. Such a structure maps a channel c into a pair, consisting of another channel (called the *peer end-channel* of c), and of a sequence of *channel* values (called the *channel queue*). Items in the channel queue can be any value v in the syntax of the language in Figure 1 (written and read by send and receive expressions), any label l

$$\frac{e \longrightarrow_{\mathsf{v}} e'}{B, \langle E[e] \rangle \longrightarrow B, \langle E[e'] \rangle} \tag{R-Thread}$$

$$B, \langle E[\mathsf{fork}\ e\ e'] \rangle \longrightarrow B, \langle e \rangle \parallel \langle E[e'] \rangle \tag{R-Fork}$$

$$\frac{B, C \longrightarrow B', C'}{B, C \parallel C'' \longrightarrow B', C' \parallel C''} \tag{R-Par}$$

$$\frac{C \equiv C' \qquad B, C' \longrightarrow B', C'' \qquad C'' \equiv C'''}{B, C \longrightarrow B, C'''} \tag{R-Struct}$$

$$\begin{aligned} &B, \langle E[\mathsf{request}\ x] \rangle \parallel \langle E'[\mathsf{accept}\ x] \rangle \longrightarrow \\ &\qquad B + \{c \mapsto (d, \varepsilon), d \mapsto (c, \varepsilon)\}, \langle E[c] \rangle \parallel \langle E'[d] \rangle \end{aligned} \tag{R-Init}$$

Figure 3: Reduction of configurations I/II.

(written and read by select and case expressions), and the special item closed (used by close expressions).

Rule R-Init synchronizes two threads trying to start a new connection on a common name x. Two new end-channels are created, c and d, one for each thread. Also, two new entries are added to the buffer table, each mentioning its peer end-channel. Symbol ε denotes an empty queue.

Rules R-Send and R-Select write on the peer end-channel. If d is the peer end-channel of c, then, rule R-Send enqueues value v in the channel queue of d. On the other hand, rule R-Select enqueues label l in the channel queue of d. In either case, the result of this operation is channel c, which can then be used for further interaction.

Rules R-Receive and R-Branch read from the channel queue. Rule R-Receive dequeues value v at the head of the queue associated to end-channel c; rule R-Branch does the same for label l_j at the head of the queue. The result of the evaluation of expression receive c is a pair composed of v and channel c itself. On the other hand, the result of the evaluation of expression case c of $\{l_i \colon e_i\}_{i \in I}$ is the application of function e_j, body of branch labelled by l_j, to channel c. In either case, channel c can then be used for further interaction, as in the case of the two read rules above.

There are two rules responsible for closing an end-channel, each inspecting the channel value at the head of the end-channel. If the special value closed is found, meaning the peer thread has closed the channel, rule R-CloseClose removes the pair of end-channels from the buffer table. On the other hand, if the mark is not found at the head of the queue, the thread enqueues closed in the peer end-channel.

4 Typing

We now introduce a static type system for our language. The syntax of types is defined in Figure 5.

Session types S will be associated with channels. End is the type of a channel which cannot be used for further communication; the only possible operation is close. $?T.S$ is the type of a channel from which a message of type T can be received; subsequently the channel is described by type S. Dually, $!T.S$ is the type of a channel on which a message of type T

$$\frac{B(c) = (d, \_) \qquad B(d) = (c, \vec{b})}{B, \langle E[\mathsf{send}\ v\ c]\rangle \longrightarrow B + \{d \mapsto (c, \vec{b}v)\}, \langle E[c]\rangle} \quad (\text{R-Send})$$

$$\frac{B(c) = (d, \_) \qquad B(d) = (c, \vec{b})}{B, \langle E[\mathsf{select}\ l\ c]\rangle \longrightarrow B + \{d \mapsto (c, \vec{b}l)\}, \langle E[c]\rangle} \quad (\text{R-Select})$$

$$\frac{B(c) = (d, v\vec{b})}{B, \langle E[\mathsf{receive}\ c]\rangle \longrightarrow B + \{c \mapsto (d, \vec{b})\}, \langle E[(v, c)]\rangle} \quad (\text{R-Receive})$$

$$\frac{B(c) = (d, l_j\vec{b}) \qquad j \in I}{B, \langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i \colon e_i\}_{i\in I}]\rangle \longrightarrow B + \{c \mapsto (d, \vec{b})\}, \langle E[e_j c]\rangle} \quad (\text{R-Branch})$$

$$\frac{B(c) = (d, \mathsf{closed}\ \_)}{B, \langle E[\mathsf{close}\ c]\rangle \longrightarrow B \setminus \{c, d\}, \langle E[()]\rangle} \quad (\text{R-CloseClose})$$

$$\frac{B(c) = (d, \vec{b}) \qquad \vec{b} \neq \mathsf{closed}\ \_ \qquad B(d) = (c, \vec{b}')}{B, \langle E[\mathsf{close}\ c]\rangle \longrightarrow B + \{d \mapsto (c, \vec{b}'\ \mathsf{closed})\}, \langle E[()]\rangle} \quad (\text{R-Close})$$

Figure 4: Reduction of configurations II/II.

$$\begin{aligned}
S &::= \mathsf{End} \mid ?T.S \mid !T.S \mid \&\langle l_i \colon S_i\rangle_{i\in I} \mid \oplus\langle l_i \colon S_i\rangle_{i\in I} \\
T &::= S \mid \mathsf{Unit} \mid T \otimes T \mid T \to T \mid T \multimap T \mid [S] \\
\Gamma &::= \emptyset \mid \Gamma, x \colon T \mid \Gamma, c \colon S
\end{aligned}$$

Figure 5: Syntax of types.

can be sent; subsequently the type of the channel is S. $\&\langle l_i \colon S_i\rangle_{i\in I}$ is the type of a channel from which a message can be received, which will be one of the labels l_i. The subsequent behaviour of the channel is described by the corresponding type S_i. Dually, $\oplus\langle l_i \colon S_i\rangle_{i\in I}$ is the type of a channel on which one of the labels l_i can be sent, with subsequent behaviour described by S_i. Each session type S has a dual type $\overline{S}$, defined in Figure 6. The present paper does not include recursive session types. There is no technical difficulty in including them, along the lines of [10, 24], but the details are rather long because of the need to define duality and subtyping coinductively.

General types are denoted by T, including session types S as one case. Because channels must be controlled linearly, so that each end-channel is owned by a unique thread within the system, the type system includes constructors for linear pairs $T \otimes U$ and linear functions $T \multimap U$ as well as standard functions $T \to U$. Each type is either *linear* or *unlimited*, as defined in Figure 7. The single-valued type Unit is included because it occurs naturally in the typing of some of the operators of the language; data types such as Int and Bool can easily be added. The type $[S]$ describes a name that can be used to establish a session. If a typed name $n \colon [S]$ occurs in the global environment then a matching request n and accept n create a channel. On the client side, request n yields a channel endpoint of type $\overline{S}$, while on the server side, accept n yields the peer endpoint of type S.

The type system includes a subtyping relation, defined in Figure 8. Part of this is the standard definition of subtyping for session types [10], specialized to non-recursive types.

$$\overline{\mathsf{End}} = \mathsf{End} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\oplus\langle l_i\colon S_i\rangle_{i\in I}} = \&\langle l_i\colon \overline{S_i}\rangle_{i\in I}$$
$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{\&\langle l_i\colon S_i\rangle_{i\in I}} = \oplus\langle l_i\colon \overline{S_i}\rangle_{i\in I}$$

Figure 6: Duality on session types.

$$\mathsf{lin}(S) \qquad \mathsf{lin}(T \otimes T) \qquad \mathsf{lin}(T \multimap T)$$
$$\mathsf{un}(\mathsf{Unit}) \qquad \mathsf{un}(T \to T) \qquad \mathsf{un}([S])$$

Figure 7: Classification of types as linear (lin) or unlimited (un).

The rest consists of the usual subtyping rules for function types, and a novel subtyping relationship between standard and linear function types [8], expressed in rule S-FunFun.

Type environments are defined by the grammar in Figure 5. The order of environment entries is unimportant: regard an environment as a partial function from variables and channels to types. Write $\mathrm{dom}(\Gamma)$ for the set of variables and channels in Γ; write $\mathrm{cdom}(\Gamma)$ for the set of channels in Γ; and say that $\mathsf{un}(\Gamma)$ is true of an environment where all types are unlimited.

In the usual way for a type system with linear types [21], we define a partial operation of addition on environments. Let α be either a variable x or channel c. If $\alpha \notin \mathrm{dom}(\Gamma)$ then $\Gamma + \alpha\colon T = \Gamma, \alpha\colon T$. If $\alpha\colon T \in \Gamma$ and $\mathsf{un}(T)$ then $\Gamma + \alpha\colon T = \Gamma$. In all other cases, $\Gamma + \alpha\colon T$ is undefined. Addition is extended inductively to a partial binary operation on environments. Typing rules in which environments are added contain an implicit condition that the addition must be defined.

Typing of expressions is defined in Figures 9 and 10. The typings in Figure 9 should be understood as schemas, which can be instantiated for any appropriate types. The schemas for send and receive capture the essence of the way in which we use linear type constructors to control the use of channels. We treat send as a curried function which is given a value and a channel and returns the same channel with the type that remains after sending the specified value. There are two versions of this schema, because the partial application send v contains v in its closure and therefore we must use a linear function type if v has a linear type. The receive function is given a channel of appropriate type and returns, with the received value, the same channel, again with its remaining type. The return type of receive is a linear pair because S, being a session type, is linear.

The type of close is similar to that of send, but simpler because there is no value to specify and no channel to return. The functions request and accept return new end-channels of dual types S and $\overline{S}$ corresponding to the type $[S]$ of the given name.

Most of the rules in Figure 10 are standard. T-Fork describes spawning a new thread, whose type is required to be unlimited in order to ensure that the thread completely consumes any channels that it uses, eventually closing them. T-Select describes sending a label to select one of the possible behaviours; like send, the channel is returned with the appropriate type. T-Case requires the case-expression e to be of a branch type; the expressions e_i in each branch must be functions accepting the appropriate channel (of type T_i). T-App includes application of standard functions, by subsumption.

Figure 12 defines typing of configurations and of buffered configurations, beginning with

$$T <: T \qquad \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3} \qquad \text{(S-Refl,S-Trans)}$$

$$\frac{U_1 <: T_1 \quad T_2 <: U_2}{T_1 \multimap T_2 <: U_1 \multimap U_2} \qquad \frac{U_1 <: T_1 \quad T_2 <: U_2}{T_1 \to T_2 <: U_1 \to U_2} \qquad T_1 \to T_2 <: T_1 \multimap T_2$$

$$\text{(S-FunL,S-Fun,S-FunFun)}$$

$$\frac{T_1 <: T_2 \quad S_1 <: S_2}{?T_1.S_1 <: ?T_2.S_2} \qquad \frac{T_2 <: T_1 \quad S_1 <: S_2}{!T_1.S_1 <: !T_2.S_2} \qquad \text{(S-In,S-Out)}$$

$$\frac{I \subseteq J \quad \forall_{i\in I}(S_i <: S_i')}{\&\langle l_i \colon S_i\rangle_{i\in I} <: \&\langle l_i \colon S_i'\rangle_{i\in J}} \qquad \frac{J \subseteq I \quad \forall_{i\in J}(S_i <: S_i')}{\oplus\langle l_i \colon S_i\rangle_{i\in I} <: \oplus\langle l_i \colon S_i'\rangle_{i\in J}} \qquad \text{(S-Branch,S-Choice)}$$

Figure 8: Subtyping.

$$\begin{array}{rll@{\qquad}rl}
() : & \mathsf{Unit} & & \mathsf{receive} : & ?T.S \to T \otimes S \\
\mathsf{fix} : & (T \to T) \to T & & \mathsf{close} : & \mathsf{End} \to \mathsf{Unit} \\
\mathsf{send} : & T \to !T.S \multimap S & & \mathsf{request} : & [S] \to \overline{S} \\
\mathsf{send} : & T \to !T.S \to S & \text{if } \mathsf{un}(T) & \mathsf{accept} : & [S] \to S
\end{array}$$

Figure 9: Typing schemas for constants.

a single thread (containing an expression) and allowing configurations to be combined in parallel. Again the type of a thread is required to be unlimited. For buffered communications we have two rules: for a configuration with an empty buffer table and a for a configuration holding at least the two ends (c, c') of a channel. This second rule, T-AddBuffers, make use of definitions from Figure 11.

The first hypothesis of T-AddBuffers say that the buffered configuration without the selected channel should be typable. The second hypothesis state that the values $\vec{b}$ in the buffer for the channel end c should match the type for c, as seen by configuration C (hence the $\Gamma(c)$). This is expressed via the matches relation, defined in Figure 11. If there is data in the buffer for c, then the session type of c must allow the data to be received; messages must have subtypes of the expected types, and labels must correspond to options existing in a branch type. The closed token indicates that the peer end-channel has been closed, and therefore c must also be ready to be closed. Intuitively, if $\Gamma \vdash \vec{b}$ matches S then $\vec{b}$ traces a path through the tree structure of S. This path does not necessarily go all the way to a leaf, and we define $S/\vec{b}$ to be the session type at which $\vec{b}$ stops, as illustrated in Figure 11. This type is used in the final hypothesis of T-AddBuffers, which states that peer end-channels have dual session types after ignoring the initial parts which match the data in their buffers. If both buffers are empty, so that input and output synchronized, then the condition would simply be that if c and c' are peers then $\Gamma_2(c) = \overline{\Gamma_2(c')}$.

Figure 13 shows an abbreviated typing derivation for the function shopLoop from Section 2. The recursive definition of shopLoop is translated into fix λf.λs.λo.**case** s of $\{\ldots\}$ in the usual way. We show part of the typing derivation for the expression within fix.

Typechecking. We typecheck a configuration, that is, a program with an empty buffer table. The typing rules for buffered configurations (T-Empty, T-AddBuffers, Figure 12)

$$\frac{\mathsf{un}(\Gamma)\quad k\colon T}{\Gamma \vdash k\colon T} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, x\colon T \vdash x\colon T} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, c\colon S \vdash c\colon S} \qquad \frac{\Gamma \vdash e\colon T \quad T <: U}{\Gamma \vdash e\colon U}$$

(T-Const,T-Var,T-Chan,T-Sub)

$$\frac{\Gamma_1 \vdash e_1\colon T \quad \Gamma_2 \vdash e_2\colon U}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2)\colon T \otimes U} \qquad \frac{\Gamma_1 \vdash e_1\colon T \otimes U \quad \Gamma_2, x\colon T, y\colon U \vdash e_2\colon V}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x,y) = e_1\ \mathsf{in}\ e_2\colon V} \qquad \text{(T-Pair,T-Split)}$$

$$\frac{\Gamma, x\colon T \vdash e\colon U \quad \mathsf{un}(\Gamma)}{\Gamma \vdash \lambda x.e\colon T \to U} \qquad \frac{\Gamma, x\colon T \vdash e\colon U}{\Gamma \vdash \lambda x.e\colon T \multimap U} \qquad \text{(T-Abs,T-AbsL)}$$

$$\frac{\Gamma_1 \vdash e_1\colon T \multimap U \quad \Gamma_2 \vdash e_2\colon T}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2\colon U} \qquad \frac{\Gamma_1 \vdash e_1\colon T \quad \Gamma_2 \vdash e_2\colon U \quad \mathsf{un}(T)}{\Gamma_1 + \Gamma_2 \vdash \mathsf{fork}\ e_1\ e_2\colon U} \qquad \text{(T-App,T-Fork)}$$

$$\frac{\Gamma \vdash e\colon \oplus\langle l_i\colon T_i\rangle_{i\in I} \quad j \in I}{\Gamma \vdash \mathsf{select}\ l_j\ e\colon T_j} \qquad \frac{\Gamma_1 \vdash e\colon \&\langle l_i\colon T_i\rangle_{i\in I} \quad \forall_{i\in I}(\Gamma_2 \vdash e_i\colon T_i \multimap T)}{\Gamma_1 + \Gamma_2 \vdash \mathsf{case}\ e\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}\colon T}$$

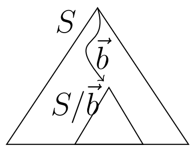
(T-Select,T-Case)

Figure 10: Typing rules for expressions.

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \varepsilon\ \mathsf{matches}\ S} \qquad \frac{\Gamma_1 \vdash v\colon T \qquad T <: U \qquad \Gamma_2 \vdash \vec{b}\ \mathsf{matches}\ S}{\Gamma_1 + \Gamma_2 \vdash v\vec{b}\ \mathsf{matches}\ ?U.S}$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{closed}\ \mathsf{matches}\ \mathsf{End}} \qquad \frac{\Gamma \vdash \vec{b}\ \mathsf{matches}\ S}{\Gamma \vdash l\vec{b}\ \mathsf{matches}\ \&\langle \ldots, l\colon S, \ldots\rangle}$$

$$\begin{aligned} S/\varepsilon &= S \\ \mathsf{End}/\mathsf{closed} &= \mathsf{End} \\ ?\_.S/\_\vec{b} &= S/\vec{b} \\ \&\langle \ldots, l\colon S, \ldots\rangle/l\vec{b} &= S/\vec{b} \end{aligned}$$



If $\Gamma \vdash \vec{b}$ matches S is defined (by the rules at the top) then we define $S/\vec{b}$ by the rules at the bottom. The diagram illustrates $S/\vec{b}$.

Figure 11: The matches relation.

are only needed to prove type preservation.

It is straightforward to convert the typing rules of Figures 9 and 10, and rules T-Thread and T-Conf from Figure 12, into a typechecking algorithm, using standard techniques for type systems with linear and unlimited types [21] (type annotation may be needed for the bound variable in a lambda abstraction). To make a syntax-directed algorithm, rule T-Abs should be used in preference to T-AbsL, so that functions are given standard types when possible. If it turns out that the standard function type was the wrong choice, the subtyping relation (rule S-FunFun) allows it to be converted to a linear function type when necessary, as illustrated in Section 2.

5 Type Safety

We prove that the operational semantics preserves typability of buffered configurations, and then prove an explicit runtime safety theorem.

$$\frac{\Gamma \vdash e\colon T \quad \mathsf{un}(T)}{\Gamma \vdash \langle e \rangle} \qquad \frac{\Gamma_1 \vdash C_1 \quad \Gamma_2 \vdash C_2}{\Gamma_1 + \Gamma_2 \vdash C_1 \parallel C_2} \qquad \frac{\Gamma \vdash C}{\Gamma \vdash \emptyset, C} \quad \text{(T-Thread,T-Conf,T-Empty)}$$

$$\frac{\Gamma \vdash B, C \quad \Gamma_1 \vdash \vec{b}\ \mathsf{matches}\ \Gamma(c) \quad \Gamma_2 \vdash \vec{b'}\ \mathsf{matches}\ \Gamma(c') \quad \Gamma(c)/\vec{b} = \overline{\Gamma(c')/\vec{b'}}}{\Gamma + \Gamma_1 + \Gamma_2 \vdash B \cup \{c \mapsto (c', \vec{b}), c' \mapsto (c, \vec{b'})\}, C} \quad \text{(T-AddBuffers)}$$

Figure 12: Typing rules for configurations and buffered configurations.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\textit{omitted}}{\mathsf{s}\colon ?\mathsf{Book}.\mathsf{Shop} \vdash \mathbf{receive}\ \mathsf{s}\colon \mathsf{Book} \otimes \mathsf{Shop}} \quad \dfrac{\textit{omitted}}{\mathsf{f}\colon ..,\mathsf{o}\colon ..,\mathsf{book}\colon \mathsf{Book}, \mathsf{s}\colon \mathsf{Shop} \vdash \mathsf{f\ s\ (addBook\ book\ o)}\colon \mathsf{Unit}}}{\mathsf{f}\colon ...,\mathsf{o}\colon ...,\mathsf{s}\colon ?\mathsf{Book}.\mathsf{Shop} \vdash \mathbf{let}\ (\mathsf{book}, \mathsf{s}) = ...\colon \mathsf{Unit}}}{\mathsf{f}\colon ...,\mathsf{o}\colon ... \vdash \lambda s.\mathbf{let}\ (\mathsf{book},\mathsf{s}) = ...\colon ?\mathsf{Book}.\mathsf{Shop} \multimap \mathsf{Unit}} \quad \vdots \quad \dfrac{\textit{omitted}}{\mathsf{f}\colon ...,\mathsf{o}\colon ... \vdash \lambda s.\mathbf{let}\ (\mathsf{card},\mathsf{s}) = ...\colon !\mathsf{Card}.!\mathsf{Address}.\mathbf{End} \multimap \mathsf{Unit}}}{\mathsf{f}\colon \mathsf{Shop} \to \mathsf{Order} \to \mathsf{Unit}, \mathsf{s}\colon \mathsf{Shop}, \mathsf{o}\colon \mathsf{Order} \vdash \mathbf{case}\ \mathsf{s\ of}\ \{\lambda \mathsf{s}.\mathbf{let}\ (\mathsf{book},\mathsf{s}) = ..., \lambda \mathsf{s}.\mathbf{let}\ (\mathsf{card},\mathsf{s}) = ...\}\colon \mathsf{Unit}}}{\vdash \lambda \mathsf{f}.\lambda \mathsf{s}.\lambda \mathsf{o}.\mathbf{case}\ \mathsf{s\ of}\ \{...\}\colon (\mathsf{Shop} \to \mathsf{Order} \to \mathsf{Unit}) \to \mathsf{Shop} \to \mathsf{Order} \to \mathsf{Unit}}$$

Figure 13: Typing derivation for shopLoop (abbreviated).

Lemma 1 *If $\Gamma \vdash e\colon T$ and $e \longrightarrow_{\mathsf{v}} e'$, then $\Gamma \vdash e'\colon T$.*

Proof: In the usual way, making use of a substitution lemma which takes subtyping into account. □

Typability of a buffered configuration (B, C) alone does not ensure type preservation or type safety for the type system does not check that the free channels in C have buffers in B, hence the condition $\mathrm{dom}(B) = \mathrm{cdom}(\Gamma)$ in the two results below.

Theorem 2 (Type Preservation) *If $\Gamma \vdash B, C$ and $\mathrm{dom}(B) = \mathrm{cdom}(\Gamma)$ and $B, C \longrightarrow B', C'$, then there exists Γ' such that $\Gamma' \vdash B', C'$ and $\mathrm{dom}(B') = \mathrm{cdom}(\Gamma')$.*

Proof: *(Sketch)* By induction on the derivation of $B, C \longrightarrow B', C'$ with a case-analysis on the last rule.

(R-Thread,R-Fork) Straightforward, making use of the usual lemmas on typability of subterms and replacement of subterms within evaluation contexts [23]. These lemmas are also used in subsequent cases.

(R-Par,R-Struct) Straightforward, making use of a lemma that structural equivalence preserves typability.

(R-Init) request x and accept x return channel endpoints c and d of dual types, and their buffers are empty, so the condition $\Gamma'(c)/\varepsilon = \overline{\Gamma'(d)/\varepsilon}$ holds.

(R-Send) $\Gamma(c) = S_c$ and $\Gamma(d) = S_d$, and $S_c = !T.S'_c$ where T is the type of v. So the matching condition means that the buffer of c is empty; also $\Gamma \vdash \vec{b}$ matches S_d where $\vec{b}$ is the buffer of d. Before the reduction, $S_c = \overline{S_d/\vec{b}}$, so $S_d/\vec{b} = ?T.S'_d$. After the reduction, $\Gamma' \vdash \vec{b}v$ matches S_d, and $S_d/\vec{b}v = S'_d = \overline{S'_c}$. The remainder of the proof consists of constructing the necessary typing derivation, in which the part of the environment needed to type v moves from the typing of C to the typing of B. The case of R-Select is similar.

(R-RECEIVE) $\Gamma(c) = S_c = ?T.S'_c$. The matching condition means that the type of v is a subtype of T and can be safely received. Before the reduction we have $S_c/v\vec{b}$ which is the same as $S'_c/\vec{b}$ after the reduction, so the duality condition is satisfied. The case of R-BRANCH is similar.

(R-CLOSE,R-CLOSECLOSE) Similar reasoning to R-SEND. □

Type safety states important properties of the typed configurations: (1) that channel-ends occur linearly, (2) that if one channel-end occurs in a configuration, then both ends have entries in the buffer table, (3,4) that buffers contain the expected data for the processes willing to read, and (5,6) that let and fix are given the expected values.

Theorem 3 (Runtime Safety) *If* $\Gamma \vdash B, C$ *and* $\mathrm{dom}(B) = \mathrm{cdom}(\Gamma)$*, then:*

1. *Every free channel in c occurs in exactly one thread within* C*.*
2. *If* $C \equiv \langle E[e]\rangle \parallel C'$ *and e is an operation on channel c, then c and its peer are in* B*.*
3. *If* $C \equiv \langle E[\mathsf{receive}\ v]\rangle \parallel C'$ *and* $B(c)$ *contains data, then the first data value is a value* v*.*
4. *If* $C \equiv \langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}]\rangle \parallel C'$ *and* $B(c)$ *contains data, then the first data value is* l_j *for some* $j \in I$*.*
5. *If* $C \equiv \langle E[\mathsf{let}\ (x, y) = v\ \mathsf{in}\ e]\rangle \parallel C'$*, then* $v = (v_1, v_2)$*.*
6. *If* $C \equiv \langle E[\mathsf{fix}\ v]\rangle \parallel C'$*, then* $v = \lambda x.e$*.*

Proof: By analyzing the derivation of $\Gamma \vdash B, C$. □

6 Related and Future Work

Singularity is an operating system where processes communicate solely via message passing [6]. The system is written in Sing#, an extension of language C#, where session types provide for invariants that enable efficient process communication with low overhead. A notable feature of the system is the asynchronous operational semantics, similar to what we formally define in this paper. Neubauer and Thiemann [16] present an asynchronous semantics for a functional language with session types, where threads communicate via a pair of streams (end-channels), similarly to Sing# and the present work. They formally define an operational semantics and a type system, and prove a type soundness result. Our formulation is simpler.

Dezani-Ciancaglini et al. [4, 5] have ported session types to the world of objects. They use a synchronous semantics and do not deal with subtyping. The language in [4] enjoys an interesting progress property, whereby well-typed programs do not starve at communication points, once a session is established. The price to pay is the impossibility of interleaving communications on different channels, by the same thread.

Cyclone [11, 12], *Vault* [3], and *adoption and focus* [7] are systems based on the C programming language that allow protocols to be statically enforced by a compiler. They share with our work the goal of statically enforcing protocols, but vary greatly in the techniques used.

Cyclone [12] is a C-like type-safe polymorphic imperative language. It features region-based memory management, and more recently threads and locks [11], via a sophisticated type system. The multithreaded version requires "a lock name for every pointer and lock type, and an effect for every function". Our locks are channels; but more than mutual exclusion, channels also allow a precise description of the protocol "between" acquiring and releasing the lock. In Cyclone a thread acquires a lock for a resource, uses the resource in whichever way it needs, and then releases the lock. Using our language a thread acquires the lock via a request operation, and then follows a specified protocol for the resource, before closing the channel obtained with request.

In the *Vault* system [3] annotations are added to C programs, in order to describe protocols that a compiler can statically enforce. Similarly to our approach, individual runtime objects are tracked by associating keys (channels, in our terminology) with resources, and function types describe the effect of the function on the keys. Although incorporating a form of selection ($\oplus$), the type system describes protocols in less detail than we can achieve with session types. "Adoption and Focus" [7], by the same authors, is a type system able to track changes in the state of objects; the system handles aliasing, and includes a form of polymorphism in functions. In contrast, our system checks the types of individual messages, as well as tracking the state of the channel. Our system is more specialized, but the specialization allows more type checking in the situation that we handle.

The present authors studied type checking for session types in a functional language [19]. In contrast to the present work, there we used a synchronous semantics and non-standard function types and typing judgments, where the initial and final types of the channels involved in each operation were made explicit.

Yoshida and Vasconcelos [24] study two similar π-calculus based systems: that of Honda et al. [14], and a variant appearing in several subsequent works on session types. They show that to model "true" channel passing, where one thread may acquire both ends of a communication channel a formal system is needed where the two ends of the channel are treated separately. Following an idea by Gay and Hole [10], the authors syntactically distinguish the two ends of a channel by tagging them with a distinct, plus or minus, polarity. The same effect is achieved in this work by equipping the entry of a channel in a buffer table with the peer end-channel (in addition to the buffer itself).

The main areas for future work are to formalize a theory of object-oriented session types in greater generality than exists at present, and to include polymorphism, either in a simple ML-style or along the lines of [9]. The relationship with various forms of static analyses, including type and effect systems [1] should be investigated.

Acknowledgements. This work was partially supported by FEDER, the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004), Fundação para a Ciência e a Tecnologia (via CITI, and project Space–Time–Types, POSC/EIA/55582/2004).

References

[1] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* IC Press, 1999.

[2] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communication. *Journal of Functional Programming*, 15(2):219–247, 2005.

[3] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Conference on Programming Language Design and Implementation*, volume 36(5) of *SIGPLAN Notices*, pages 59–69. ACM Press, 2001.

[4] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006.

[5] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *Proceedings of the Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys2006*, ACM SIGOPS, 2006.

[7] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Conference on Programming Language Design and Implementation*, volume 37(5) of *SIGPLAN Notices*, pages 13–24, 2002.

[8] S. J. Gay. Subtyping between standard and linear function types. `www.dcs.gla.ac.uk/~simon/publications/StandardLinearSubtyping.pdf`. Manuscript, 2006.

[9] S. J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 2007. To appear.

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

[11] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation*, volume 38(3) of *SIGPLAN Notices*, pages 13–25. ACM Press, 2003.

[12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Conference on Programming Language Design and Implementation*, volume 37(5) of *SIGPLAN Notices*, pages 282–293. ACM Press, 2002.

[13] K. Honda. Types for dyadic interaction. In *CONCUR'93: Proceedings of the International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[14] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98: Proceedings of the European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[15] M. Neubauer and P. Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[16] M. Neubauer and P. Thiemann. Session types for asynchronous communication. Unpublished, 2004.

[17] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE '94: Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*. Springer, 1994.

[18] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticae*, 73(4):583–598, 2006.

[19] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

[20] V. T. Vasconcelos, A. Ravara, and S. J. Gay. Session types for functional multithreading. In *CONCUR'04: Proceedings of the International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2004.

[21] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–43. MIT Press, 2005.

[22] Web Services Choreography Working Group. Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.

[23] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[24] N. Yoshida and V. 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*, ENTCS, 2007. To appear.