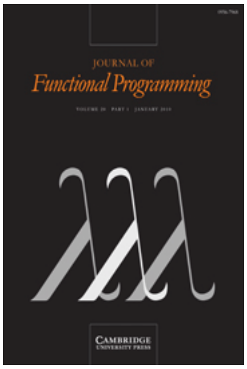


Linear type theory for asynchronous session types

SIMON J. GAY and VASCO T. VASCONCELOS

Journal of Functional Programming / Volume 20 / Issue 01 / January 2010, pp 19 - 50
DOI: 10.1017/S0956796809990268, Published online: 08 December 2009

JFP **20** (1): 19–50, 2010. © Cambridge University Press 2009
doi:10.1017/S0956796809990268 First published online 8 December 2009

Linear type theory for asynchronous session types

SIMON J. GAY
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
(*e-mail:* `simon@dcs.gla.ac.uk`)

VASCO T. VASCONCELOS
Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal
(*e-mail:* `vv@di.fc.ul.pt`)

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 typechecking. 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 four main contributions. First is an operational semantics with buffered channels, instead of the synchronous communication of previous work. Second, we prove that the session type of a channel gives an upper bound on the necessary size of the buffer. Third, session types are manipulated by means of the standard structures of a linear type theory, rather than by means of new forms of typing judgement. Fourth, 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, which allows the typechecker to handle linear types conveniently. 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.

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 reuse of existing components and strict requirements for availability and correctness. In this setting, formal development methods and 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.

The current paper concerns one approach to static analysis of the communication behaviour of agents in a distributed system: session types (Honda 1993; Takeuchi *et al.* 1994; Honda *et al.* 1998; Yoshida & Vasconcelos 2007). 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 $\mathsf{S} = \&\langle \mathsf{service}\colon ?\mathbf{Int}.!\,\mathbf{Int}.\mathsf{S},\ \mathsf{quit}\colon\mathsf{End}\rangle$ 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 and 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 (W3C 2005) and operating system services (Fähndrich *et al.* 2006). By incorporating correspondence assertions, the behavioural guarantees offered by session types have been strengthened and applied to security analysis (Bonelli *et al.* 2005). A theory of subtyping for session types has been developed (Gay & Hole 2005) and adapted for specifying distributed software components (Vallecillo *et al.* 2006). Session types are 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. 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 (Gay *et al.* 2003; Vasconcelos *et al.* 2004, 2006) was the first proposal for a functional language with session types. Neubauer and Thiemann (2004a) 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.* (2005, 2006), Coppo *et al.* (2007) and Capecchi *et al.* (2009) and is included in the work of Fähndrich *et al.* (2006).

In the present paper we define a multithreaded functional language with session types, unifying and simplifying several strands of previous work and extending the preliminary version (Gay & Vasconcelos 2007) and furthermore clarifying the relationship between session types and standard functional type theory. The contributions of the paper are as follows:

1. Building on our previous work (Gay & Vasconcelos 2007), we formalise an operational semantics in which communication is buffered, instead of assuming synchronisation between send and receive, as in previous work (Vasconcelos *et al.* 2006, 2004). This is more realistic and means that send and select never block. The semantics is similar to, but simpler than, the unpublished work by Neubauer and Thiemann (2004c). Fähndrich *et al.* (2006) have also used buffered communication but have not published a formal semantics. Lanese *et al.* (2007) used asynchronous buffered communication in their stream-based

service-centred calculus (SSCC), but it is a process calculus rather than a functional language.

2. We give a formal proof that the session type of a channel can provide a static upper bound on the size of its buffer, as observed informally by Fähndrich *et al.* (2006). We additionally show that static type information can be used to decrease the runtime buffer size and ultimately deallocate the buffer.
3. 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, operation receive is of type $?T.S \to T \otimes S$ so that the channel, with its new type, is returned with the received value. This gives a huge simplification of our previous work (Vasconcelos *et al.* 2006, 2004) which instead used a complex system of alias types.
4. We include two forms of subtyping – the standard subtyping relation for session types (Gay & Hole 2005) and a novel form of subtyping between standard and linear function types (Gay 2006). The former supports modular development by permitting compatible changes in agents' views of a protocol. The latter reduces the burden of linear typing on the programmer, by allowing standard function types to be inferred by default and converted to linear types if necessary.

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 subtyping relation used in our typing system. Section 5 defines the typing system itself and gives the main results of the paper. Section 7 discusses related and future work.

2 Example: Business protocol

We present a small example containing typical features of many Web service business protocols (W3C 2005; Dezani-Ciancaglini *et al.* 2006). A mother and her young son are using an online bookshop. 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.

Shops only exist because there are shoppers. Shoppers also implement a protocol, where they choose zero or more books followed by checking out, upon which they provide the shop with the credit card details and a delivery address. We write all this as

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

Notice that protocols for shops and shoppers are *compatible*, in the sense that an interaction between the two will not terminate prematurely because of a mismatch in the expectations of one of the partners. A shopper starts by choosing (*selecting* in the terminology of session types) one of two options – add or checkout – and these are exactly the options provided by shops. If the shopper selects option add, she then sends a Book; after accepting option add, a shop expects a Book. For the other option, checkout, shops expect a Card and an Address, in this order, and that is exactly what shoppers provide. After checking out, the run of protocol is terminated for both parties, as indicated in the terminal **end** in each type.

Types Shop and Shopper are also *dual*; the latter can be obtained from the former, by exchanging ! and ?, and ⊕ and &; duality ensures compatibility.

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 or type Shopper, depending on the intended usage. A name such as this is analogous to a URL or an IP address. The access point is used both by the shop and its clients. In order to publish a service, the shop only needs the server capability of the access point, which we have (arbitrarily) chosen to be the *accept* capability. We express this by saying that the shop uses an access point of type $\langle \mathsf{Shop} \rangle^{\mathsf{a}}$, where tag a reminds us of the accept capability. The shopper, on the other hand, will exercise the *request* capability and so uses the *same* access point but with type $\langle \mathsf{Shopper} \rangle^{\mathsf{r}}$. In possession of the accept capability, the shop contains an expression **accept** shopAccess, whereas the shopper exercises their request capability by executing an expression **request** shopAccess. At runtime these expressions interact to create a new private channel, known only to the two threads (shop and shopper).

The shopper (mother, in our example) is implemented as a function parameterised on relevant data including the access point of the shop. Channels, such as c in mother, are *linear* values; session types are linear types. Giving a channel as a parameter to an operator such as **send** removes it from the environment. In order that channels can be used repeatedly for sequences of communication, operations such as **send** 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 that part of the session type is consumed by a communication operation. We discuss alternatives to this repeated re-binding in Section 7.

```
mother :: Card → Address → ⟨Shopper⟩^r → Book → end
mother card address shopAccess book =
  let c = request shopAccess in
  let c = select add c in
  let c = send book c in c
  let c = select checkout c in
  let c = send card c in
  let c = send address c in c
```

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:

```
shop :: ⟨Shop⟩^a → end
shop shopAccess = shopLoop (accept shopAccess) emptyOrder

shopLoop :: Shop → Order → end
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 s
  }
```

The **case** expression combines receiving an option and case-analysis of the option; the code includes a branch for each possibility. Notice that each branch of the **case** is abstracted on a channel. The semantics of **case** is that a channel is read, yielding a label and the channel; the appropriate branch is then applied to the channel. The abstraction therefore re-binds the channel in the same way as **send** and **select** in mother.

We now extend the example, showing that our type system supports programming with higher-order functions on channels in a very natural way. Mother will choose a book for herself and 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, of type Book ⊸Book, plays the role of a gift voucher. Communication between mother and the gift recipient is also described by a session type,

```
Gift = !(Book ⊸ Book).?Book.end
```

where mother sends the voucher and expects back the book chosen by her son. The son, on the other hand, conducts a *dual* protocol, expecting the voucher and replying with a book:

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

As in the case of shopAccess, mother chooses one capability (**request** in the code below) and son the other capability (**accept**) from a *common* access point:

```
mother :: Card → Address → ⟨Shopper⟩^r → ⟨Gift⟩^r → Book → end
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 s
```

The type of voucher again illustrates the linearity of channels. The linear function type constructor ⊸ appears because applying voucher to a channel of type Shop yields a function closure which contains a channel – and 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:

```
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 book


son :: ⟨Son⟩^a → Book → end
son sonAccess book =
  let s = accept sonAccess in
  let (f,s) = receive s in
  let s = send (f book) s in s
```

The complete system is a *configuration* of expressions in parallel, running as separate threads and typed in a suitable environment. The two usages of name shopAccess (that of the shop with type $\langle$Shop$\rangle^{a}$ and that of mother with type $\langle$Shopper$\rangle^{r}$) are reconciled by giving shopAccess the type ⟨Shop,Shopper⟩, which turns out to be a supertype of its two views. We proceed similarly for sonAccess, noting that the type environment below should also include the types of all of the functions used above, as well as mCard and the like:

```
shopAccess:⟨Shop,Shopper⟩, sonAccess:⟨Son,Gift⟩ ⊢
  ⟨shop shopAccess⟩ ∥ ⟨son sonAccess sBook⟩ ∥
  ⟨mother mCard mAddress shopAccess sonAccess mBook⟩
```

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.* (2006) 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, and we give a formal proof in Section 6. For example, the type S in Section 1 yields a bound of 2 because after sending 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. Furthermore, some branches of the protocol may have smaller bounds, and information obtained during typechecking would enable a compiler to generate code to deallocate buffer space; the extreme case is that the compiler can also work out when to completely deallocate the buffer. We should point out, however, that the bound applies to the number of items in the buffer, and unless we can statically bound the size of each item, it does not give a bound on the memory required by the buffer.

We end this section with a few variations of the example, to illustrate subtyping, (explicit) channel sending and forking new threads.

Subtyping function types: 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 Book → Book and Book ⊸ Book. We suggest that a typechecking system should produce the type Book → Book. Because we have Book → Book <: Book ⊸ Book (Section 4), the expression **send** (f book) in the code of son is still typable; there is no need to change the type or the code of son.

The code for mother, however, becomes untypable. Because the new version of the voucher does not invoke checkout, after mother has received son's choice of book, the type of channel c is still Shopper. Moreover, mother still owns c because it was not given as a parameter to voucher'. Mother has to complete the protocol, by checking out after adding zero or more books (possibly including the son's book):

```
mother card address shopAccess sonAccess book =
  ...
  let s = send voucher' s in
  let (sonBook,s) = receive s in
  ...
  let c = select checkout c in
  let c = send card c in
  let c = send address c in c
```

Subtyping session types: adding options to the session type Shop. The shop adds an option to remove a book from the order, changing the session type to

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

We have that NewShop is still *compatible* with the old Shopper; in fact, code following each type can still operate without violating the other side's expectations. Where NewShop offers three options, the old Shopper only takes advantage of two; compatibility rests assured. For this exact reason, we say that the old Shop is a subtype of NewShop. In fact, Shop <: NewShop because the dual of Shop (Shopper above) is compatible with NewShop, as we have just shown.

Subtyping session types: removing options from the session type Shopper. Unkind mother changes her mind. After starting a session with shop, she decides her budget has no room for more books. In order to conform to the protocol, she still has to proceed to checkout, providing her card and address details. The type of shopAccess, as seen from her side, is now

```
UnkindShopper = ⊕⟨checkout:!Card.!Address.end⟩
```

and her code as below:

```
unkindMother :: Card → Address → ⟨UnkindShopper⟩ʳ → end
unkindMother card address shopAccess =
  let c = request shopAccess in
  let c = select checkout c in
  let c = send card c in
  let c = send address c in c
```

Unkind mother is still compatible with the old shop, only that she does not take advantage of the add option. We have that Shopper <: UnkindShopper, since the dual of Shopper (Shop above) is compatible with UnkindShopper as we have just shown.

Can UnkindShopper safely interact with NewShop? Reasoning as for the case of UnkindMother and Shop, we can easily conclude so. Alternatively, we notice that the subtyping direction is inverted by duality. Then, the dual of NewShop is a subtype of the dual of Shop (that is Shopper), and by the transitivity of subtyping we obtain that the dual of NewShop is a subtype of UnkindShopper; hence NewShop and UnkindShopper are compatible. Section 4 discusses the relationship between subtyping, duality and compatibility.

Subtyping access points. The original shop–mother configuration classified the name shopAccess with type ⟨Shop,Shopper⟩. We now have that the new shop uses the access point with type $\langle \mathsf{NewShop}\rangle^{\mathsf{a}}$ and unkind mother with $\langle \mathsf{UnkindShopper}\rangle^{\mathsf{r}}$, making the access point proper of type ⟨NewShop,UnkindShopper⟩. By putting ⟨NewShop,UnkindShopper⟩ <: ⟨Shop,Shopper⟩, we can safely replace shop by the new shop and mother by the unkind mother, thus obtaining the following configuration, where son will never get its **accept** challenge matched:

```
shopAccess:⟨NewShop,UnkindShopper⟩, sonAccess:⟨Son,Gift⟩ ⊢
  ⟨newShop shopAccess⟩ ∥ ⟨son sonAccess sBook⟩ ∥
  ⟨unkindMother mCard mAddress shopAccess⟩
```

Subtyping session types: receiving values of a more general nature. The online business blossoms, and the shop now provides a wide range of products, including books as before but also audio-visual material and electronic gadgets. The new protocol for the shop reflects the change,

```
ProductShop = &⟨add: ?Product.NewShop,
                remove: ?Product.NewShop,
                checkout: ?Card.?Address.end⟩
```

where the shop has made sure that old Book is a subtype of new Product. Once again, the new shop is still compatible with the old shopper, who shops for books alone, thus not taking advantage of the new kinds of items on sale.

Subtyping session types: sending values of a more specific nature. The shop accepts any card as payment; mother now uses a particular kind of card, the Lunchers card, her type becoming

```
LunchersShopper = ⊕⟨checkout:!Lunchers.!Address.end⟩
```

She can still interact safely with any shop, for her card is a particular kind of that accepted by the shop; hence compatibility rests assured. Notice that for output one can replace a type by its subtype, whereas in input one must replace a type by a supertype (as for ProductShop above).

Sending channels on channels: using a third-party shipper. Like previous systems of session types, our type system allows channels to be sent on channels. Implicit channel sending occurs when mother sends son a voucher, as explained above. For a more explicit example, suppose that the shop uses a separate service, shipper, to arrange delivery of orders. 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.

Forking new threads: a multi-threaded shop. The following definition of shop uses **fork** to create a new thread every time a connection is accepted. The definition of shopLoop is unchanged.

```
shop :: ⟨Shop⟩ᵃ → end
shop shopAccess =
  let s = accept shopAccess in
  let x = fork (shopLoop s emptyOrder) in
  shop shopAccess
```

The semantics of **fork** is to create a new thread running shopLoop s emptyOrder and to evaluate to unit in the original thread. In our simple execution model there is no distinction between threads and locations. In a more realistic model we would expect the new thread to run in the same location as shop.

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) *channel endpoints* c and use α to range over both kinds of *identifiers*. We also rely on a set of *labels* l and let n range over $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. *Identifiers* α, *constants* k, *values* v, *expressions* e and *configurations* C are defined as in Figure 1. Values of base types, such as integers, could be added in the same way as unit.

Variable bindings are introduced by λ and by let; channel bindings are introduced by ν. The definition of bound and free identifiers is standard. We work up to alpha-conversion and follow Barendregt's variable convention. The grammar defines let $x, y = e$ in e', which is used for splitting pairs. The form let $x = e$ in e', also used in the examples in Section 2, is syntactic sugar for $(\lambda x.e')e$. We write $fc(C)$ for the *free channels* of a configuration C. Structural congruence, which is the smallest

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

Fig. 1. Syntax.

$$C_1 \parallel C_2 \;\equiv\; C_2 \parallel C_1 \qquad C_1 \parallel (C_2 \parallel C_3) \;\equiv\; (C_1 \parallel C_2) \parallel C_3 \qquad \text{(E-COMM,E-ASSOC)}$$

$$C_1 \parallel (\nu cd)C_2 \;\equiv\; (\nu cd)(C_1 \parallel C_2)\ \text{ if } c,d \notin fc(C_1) \qquad \text{(E-SCOPE)}$$

Fig. 2. Structural congruence.

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

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

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

Fig. 3. Reduction of expressions.

$$\frac{e \longrightarrow_{\mathsf{v}} e'}{\langle E[e]\rangle \longrightarrow \langle E[e']\rangle} \qquad \langle E[\mathsf{fork}\ e]\rangle \longrightarrow \langle e\rangle \parallel \langle E[\mathsf{unit}]\rangle \qquad \text{(R-THREAD,R-FORK)}$$

$$\frac{C \longrightarrow C'}{C \parallel C'' \longrightarrow C' \parallel C''} \qquad \frac{C \longrightarrow C'}{(\nu cd)C \longrightarrow (\nu cd)C'} \qquad \frac{C \equiv C' \quad C' \longrightarrow C'' \quad C'' \equiv C'''}{C \longrightarrow C'''}$$

$$\text{(R-PAR,R-NEW,R-STRUCT)}$$

$$\langle E[\mathsf{request}\ n\ x]\rangle \parallel \langle E'[\mathsf{accept}\ n'\ x]\rangle \longrightarrow \\ (\nu cd)(c \mapsto (d,n,\varepsilon) \parallel d \mapsto (c,n',\varepsilon) \parallel \langle E[c]\rangle \parallel \langle E'[d]\rangle) \qquad \text{(R-INIT)}$$

$$c \mapsto (d,n',\vec{b'}) \parallel d \mapsto (c,n,\vec{b}) \parallel \langle E[\mathsf{send}\ v\ c]\rangle \longrightarrow \\ c \mapsto (d,n',\vec{b'}) \parallel d \mapsto (c,n,\vec{b}v) \parallel \langle E[c]\rangle \quad \text{if } |\vec{b}| < n \qquad \text{(R-SEND)}$$

$$c \mapsto (d,n,\vec{b'}) \parallel d \mapsto (c,n,\vec{b}) \parallel \langle E[\mathsf{select}\ l\ c]\rangle \longrightarrow \\ c \mapsto (d,n',\vec{b'}) \parallel d \mapsto (c,n,\vec{b}l) \parallel \langle E[c]\rangle \quad \text{if } |\vec{b}| < n \qquad \text{(R-SELECT)}$$

$$c \mapsto (d,n,v\vec{b}) \parallel \langle E[\mathsf{receive}\ c]\rangle \longrightarrow c \mapsto (d,n,\vec{b}) \parallel \langle E[(v,c)]\rangle \qquad \text{(R-RECEIVE)}$$

$$c \mapsto (d,n,l_j\vec{b'}) \parallel \langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}]\rangle \longrightarrow c \mapsto (d,n,\vec{b}) \parallel \langle E[e_j c]\rangle \quad \text{if } j \in I \qquad \text{(R-BRANCH)}$$

Fig. 4. Reduction of configurations.

relation that satisfies the rules in Figure 2 and is also a congruence with respect to ν and parallel composition, allows changing the syntactic order of the components in a configuration.

The operational semantics of the language is defined via the reduction relation in Figures 3 and 4. Figure 3 defines reduction of expressions by means of standard rules. To simplify the presentation of inter-thread reduction, we use evaluation contexts

(Figure 1) (Wright & Felleisen 1994) and structural equivalence on configurations (Figure 2). An evaluation context is an expression with a hole, denoted [ ], where computation happens next. The syntax $E[e]$ denotes the result of filling the hole of context E with expression e.

Figure 4 presents the rules for inter-thread, or configuration, reduction. Rule R-Thread allows reduction within the hole of a thread; rule R-Fork spawns a new thread. Rules R-Par, R-New and R-Struct isolate threads that will engage in inter-thread communication via the remaining rules.

As well as threads, a configuration contains buffers. The buffer for endpoint c is represented by $c \mapsto (d, n, \vec{b})$. Here d is another channel, called the *peer endpoint* of c; n is the size of the buffer; and $\vec{b}$ is the data in the buffer, called the *channel queue*. The data $\vec{b}$ is a list of items in which each item is either a value v (written and read by send and receive expressions) or a label l (written and read by select and case expressions). We write $|\vec{b}|$ for the number of items in $\vec{b}$.

Rule R-Init synchronises two threads trying to start a new connection on a common name x, which must be free in each thread because of the variable convention that we assume. In our type system, names on which connections can be created are treated as free variables; so we do not have a separate syntactic category for them. R-Init creates two new endpoints, c and d, one for each thread. It also creates two new buffers, each mentioning its peer endpoint and containing the buffer size declared by request or accept. Symbol ε denotes an empty queue. (The example of Section 2 omitted the buffer sizes because they can be inferred; see Section 5).

Rules R-Send and R-Select write on the peer endpoint of c: a value v in the case of R-Send and a label l in the case of R-Select. The result is c, which can be used for further interaction. Notice that these two rules require an indirection step in order to obtain the peer's endpoint channel d from the thread's endpoint c. Further, notice that the semantics explicitly tests for space in the buffer; our type system makes this test redundant (see Section 6).

Rules R-Receive and R-Branch read from the head of the channel queue: value v for R-Receive and label l_j for R-Case. The result of receive c is a pair composed of v and the channel c itself. The result of case c of $\{l_i : e_i\}_{i \in I}$ is the application of the function e_j, the body of the branch labelled by l_j, to channel c (recall from Section 2 that the branch is abstracted over the channel). In either case, again, c can be used for further interaction.

4 Types, subtyping and bounds

This section introduces types, the subtyping relation and the notion of the bound of a session type.

The syntax of types is defined in Figure 5. *Session types* S are associated with channels. Now, end is the type of a channel which cannot be used for further communication. Furthermore, $?T.S$ is the type of a channel from which a message of type T can be received, and subsequently the channel is described by type S. Dually, $!T.S$ is the type of a channel on which a message of type T can be sent;

$$\begin{aligned}
T &::= S \mid D \mid T \otimes T \mid T \rightarrow T \mid T \multimap T \mid \langle S \rangle^{\mathsf{r}} \mid \langle S \rangle^{\mathsf{a}} \mid \langle S, S' \rangle \\
S &::= \mathsf{end} \mid ?T.S \mid !T.S \mid \&\langle l_i \colon S_i \rangle_{i \in I} \mid \oplus \langle l_i \colon S_i \rangle_{i \in I} \mid X \mid \mu X.S \\
D &::= \mathsf{Unit} \\
B &::= T \mid l
\end{aligned}$$

Fig. 5. Syntax of types and session types.

subsequently the type of the channel is S. In addition, $\&\langle l_i : S_i \rangle_{i \in I}$ is the type of a channel from which a message can be received, which will be one of the set of distinct labels l_i. The subsequent behaviour of the channel is described by the corresponding type S_i. Dually, $\oplus\langle l_i : S_i \rangle_{i \in I}$ is the type of a channel on which one of the labels l_i can be sent, with subsequent behaviour described by S_i.

We include *recursive session types* $\mu X.S$, which are required to be *contractive*, i.e. containing no subexpression of the form $\mu X_1.\cdots \mu X_n.X_1$. The μ operator is a binder, giving rise, in the standard way, to notions of bound and free variables and alpha-equivalence. A type is *closed* if it includes no free variables. We denote by $T\{U/X\}$ the capture-avoiding substitution of U for X in T.

General types are denoted by T, including session types S as one case. Type $T \otimes U$ denotes the type of a pair composed of an element of type T and an element of type U. Type $T \rightarrow U$ denotes a conventional function from values of type T into values of type U. Type $T \multimap U$ describes a linear function, i.e. a function that is itself a linear value. Whether the parameter must be used exactly once depends on whether T is a linear type.

As for session types, $\langle S \rangle^{\mathsf{r}}$ describes an access point that can only be used to *request* the establishment of a session. Similarly, $\langle S \rangle^{\mathsf{a}}$ describes an access point that can only be used to *accept* a connection. An access point which can be used to request a connection of type S and to accept a connection of type S' is denoted by $\langle S, S' \rangle$. The two types, S and S', are supposed to be *compatible*, a notion introduced below. If a typed access point $a \colon \langle S, S' \rangle$ occurs in the global environment, then a matching request n a and accept n' a create a channel. On one side, accept yields a channel endpoint of type S, while on the other side, request yields the peer endpoint whose type is S'. Base types such as Int and Bool can be added in the same way as Unit. Compound data types such as non-linear pairs, or general recursive types, can also easily be added. The definition of B in Figure 5 is used in Section 5 to type the contents of buffers.

We let $\mathscr{S}$ denote the set of contractive, closed session types and $\mathscr{T}$ the set of types in which all session types are contractive and closed.

The type system includes a *subtyping relation*. This combines the standard definition of subtyping for session types (Gay & Hole 2005), the standard subtyping rules for function types and pairs (Pierce 2002) and the novel relationship $T \rightarrow U <: T \multimap U$ between standard and linear function types (Gay 2006). The key features of subtyping for session types are that $?T.S$ is covariant in T; $!T.S$ is contravariant in T; $\&\langle l_i : S_i \rangle_{i \in I}$ is covariant in I; $\oplus\langle l_i : S_i \rangle_{i \in I}$ is contravariant in I; and they are all covariant in S and in each S_i.

Definition 1 (*Subtyping*)
Define the operator $F \in \mathscr{P}(\mathscr{T} \times \mathscr{T}) \to \mathscr{P}(\mathscr{T} \times \mathscr{T})$ as follows:

$$
\begin{aligned}
F(R) = {} & \{(\mathsf{end}, \mathsf{end})\} \\
& \cup \{(?T.S, ?T'.S') \mid (T, T'), (S, S') \in R\} \\
& \cup \{(!T.S, !T'.S') \mid (T', T), (S, S') \in R\} \\
& \cup \{(\&\langle l_i : S_i\rangle_{i\in I}, \&\langle l_j : S'_j\rangle_{j\in J}) \mid I \subseteq J, (S_i, S'_i) \in R, \forall i \in I\} \\
& \cup \{(\oplus\langle l_i : S_i\rangle_{i\in I}, \oplus\langle l_j : S'_j\rangle_{j\in J}) \mid J \subseteq I, (S_i, S'_i) \in R, \forall i \in J\} \\
& \cup \{(\langle S, S'\rangle, \langle S\rangle^{\mathsf{a}}), (\langle S', S\rangle, \langle S\rangle^{\mathsf{r}}) \mid S, S' \in \mathscr{S}\} \\
& \cup \{(\langle S\rangle^{\mathsf{a}}, \langle S'\rangle^{\mathsf{a}}) \mid (S, S') \in R\} \\
& \cup \{(\langle S\rangle^{\mathsf{r}}, \langle S'\rangle^{\mathsf{r}}) \mid (S, S') \in R\} \\
& \cup \{(\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle) \mid (S_1, S_2), (S_1, S'_2) \in R\} \\
& \cup \{(T_1 \to T'_1, T_1 \multimap T'_1) \mid T_1, T'_1 \in \mathscr{T}\} \\
& \cup \{(T_1 \to T'_1, T_2 \to T'_2) \mid (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(T_1 \multimap T'_1, T_2 \multimap T'_2) \mid (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(\mu X.S, S') \mid (S\{\mu X.S/X\}, S') \in R\} \\
& \cup \{(S, \mu X.S') \mid (S, S'\{\mu X.S'/X\}) \in R\}
\end{aligned}
$$

Contractivity ensures that F is monotone. By the Knaster–Tarski theorem, F has least and greatest fixed points; we take the greatest fixed point to be the subtyping relation, writing $T <: U$ if the pair (T, U) is in the relation.

We define *equivalence of types* T and U as $T <: U$ and $U <: T$. Henceforth types are understood up to type equivalence, so that for example, in any mathematical context, types $\mu X.T$ and $T\{(\mu X.T)/X\}$ can be used interchangeably, effectively adopting the equi-recursive approach (Pierce, 2002, Chapter 21).

When restricted to session types, the subtyping relation we use is essentially that of Gay & Hole (2005) (defined via a type simulation) and that of Vallecillo *et al.* (2006) (defined algorithmically). Yoshida & Vasconcelos (2007) present a co-inductive definition of type equivalence, similarly to what we do above for subtyping.

The definition of subtyping on branch (&) and choice (⊕) types is a little counter-intuitive. It is tempting to think that the inclusion relationship between I and J should be in the opposite direction in each case of Definition 1. Consider the example of NewShop from Section 2. A shop that follows the protocol NewShop, meaning that it expects to communicate on a channel of type NewShop, must contain a **case** with branches add, remove and checkout. If, however, the shop is given a channel of type Shop, in which the branching constructor contains only the labels add and checkout, correct communication still occurs; the shop is able to respond to every label that can be received from the channel. Therefore the subtyping relationship Shop $<:$ NewShop is consistent with the usual principle of safe substitutability. Technically, if we have s: NewShop $\vdash$ shopcode, then it is safe to conclude s: Shop $\vdash$ shopcode, replacing a type by its subtype in the environment. Dually, the relationship Shopper $<:$ UnkindShopper is correct. A shopper who only selects the checkout label,

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

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

Fig. 6. The dual function on session types.

and therefore expects to use a channel of type UnkindShopper, is still safe if given a channel of type Shopper.

Proposition 2
Subtyping is a pre-order.

Proof
We prove reflexivity and transitivity by standard coinductive arguments, as an instance of the general approach in Theorems 21.3.6 and 21.3.7 of Pierce (2002). Reflexivity and transitivity of subtyping on session types have been proved explicitly by Gay and Hole (2005), and transitivity of a similar (equivalence) relation on session types has been proved explicitly by Yoshida and Vasconcelos (2007). □

Duality is a central concept in the theory of session types. The function $\overline{S}$, defined in Figure 6, yields the canonical dual of a session type S. Previous work by Gay and Hole (2005) and Vallecillo *et al.* (2006) defined a duality relation coinductively. Here we just write $S = \overline{S'}$ on the understanding that we are always working up to type equivalence, so that for example, $\overline{\mu X.\&\langle l : X\rangle} = \oplus\langle l : \mu Y.\oplus\langle l : Y\rangle\rangle$.

Equipped with the notions of subtyping and duality, we say that session types S and S' are *compatible*, written $S \asymp S'$, when $\overline{S} <: S'$. Henceforth we assume that in a type $\langle S, S'\rangle$, session types S and S' are always compatible. The following results on the triangle subtyping-duality-compatibility follow Vallecillo *et al.* (2006).[1]

Proposition 3
1. Duality is self-inverse.
2. $S_1 <: S_2$ if and only if $\overline{S_2} <: \overline{S_1}$.
3. Compatibility is symmetric, not reflexive and/or transitive.
4. If $S_1 \asymp S_2$ and $S_2 <: S_3$, then $S_1 \asymp S_3$.

Proof
1. Directly from the definition.
2. By a coinductive argument based on Definition 1.
3. Symmetry follows from the definition and (2). Lack of reflexivity and transitivity is because duality changes the outermost type constructor.
4. Follows from transitivity of subtyping. □

Describing protocols, session types ‘advance’ during computation. The reduction relation on session types (Figure 7) makes this notion precise. Note that the relation is defined on syntactic session type expressions, not on equivalence classes of session types. The bound of a session type S, if this bound is finite, gives an upper bound

[1] For technical reasons the definition of compatibility appears reversed with respect to that of Vallecillo *et al.* (2006).

$$?T.S \mapsto S \qquad !T.S \mapsto S \qquad \&\langle \dots, l : S, \dots \rangle \mapsto S \qquad \oplus\langle \dots, l : S, \dots \rangle \mapsto S$$
$$\mu X.S \mapsto S' \text{ if } S\{\mu X.S/X\} \mapsto S'$$

Fig. 7. Relation $\mapsto$ on session types.

on the runtime size of the buffer holding the values received on a channel of type S. If the bound of S is infinite, then the size of the buffer cannot be bounded, and a buffer of size ∞, i.e. an unbounded buffer, must be used. Notice that S and $\overline{S}$ will in general have different bounds. We emphasise that 'size' means the number of items in the buffer and does not give an upper bound on the memory requirement unless the size of messages is also bounded.

Definition 4 (*Bound of a session type*)
The set of maps $\mathcal{S} \to \mathbb{N}^\infty$ is a complete lattice if we define $f \sqsubseteq g$ to mean $f(S) \leqslant g(S)$, $\forall S \in \mathcal{S}$ and take meets and joins pointwise. The bottom function maps everything to 0, and the top function maps everything to ∞. We also define $\infty + 1 = \infty$ and $max(n, \infty) = \infty$, for every $n \in \mathbb{N}^\infty$.

Define the operator $B \in (\mathcal{S} \to \mathbb{N}^\infty) \to \mathcal{S} \to \mathbb{N}^\infty$ as follows:

$$B(f)(!T.S) = 0 \qquad B(f)(?T.S) = 1 + f(S)$$
$$B(f)(\oplus\langle l_i : S_i\rangle_{i\in I}) = 0 \qquad B(f)(\&\langle l_i : S_i\rangle_{i\in I}) = 1 + max\{f(S_i)\}_{i\in I}$$
$$B(f)(\mathsf{end}) = 0 \qquad B(f)(\mu X.S) = f(S\{\mu X.S/S\})$$

Contractivity ensures that B is monotone. The Knaster–Tarski theorem gives least and greatest fixed points of B.[2] Define bound$(S) = \max\{\mu(S')|S \mapsto^* S'\}$, where μ is the least fixed point of B.

The definition yields an algorithm for calculating bound(S). Construct a directed graph with $\{S'|S \mapsto^* S'\}$ as the vertices and $\mapsto$ as the edge relation. The set of vertices is finite because recursive are only unfolded on demand – the number of vertices is bounded by the number of constructors in S. Label every end, $!T.S$ and $\oplus\langle l_i : S_i\rangle_{i\in I}$ vertex with 0. Iterate the following steps until a fixed point is reached: label vertex $?T.S$ with $n+1$ if S is labelled with n and label vertex $\&\langle l_i : S_i\rangle_{i\in I}$ with $max\{n_i\}_{i\in I}$ if every S_i is labelled with n_i. This process terminates because labels are never changed; so each vertex is labelled at most once. Finally label any unlabelled nodes with ∞. bound(S) is the largest label.

The main property of the bound of a type is that it does not grow with reduction, a fact exploited by type preservation (Theorem 24).

Lemma 5
For all session types S and S', if $S \mapsto S'$, then bound$(S') \leqslant$ bound(S).

Proof
Let μ be the least fixed point of B, as defined in Definition 4. Furthermore, bound$(S) = max\{\mu(T)|S \mapsto^* T\}$ and bound$(S') = max\{\mu(T)|S' \mapsto^* T\}$. Because $S \mapsto S'$, $\{\mu(T)|S' \mapsto^* T\} \subseteq \{\mu(T)|S \mapsto^* T\}$. The result follows. □

[2] It turns out that the greatest and least fixed points coincide, but we do not need this fact.

$$\mathsf{lin}(S \neq \mathsf{end}) \qquad \mathsf{lin}(T \otimes T) \qquad \mathsf{lin}(T \multimap T)$$
$$\mathsf{un}(\mathsf{end}) \qquad \mathsf{un}(T \to T) \qquad \mathsf{un}(\langle S, \overline{S} \rangle) \qquad \mathsf{un}(\langle S \rangle^{\mathsf{a}}) \qquad \mathsf{un}(\langle S \rangle^{\mathsf{r}})$$

Fig. 8. Type classification as linear (lin) or unlimited (un).

$$\begin{array}{rlrl}
\mathsf{fix}: & (T \to T) \to T & \mathsf{receive}: & ?T.S \to T \otimes S \\
\mathsf{send}: & T \to\ !T.S \multimap S & \mathsf{request}\ n: & \langle S \rangle^{\mathsf{r}} \to \overline{S} \quad \text{if bound}(\overline{S}) \leqslant n \\
\mathsf{send}: & T \to\ !T.S \to S \quad \text{if } \mathsf{un}(T) & \mathsf{accept}\ n: & \langle S \rangle^{\mathsf{a}} \to S \quad \text{if bound}(S) \leqslant n \\
\mathsf{fork}: & T \to \mathsf{Unit} \quad \text{if } \mathsf{un}(T) & \mathsf{unit}: & \mathsf{Unit}
\end{array}$$

Fig. 9. Type schemas for constants k.

5 Typing

This section introduces a static type system for our language.

Because channels must be controlled linearly, so that each endpoint 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 classified as either *linear* or *unlimited*, as defined in Figure 8. Type end is unlimited because we do not explicitly close channels.

Type environments are finite maps from variables or channels (collectively written α) into types. Write dom(Γ) for the set of variables and channels in Γ and cdom(Γ) for the set of channels in Γ, and say that un(Γ) is true of an environment in which all types are unlimited. In the usual way for a type system with linear types (Walker 2005), we define a partial operation of addition on environments:

$$\Gamma + \alpha : T = \begin{cases} \Gamma, \alpha : T & \text{if } \alpha \notin \text{dom}(\Gamma) \\ \Gamma & \text{if } \alpha : T \in \Gamma \text{ and } \mathsf{un}(T) \\ \text{undefined} & \text{otherwise} \end{cases}$$

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 are schemas which can be instantiated for any appropriate type. 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. Channel passing constitutes a particular case of the latter. The receive function is given a channel of appropriate type and returns, together 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 functions request n and accept n return each a new endpoint of the corresponding type if the size of the buffer necessary to hold all the values produced does not exceed n. It is possible for n to be ∞, corresponding to an unbounded

$$\frac{\mathsf{un}(\Gamma) \quad k\colon T}{\Gamma \vdash k\colon T} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, \alpha\colon T \vdash \alpha\colon T} \qquad \frac{\Gamma \vdash e\colon T \quad T <: U}{\Gamma \vdash e\colon U} \qquad \text{(T-CONST,T-ID,T-SUB)}$$

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

$$\frac{\Gamma, x\colon T \vdash e\colon U \quad \mathsf{un}(\Gamma)}{\Gamma \vdash \lambda x.e\colon T \to U} \qquad \frac{\Gamma, x\colon T \vdash e\colon U}{\Gamma \vdash \lambda x.e\colon T \multimap U} \qquad \frac{\Gamma_1 \vdash e_1 : T \multimap U \quad \Gamma_2 \vdash e_2 : T}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 \colon U}$$

(T-ABS,T-ABSL,T-APP)

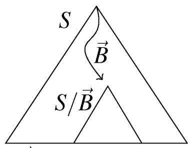
$$\frac{\Gamma \vdash e\colon \oplus\langle l_i\colon T_i\rangle_{i\in I} \quad j \in I}{\Gamma \vdash \mathsf{select}\; l_j\; e\colon T_j} \qquad \frac{\Gamma_1 \vdash e\colon \&\langle l_i\colon T_i\rangle_{i\in I} \quad \forall_{i\in I}(\Gamma_2 \vdash e_i\colon T_i \multimap T)}{\Gamma_1 + \Gamma_2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

(T-SELECT,T-CASE)

Fig. 10. Typing rules for expressions.

$$\varepsilon \,\text{matches}\, S \qquad \frac{\vec{B}\,\text{matches}\, S \quad U <: T}{U\vec{B}\,\text{matches}\, ?T.S} \qquad \frac{\vec{B}\,\text{matches}\, S}{l\vec{B}\,\text{matches}\, \&\langle ..l\colon S..\rangle}$$

$$\begin{aligned} S/\varepsilon &= S \\ ?T.S/U\vec{B} &= S/\vec{B} \\ \&\langle \ldots, l\colon S, \ldots\rangle/l\vec{B} &= S/\vec{B} \end{aligned}$$



If $\vec{B}$ 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 $S/\vec{B}$.

Fig. 11. The matches relation.

buffer, and this would be necessary in the case of a session type with infinite bound. The type of the expression given to fork is required to be unlimited in order to ensure that the new thread completely consumes any channel that it uses. Most of the rules in Figure 10 are standard. Values of other base types can be included in the same way as unit : Unit. Note that by using rule T-SUB after T-ABS, a standard function can be given a linear function type if desired. This means that although T-APP requires a linear function type, it can also be used to apply standard functions. T-SELECT is like the typing schema for send but is expressed as a rule because the result type depends on the label. 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).

Figure 11 defines two notions. First, $\vec{B}$ matches S means that the sequence of types and labels $\vec{B}$ can describe an initial sequence of inputs and branches in S. In that case, $S/\vec{B}$ is the remaining session type. These notions are used to characterise the relationship between the types of endpoints and the contents of their buffers.

Figure 12 defines typing of configurations. Sequents are of the form $\Gamma \vdash C \triangleright \Delta$, where Δ contains the buffer entries in C. More precisely, Δ is a map from channels into *buffer types* $(d, n, \vec{B})$, endowed with a partial operation $+$ of disjoint union. Again $\vec{B}$ is a sequence of types and labels.

T-THREAD begins with a single thread (containing an expression), which must have an unlimited type, since we expect all sessions to be taken to the end. T-BUFFER types a buffer for a channel, assigning type $\vec{B}$ to data $\vec{b}$ and checking that the buffer's

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \varepsilon \colon \varepsilon} \quad \frac{\Gamma_1 \vdash v \colon T \quad \Gamma_2 \vdash \vec{b} \colon \vec{B}}{\Gamma_1 + \Gamma_2 \vdash v\vec{b} \colon T\vec{B}} \quad \frac{\Gamma \vdash \vec{b} \colon \vec{B}}{\Gamma \vdash l\vec{b} \colon l\vec{B}} \qquad \text{(T-EMPTY,T-SEQV,T-SEQL)}$$

$$\frac{\Gamma \vdash e \colon T \quad \mathsf{un}(T)}{\Gamma \vdash \langle e \rangle \triangleright \emptyset} \quad \frac{\Gamma \vdash \vec{b} \colon \vec{B} \quad |\vec{b}| \leqslant n}{\Gamma \vdash c \mapsto (d, n, \vec{b}) \triangleright c \colon (d, n, \vec{B})} \qquad \text{(T-THREAD,T-BUFFER)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash C_1 \triangleright \Delta_1 \quad \Gamma_2 \vdash C_2 \triangleright \Delta_2 \quad \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta = \Delta_1 + \Delta_2 \\ \forall c \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta).(\Delta(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}\,\mathrm{matches}\,\Gamma(c) \text{ and } \mathrm{bound}(\Gamma(c)) \leqslant n)) \\ \forall c, d \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta).(\Delta(c) = (d, n, \vec{B}) \text{ and } \Delta(d) = (c, n', \vec{B}') \Rightarrow \Gamma(c)/\vec{B} \asymp \Gamma(d)/\vec{B}')\end{array}}{\Gamma \vdash C_1 \parallel C_2 \triangleright \Delta}$$

(T-PAR)

$$\frac{\Gamma + c_1 \colon S_1 + c_2 \colon S_2 \vdash C \triangleright \Delta + c_1 \colon (c_2, n_1, \vec{B}_1) + c_2 \colon (c_1, n_2, \vec{B}_2)}{\Gamma \vdash (\nu c_1 c_2) C \triangleright \Delta} \qquad \text{(T-NEW)}$$

Fig. 12. Typing rules for configurations.

capacity is not exceeded. T-PAR combines configurations in parallel, by combining the environments and the buffers in each configuration. There are three further checks: (1) that the session type of a channel is matched by the type of the data in its buffer; (2) that the capacity of a channel's buffer is at least as large as the bound of the channel's session type; (3) that the session types of the endpoints of a channel, after the buffers have been emptied, are dual. Finally, rule T-NEW simply removes private channel endpoints and their buffers from the typing environments.

6 Type safety

In this section we prove that our type system guarantees safe execution of programs. The safety property is a version of the usual statement that well-typed programs do not get stuck. We formulate 'getting stuck' in terms of *blocked threads*.

Definition 6 (*Buffers in configurations*)
If $C \equiv (\nu c_1 c_1') \ldots (\nu c_n c_n')(c \mapsto (c', k, \vec{b}) \parallel C')$, then we write $c \mapsto (c', k, \vec{b}) \in C$ or just $c \in C$.

Definition 7 (*Blocked thread*)
Let C be a configuration and C' one of its threads; C' is *blocked* if there do not exist $c_1 \mapsto (c_2, n_1, \vec{b}_1), c_2 \mapsto (c_1, n_2, \vec{b}_2) \in C$ and C'' such that $C' \parallel c_1 \mapsto (c_2, n_1, \vec{b}_1) \parallel c_2 \mapsto (c_1, n_2, \vec{b}_2) \longrightarrow C''$.

By analysing the reduction rules, we see that a thread can be blocked in several ways: trying to read from a channel when there is no data in the buffer; trying to send on a channel when the buffer is full; trying to communicate when the required channel does not exist; reading an inappropriate value from a channel; trying to evaluate an expression for which there is no reduction rule; or simply when it terminates execution.

The runtime safety theorem states that the type system guarantees that a thread can only become blocked by terminating, or by trying to read from an empty buffer, or by trying to request or accept a connection when there is no matching partner.

Typability of the expressions in threads is not sufficient to guarantee runtime safety. For example, $\langle \mathsf{send}\ x\ c\rangle$ is typable but cannot progress because of the absence of buffers for c and d (cf. rule R-Send in Figure 4). Similarly $\langle \mathsf{let}\ x, d = \mathsf{receive}\ c\ \mathsf{in}\ d\rangle \parallel c \mapsto (\_, \_, l)$ is typable but cannot progress because labels are not values (cf. rule R-Receive in Figure 4). The runtime safety theorem applies to typable configurations that also satisfy the following property. As we will see, it holds for our intended initial configurations and is preserved by reduction.

Definition 8 (*Fully buffered configuration*)
A configuration C is *fully buffered* if whenever $c_1 \mapsto (c_2, n_1, \vec{b_1}) \in C$ there exist $n_2, \vec{b_2}$ such that $c_2 \mapsto (c_1, n_2, \vec{b_2}) \in C$.

Considering the two examples above, the first is not a violation of the runtime safety theorem because the configuration is not fully buffered; the second is not a violation because although the thread is typable, the configuration is not typable (because the label l cannot match the type T returned by $\mathsf{receive}$).

We can now state the runtime safety theorem.

Theorem (Runtime safety)
Let $\Gamma \vdash C \triangleright \Delta$ be fully buffered, and assume that $C \longrightarrow^* C'$. If C'' is a blocked thread in C', then one of the following applies:

1. C'' is $\langle v\rangle$ or $\langle \mathsf{send}\ v\rangle$ or $\langle \mathsf{request}\ n\ x\rangle$ or $\langle \mathsf{accept}\ n\ x\rangle$;
2. C'' is $\langle E[\mathsf{receive}\ c]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$;
3. C'' is $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i : e_i\}_{i\in I}]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$.

The presence of $\langle \mathsf{send}\ v\rangle$ in the first clause of the runtime safety theorem is due to the fact that send has a curried type. The partial application $\mathsf{send}\ v$ cannot reduce because a channel has not been given. This does not represent a failure of communication – the blocking is not because of lack of buffer space or a runtime type error – and so it is not eliminated by the type system. But $\mathsf{send}\ v$ is not syntactically a value; so it requires a special case in the statement of runtime safety. The alternative would be to use the typing $\mathsf{send} : T \otimes\ !T.S \multimap S$ instead.

It is interesting to observe that previous work (Gay & Hole 2005) required, in the statements of type preservation and type safety, a global condition that the two endpoints of every channel have compatible types. In the present paper, this global condition has been replaced by conditions in the typing rule T-Par; the global condition becomes Lemma 21 instead of an assumption. This is possible because we have asynchronous buffered communication. Compatibility between the endpoints of a channel can only be stated after the buffers have been added to the configuration, because it is the buffers that introduce the connection between the endpoints. Rule T-Par is used to combine the buffers with the threads; so it can also be used to check compatibility.

Our type system does not guarantee deadlock-freedom. It is possible for two threads to be blocked waiting for input, where the matching outputs exist but are behind the inputs in the same two threads. For example, the following code is typable with a suitable choice of session types:

$$\langle \mathsf{let}\ c_1 = \mathsf{request}\ a_1\ \mathsf{in}\ \mathsf{let}\ c_2 = \mathsf{request}\ a_2\ \mathsf{in}\ \mathsf{let}\ (c_1, x) = \mathsf{receive}\ c_1\ \mathsf{in}\ \mathsf{send}\ v\ c_2\rangle$$
$$\|$$
$$\langle \mathsf{let}\ d_1 = \mathsf{accept}\ a_1\ \mathsf{in}\ \mathsf{let}\ d_2 = \mathsf{accept}\ a_2\ \mathsf{in}\ \mathsf{let}\ (d_1, y) = \mathsf{receive}\ d_2\ \mathsf{in}\ \mathsf{send}\ w\ d_1\rangle$$

This example is artificial, but similar situations can easily arise when channels are passed from thread to thread. Dezani-Ciancaglini *et al.* (2006) and Coppo *et al.* (2007) have defined type systems which guarantee strong progress or deadlock-freedom properties but at the expense of not allowing sessions to be interleaved within a single thread. We take the view that interleaving of sessions is important for expressivity; so our type system simply guarantees that the necessary communication operations for every session are present in the correct sequence.

In order to prove runtime safety we make use of a type preservation theorem. The most important part of the theorem is stated below. Later we will use a stronger statement in order to explain the relationships between Γ and Γ' and between Δ and Δ'.

Theorem (Type preservation)
If $\Gamma \vdash C \triangleright \Delta$ and $C \longrightarrow C'$, then there exist Γ' and Δ' such that $\Gamma' \vdash C' \triangleright \Delta'$.

We will now work towards the proofs of type preservation and runtime safety. The structure of the proof of type preservation follows the approach of Wright and Felleisen (1994). We omit the proofs of most lemmas, which either are easy structural inductions or follow directly from definitions.

Lemma 9
If $\Gamma \vdash C \triangleright \Delta$ and $C \equiv C'$, then $\Gamma \vdash C' \triangleright \Delta$.

Lemma 10 (*Weakening*)
If $\Gamma_1 \vdash e : T$ and $\mathsf{un}(\Gamma_2)$ and $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1 + \Gamma_2 \vdash e : T$.

Lemma 11
If $\Gamma \vdash v : T$ and $\mathsf{un}(T)$, then $\mathsf{un}(\Gamma)$.

Lemma 12 (*Typability of Subterms*)
If $\mathscr{D}$ is a derivation of $\Gamma \vdash E[e] : T$ or $\Gamma \vdash \langle E[e]\rangle \triangleright \emptyset$, then there exist Γ_1, Γ_2 and U such that $\Gamma = \Gamma_1 + \Gamma_2$, $\mathscr{D}$ has a subderivation $\mathscr{D}'$ concluding $\Gamma_1 \vdash e : U$ and the position of $\mathscr{D}'$ in $\mathscr{D}$ corresponds to the position of the hole in $E[\,]$.

Lemma 13 (*Replacement*)
If

1. $\mathscr{D}$ is a derivation of $\Gamma_1 + \Gamma_2 \vdash E[e] : T$ or $\Gamma_1 + \Gamma_2 \vdash \langle E[e]\rangle \triangleright \emptyset$,
2. $\mathscr{D}'$ is a subderivation of $\mathscr{D}$ concluding $\Gamma_2 \vdash e : U$,
3. the position of $\mathscr{D}'$ in $\mathscr{D}$ corresponds to the position of the hole in $E[\,]$,
4. $\Gamma_3 \vdash e' : U$,
5. $\Gamma_1 + \Gamma_3$ is defined,

then $\Gamma_1 + \Gamma_3 \vdash E[e'] : T$ or $\Gamma_1 + \Gamma_3 \vdash \langle E[e']\rangle \triangleright \emptyset$ as appropriate.

Lemma 14 (Substitution)
If $\Gamma_1, x\colon T \vdash e\colon U$ and $\Gamma_2 \vdash e'\colon V$ and $V <: T$ and $(\mathsf{un}(T) \implies \mathsf{un}(\Gamma_2))$ and $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1 + \Gamma_2 \vdash e\{e'/x\}\colon U$.

Lemma 15
If $\Gamma \vdash \lambda x.e\colon T \to U$, then there is a derivation in which the last rule is T-Abs.

Proof
By induction on the derivation of $\Gamma \vdash \lambda x.e\colon T \to U$. The last rule can only be T-Abs or T-Sub. If it is T-Abs we have finished. So suppose we have

$$\frac{\Gamma \vdash \lambda x.e\colon T' \to U' \quad T <: T' \quad U' <: U}{\Gamma \vdash \lambda x.e\colon T \to U}$$

By the induction hypothesis there is a derivation of $\Gamma \vdash \lambda x.e\colon T' \to U'$ in which the last rule is T-Abs:

$$\frac{\Gamma, x\colon T' \vdash e\colon U' \quad \mathsf{un}(\Gamma)}{\Gamma \vdash \lambda x.e\colon T' \to U'}$$

By Substitution (Lemma 14) we have $\Gamma, x\colon T \vdash e\colon U'$, and then we can construct the required derivation:

$$\frac{\dfrac{\Gamma, x\colon T \vdash e\colon U' \quad U' <: U}{\Gamma, x\colon T \vdash e\colon U} \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash \lambda x.e\colon T \to U}$$

□

Lemma 16
If $\Gamma \vdash \lambda x.e\colon T \multimap U$, then there is a derivation in which the last rule is T-AbsL.

Proof
Similar to the proof of Lemma 15. There is an additional case in which the final instance of T-Sub involves subtyping between a standard and a linear function type:

$$\frac{\Gamma \vdash \lambda x.e\colon T' \to U' \quad T <: T' \quad U' <: U}{\Gamma \vdash \lambda x.e\colon T \multimap U}$$

We use the same reasoning as in the proof of Lemma 15, ignoring $\mathsf{un}(\Gamma)$. □

Lemma 17
For all $\vec{T}$ and S, if $\vec{T}$ matches S, then $|\vec{T}| \leqslant \text{bound}(S)$.

Proof
By induction on the derivation of $\vec{T}$ matches S with a case-analysis on the last rule (equivalently on the form of $\vec{T}$). Let μ and B be as defined in Definition 4.

- $\vec{T} = \varepsilon$. This case is trivial, as $|\varepsilon| = 0$.
- $\vec{T} = v\vec{T'}$. From the derivation, $S = ?U.S'$ and $\vec{T'}$ matches S'. Because $\mu = B(\mu)$, $\mu(S) = 1 + \mu(S')$. By the induction hypothesis, $|\vec{T'}| \leqslant \mu(S')$. Therefore $|\vec{T}| \leqslant \mu(S)$. This reasoning is valid even if $\mu(S') = \infty$.

- $\vec{T} = l\vec{T}'$. From the derivation, $S = \&\langle l_i : S_i\rangle_{i\in I}$ with $l = l_j$ for some $j \in I$, and $\vec{T}'$ matches S_j. Because $\mu = B(\mu)$, $\mu(S) = 1 + max_{i\in I}\{\mu(S_i)\}$. By the induction hypothesis, $|\vec{T}'| \leqslant \mu(S_j) \leqslant max_{i\in I}\{\mu(S_i)\}$. Therefore $|\vec{T}| \leqslant \mu(S)$. Again, this reasoning is valid even if some of the $\mu(S_i)$ are ∞. □

Lemma 18
If $\Gamma \vdash e : T$ and $e \longrightarrow_{\mathsf{v}} e'$, then $\Gamma \vdash e' : T$.

Lemma 19
If $C \equiv C'$, then in the sense of Definition 6, C and C' contain exactly the same buffers: for all buffers $c \mapsto (c', k, \vec{b})$, $c \mapsto (c', k, \vec{b}) \in C$ if and only if $c \mapsto (c', k, \vec{b}) \in C'$.

Lemma 20
1. If $\vec{B}$ matches S and $S/\vec{B} = ?T.S'$ and $U <: T$, then $\vec{B}U$ matches S.
2. If $\vec{B}$ matches S and $S/\vec{B} = \&\{\ldots, l : S, \ldots\}$, then $\vec{B}l$ matches S.

Lemma 21
If $\Gamma \vdash C \triangleright \Delta$, then

1. $\forall c \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta).(\Delta(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}$ matches $\Gamma(c)$ and $\mathrm{bound}(\Gamma(c)) \leqslant n))$
2. $\forall c, d \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta).(\Delta(c) = (d, n, \vec{B})$ and $\Delta(d) = (c, n', \vec{B}') \Rightarrow \Gamma(c)/\vec{B} \asymp \Gamma(d)/\vec{B}')$

Proof
By induction on the typing derivation. In the case of T-Par, the conclusions of the lemma are among the hypotheses of the typing rule. The cases of T-Thread and T-Buffer are trivial because $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ (for T-Buffer this is implied by $c \notin \vec{b}$). In the case of T-New, the environments become smaller, which preserves the desired property. □

Lemma 22
If C is fully buffered and $C \longrightarrow C'$, then C' is fully buffered.

Proof
Inspection of the reduction rules in Figure 4 shows that buffers are created in pairs and are never destroyed. □

Definition 23
If $C \longrightarrow C'$, then let R be the rule from Figure 4 that appears earliest in the derivation sequence. We say that R is the *original* rule of the reduction or that the reduction *originates* from rule R. If the original rule is R-Send, R-Select, R-Receive or R-Branch, then the rule identifies a unique channel on which the communication takes place and a unique buffer whose contents are changed by the reduction.

Theorem 24 (*Type preservation*)
If $\Gamma \vdash C \triangleright \Delta$ and $C \longrightarrow C'$, then there exist Γ' and Δ' such that $\Gamma' \vdash C' \triangleright \Delta'$ and $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta)$. Furthermore,

1. if the reduction is a communication on channel c, then $\Gamma(c) \mapsto \Gamma'(c)$; if the reduction originates from R-Select or R-Branch, then the exact relationship between $\Gamma(c)$ and $\Gamma'(c)$ is determined by the label sent or received.
2. for every channel c, if the reduction is not a communication on c, then $\Gamma'(c) = \Gamma(c)$.
3. if the reduction originates from R-Receive or R-Branch on channel c with $c : (d, n, \vec{B}) \in \Delta$, then $c : (d, n, tail(\vec{B})) \in \Delta'$ and for all $c' \neq c$, $\Delta'(c') = \Delta(c)$. Also $\Gamma'(c)/tail(\vec{B}) = \Gamma(c)/\vec{B}$.
4. If the reduction originates from R-Send or R-Select on channel c with $c : (d, n, \vec{B}), d : (c, n', \vec{B}') \in \Delta$, then $\vec{B} = \varepsilon, c : (d, n, \vec{B}), d : (c, n', \vec{B}'') \in \Delta', \vec{B}'' = \vec{B}'T$ (where T is the type of the message) or $\vec{B}'l$, and for all $c' \neq c$, $\Delta'(c') = \Delta(c')$.

Proof
By induction on the derivation of $C \longrightarrow C'$, with a case-analysis on the last rule. First note that the detailed statements in clauses 1–4 follow easily from the information in the typing derivations considered during the proof; so we will not explicitly discuss them.

- R-Thread. We have $\langle E[e] \rangle \longrightarrow \langle E[e'] \rangle$ because $e \longrightarrow_{\mathsf{v}} e'$, and $\Gamma \vdash \langle E[e] \rangle \triangleright \emptyset$. By Lemmas 12, 13 and 18 we obtain $\Gamma \vdash \langle E[e'] \rangle \triangleright \emptyset$. Conclusions 1–4 are trivially satisfied because the reduction is not a communication and Γ is preserved.
- R-Fork. We have $\langle E[\mathsf{fork}\ e] \rangle \longrightarrow \langle e \rangle \parallel \langle E[\mathsf{unit}] \rangle$ and $\Gamma \vdash \langle E[\mathsf{fork}\ e] \rangle \triangleright \emptyset$. Let $\mathscr{D}$ be the derivation of this typing. By Lemma 12 there exist Γ_1, Γ_2 and T such that $\Gamma = \Gamma_1 + \Gamma_2$, and there is a subderivation $\mathscr{D}'$ of $\mathscr{D}$ concluding $\Gamma_1 \vdash \mathsf{fork}\ e : \mathsf{Unit}$. The end of $\mathscr{D}'$ has the form
$$\frac{\Gamma_1 \vdash e : T \quad \vdash \mathsf{fork} : T \to \mathsf{Unit} \quad \mathsf{un}(T)}{\Gamma_1 \vdash \mathsf{fork}\ e : \mathsf{Unit}}$$
By Lemma 13 we have $\Gamma_2 \vdash \langle E[\mathsf{unit}] \rangle \triangleright \emptyset$, and so we can construct the derivation
$$\frac{\Gamma_1 \vdash \langle e \rangle \triangleright \emptyset \qquad \Gamma_2 \vdash \langle E[\mathsf{unit}] \rangle \triangleright \emptyset}{\Gamma \vdash \langle e \rangle \parallel \langle E[\mathsf{unit}] \rangle \triangleright \emptyset}$$
The additional hypotheses of T-Par are trivial because the configuration contains no buffers. Conclusions 1–4 are trivially satisfied because the reduction is not a communication and Γ is preserved.
- R-Par. We have $C \parallel C'' \longrightarrow C' \parallel C''$ because $C \longrightarrow C'$. We have $\Gamma \vdash C \parallel C'' \triangleright \Delta$. The typing derivation has the form
$$\frac{\Gamma_1 \vdash C \triangleright \Delta_1 \qquad \Gamma_2 \vdash C'' \triangleright \Delta_2}{\Gamma_1 + \Gamma_2 \vdash C \parallel C'' \triangleright \Delta_1 + \Delta_2}$$
where $\Gamma_1 + \Gamma_2 = \Gamma$ and $\Delta_1 + \Delta_2 = \Delta$ and the other hypotheses are

$$\forall c \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta).(\Delta(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}\,\mathrm{matches}\,\Gamma(c) \text{ and } \mathrm{bound}(\Gamma(c)) \leqslant n)) \quad (1)$$

$$\forall c,d \in \mathrm{dom}(\Gamma)\cap\mathrm{dom}(\Delta).(\Delta(c) = (d,n,\vec{B}) \text{ and } \Delta(d) = (c,n',\vec{B'}) \Rightarrow \Gamma(c)/\vec{B} \asymp \Gamma(d)/\vec{B'}) \quad (2)$$

By induction we have $\Gamma'_1 \vdash C' \rhd \Delta'_1$, $\mathrm{dom}(\Gamma'_1) = \mathrm{dom}(\Gamma_1)$ and $\mathrm{dom}(\Delta'_1) = \mathrm{dom}(\Delta_1)$. Hence $\Gamma'_1 + \Gamma_2$ and $\Delta'_1 + \Delta_2$ are defined. We can derive $\Gamma'_1 + \Gamma_2 \vdash C' \parallel C'' \rhd \Delta'_1 + \Delta_2$ if we can establish conditions (1) and (2) for $\Gamma'_1 + \Gamma_2$ and $\Delta'_1 + \Delta_2$. By Lemma 21 we have (1) and (2) for Γ'_1 and Δ'_1 and for Γ_2 and Δ_2. Therefore the required version of (1) is true because it is a property of individual channels. For condition (2), we need to consider the case of a pair of channels $c \in \mathrm{dom}(\Gamma'_1)\cap\mathrm{dom}(\Delta'_1)$ and $d \in \mathrm{dom}(\Gamma_2)\cap\mathrm{dom}(\Delta_2)$, where the types of c and its buffer are changed by the reduction but the types of d and its buffer, by assumption, are unchanged. Let $\Delta_1(c) = (d,n,\vec{B'_c})$, $\Delta'_1(c) = (d,n',\vec{B'_c})$ and $\Delta_2(d) = (c,m,\vec{B_d})$. The only possibility is that the reduction is by R-Receive or R-Branch on c; if it is by R-Send or R-Select, then the buffer of d, which would change, must be in C, contradicting the assumption that $d \in \mathrm{dom}(\Delta_2)$. By clauses 1 and 3 of the induction hypothesis and the definitions of matching and reduction of session types, we have $\Gamma'_1(c)/\vec{B'_c} = \Gamma_1(c)/\vec{B_c} \asymp \Delta_2(d)/\vec{B_d}$.

- R-Struct. Follows from the induction hypothesis and Lemma 9.
- R-New. We have $(\nu c_1c_2)C \longrightarrow (\nu c_1c_2)C'$ because $C \longrightarrow C'$. We have $\Gamma \vdash (\nu c_1c_2)C \rhd \Delta$ with the derivation

$$\frac{\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \rhd \Delta + c_1 : (c_2,n_1,\vec{B_1}) + c_2 : (c_1,n_2,\vec{B_2})}{\Gamma \vdash (\nu c_1c_2)C \rhd \Delta}$$

By induction we have $\Gamma' + c : S'_1 + d : S'_2 \vdash C' \rhd \Delta' + c : (d,n_1,\vec{B'_1}) + d : (c,n_2,\vec{B'_2})$ with $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta)$. We can therefore derive

$$\frac{\Gamma' + c_1 : S'_1 + c_2 : S'_2 \vdash C' \rhd \Delta' + c_1 : (c_2,n_1,\vec{B'_1}) + c_2 : (c_1,n_2,\vec{B'_2})}{\Gamma' \vdash (\nu c_1c_2)C' \rhd \Delta'}$$

The remaining conditions follow directly by induction.

- R-Init. We have

$$\langle E[\mathsf{request}\ n\ x]\rangle \parallel \langle E'[\mathsf{accept}\ n'\ x]\rangle \longrightarrow (\nu cd)(c \mapsto (d,n,\varepsilon) \parallel d \mapsto (c,n',\varepsilon) \parallel \langle E[c]\rangle \parallel \langle E'[d]\rangle)$$

We have $\Gamma \vdash \langle E[\mathsf{request}\ n\ x]\rangle \parallel \langle E'[\mathsf{accept}\ n'\ x]\rangle \rhd \emptyset$ with the derivation

$$\frac{\dfrac{\Gamma_1 \vdash E[\mathsf{request}\ n\ x] : T_1}{\Gamma_1 \vdash \langle E[\mathsf{request}\ n\ x]\rangle \rhd \emptyset} \qquad \dfrac{\Gamma_2 \vdash E'[\mathsf{accept}\ n'\ x] : T_2}{\Gamma_2 \vdash \langle E'[\mathsf{accept}\ n'\ x]\rangle \rhd \emptyset}}{\Gamma_1 + \Gamma_2 \vdash \langle E[\mathsf{request}\ n\ x]\rangle \parallel \langle E'[\mathsf{accept}\ n'\ x]\rangle \rhd \emptyset}$$

where $\Gamma_1 + \Gamma_2 = \Gamma$. By Lemma 12 and the typing rule for request, there exist Γ_3 and Γ_4 such that $\Gamma_1 = \Gamma_3 + \Gamma_4$ and $\Gamma_3 \vdash \mathsf{request}\ n\ x : \overline{S}$, with $\Gamma_3 \vdash x : \langle S\rangle^{\mathsf{r}}$

and $\mathrm{bound}(\overline{S}) \leqslant n$. Similarly there exist Γ_5 and Γ_6 such that $\Gamma_2 = \Gamma_5 + \Gamma_6$ and $\Gamma_5 \vdash \mathsf{accept}\ n'\ x : S$, with $\Gamma_5 \vdash x : \langle S\rangle^{\mathsf{a}}$ and $\mathrm{bound}(S) \leqslant n'$.
Taking c and d to be fresh channels, Lemma 13 gives $\Gamma_1 + c : \overline{S} \vdash \langle E[c]\rangle \triangleright \emptyset$ and $\Gamma_2 + d : S \vdash \langle E[d]\rangle \triangleright \emptyset$. We also have $\vdash c \mapsto (d, n, \varepsilon) \triangleright c : (d, n, \varepsilon)$ and $\vdash d \mapsto (c, n', \varepsilon) \triangleright d : (c, n', \varepsilon)$ from which we use T-Par and T-New to derive

$$\Gamma_1 + \Gamma_2 \vdash (\nu cd)(c \mapsto (d, n, \varepsilon) \parallel d \mapsto (c, n', \varepsilon) \parallel \langle E[c]\rangle \parallel \langle E'[d]\rangle) \triangleright \emptyset$$

T-Par requires $\mathrm{bound}(\overline{S}) \leqslant n$ and $\mathrm{bound}(S) \leqslant n'$, which are among the data above; the other requirements reduce to $S \asymp \overline{S}$ because the buffers are empty. Conclusions 1–4 are trivially satisfied because the reduction is not a communication and Γ is preserved.

- R-Send. We have

$$c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\mathsf{send}\ v\ c]\rangle \longrightarrow$$
$$c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}v) \parallel \langle E[c]\rangle$$

and $|\vec{b}| < n$. We have $\Gamma \vdash c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\mathsf{send}\ v\ c]\rangle \triangleright \Delta$ with the derivation

$$\dfrac{(1) \quad \dfrac{\Gamma_2 \vdash \vec{b} : \vec{B} \quad |\vec{b}| \leqslant n}{\Gamma_2 \vdash d \mapsto (c, n, \vec{b}) \triangleright d : (c, n, \vec{B})} \quad \Gamma' \vdash \langle E[\mathsf{send}\ v\ c]\rangle \triangleright \emptyset}{\Gamma' + \Gamma_1 + \Gamma_2 \vdash c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\mathsf{send}\ v\ c]\rangle \triangleright c : (d, n', \vec{B'}), d : (c, n, \vec{B})}$$

where (1) is

$$\dfrac{\Gamma_1 \vdash \vec{b'} : \vec{B'} \quad |\vec{b'}| \leqslant n'}{\Gamma_1 \vdash c \mapsto (d, n', \vec{b'}) \triangleright c : (d, n', \vec{B'})}$$

and we also have $\Gamma' + \Gamma_1 + \Gamma_2 = \Gamma$, $\Delta = c : (d, n', \vec{B'}), d : (c, n, \vec{B})$, $\mathrm{bound}(\Gamma'(d)) \leqslant n$, $\mathrm{bound}(\Gamma'(c)) \leqslant n'$, $\vec{B'}$ matches $\Gamma'(c)$ and $\vec{B}$ matches $\Gamma'(d)$.
By Lemma 12 there exist Γ_3 and Γ_4 such that we have the subderivation

$$\dfrac{\Gamma_5 \vdash v : T \qquad \Gamma_6 + c : {!T.S} \vdash c : {!T.S}}{\Gamma_3 \vdash \mathsf{send}\ v\ c : S}$$

with $\Gamma' = \Gamma_3 + \Gamma_4$ and $\Gamma_3 = \Gamma_5 + \Gamma_6 + c : {!T.S}$. Lemma 13 gives $\Gamma_4 + \Gamma_6 + c : S \vdash \langle E[c]\rangle \triangleright \Delta$.
By Lemma 21 we have $\Gamma'(c)/\vec{B'} \asymp \Gamma'(d)/\vec{B}$. Also, $\Gamma'(c) = {!T.S}$. Because $\vec{B'}$ matches $!T.S$ we have $\vec{B'} = \varepsilon$ and hence $!T.S/\vec{B'} = {!T.S}$. Therefore $?T.\overline{S} <: \Gamma'(d)/\vec{B}$; so $\Gamma'(d)/\vec{B} = {?T'.S'}$ with $\overline{S} <: S'$ and $T <: T'$. Lemma 20 gives $\vec{B}T$ matches $\Gamma'(d)$, and we can build the following derivation:

$$\dfrac{(1) \quad \dfrac{\Gamma_2 + \Gamma_5 \vdash \vec{b}v : \vec{B}T \quad |\vec{b}v| \leqslant n}{\Gamma_2 + \Gamma_5 \vdash d \mapsto (c, n, \vec{b}v) \triangleright d : (c, n, \vec{B}T)} \quad \Gamma_4 + \Gamma_6 + c : S \vdash \langle E[c]\rangle \triangleright \emptyset}{\Gamma_1 + \Gamma_2 + \Gamma_4 + \Gamma_5 + \Gamma_6 + c : S \vdash c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}v) \parallel \langle E[c]\rangle \triangleright c : (d, n', \vec{B'}), d : (c, n, \vec{B}T)}$$

Most of the additional hypotheses of T-Par follow from the previous reasoning and the original derivation. We also need $S/\vec{B}' \asymp \Gamma'(d)/\vec{B}T$. For this, we have $S/\vec{B}' = S/\varepsilon = S$ and $\Gamma'(d)/\vec{B}T = S'$. By definition, $S \asymp S'$ because $\overline{S} <: S'$. Finally, we also need $\mathrm{bound}(S) \leqslant n$, which follows from $\mathrm{bound}(\Gamma'(c)) \leqslant n$ and Lemma 5.

- R-Select. Similar to the previous case.
- R-Receive. We have

$$c \mapsto (d, n, v\vec{b}) \parallel \langle E[\mathsf{receive}\ c]\rangle \longrightarrow c \mapsto (d, n, \vec{b}) \parallel \langle E[(v, c)]\rangle$$

and $\Gamma \vdash c \mapsto (d, n, v\vec{b}) \parallel \langle E[\mathsf{receive}\ c]\rangle \triangleright \Delta$ with the derivation

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v : T \quad \Gamma_2 \vdash \vec{b} : \vec{B}}{\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B}}}{\Gamma_1 + \Gamma_2 \vdash c \mapsto (d, n, v\vec{b}) \triangleright c : (d, n, T\vec{B})} \quad \Gamma' \vdash \langle E[\mathsf{receive}\ c]\rangle \triangleright \emptyset}{\Gamma \vdash c \mapsto (d, n, v\vec{b}) \parallel \langle E[\mathsf{receive}\ c]\rangle \triangleright \Delta}$$

where $\Gamma' + \Gamma_1 + \Gamma_2 = \Gamma$ and $\Delta = c : (d, n, T\vec{B})$ and $T\vec{B}\,\mathrm{matches}\,\Gamma'(c)$ and $\mathrm{bound}(\Gamma'(c)) \leqslant n$. We therefore have $\Gamma'(c) = ?U.S$ with $T <: U$ and $\vec{B}\,\mathrm{matches}\,S$.

By Lemma 12 there exist Γ_3 and Γ_4 such that we have the subderivation

$$\dfrac{\Gamma_3 + c : ?U.S \vdash c : ?U.S}{\Gamma_3 + c : ?U.S \vdash \mathsf{receive}\ c : U \otimes S}$$

with $\Gamma' = \Gamma_3 + \Gamma_4$. Lemma 13 gives $\Gamma_1 + \Gamma_3 + \Gamma_4 + c : S \vdash \langle E[(v, c)]\rangle \triangleright \emptyset$, and we can build the derivation

$$\dfrac{\dfrac{\Gamma_2 \vdash \vec{b} : \vec{B}}{\Gamma_2 \vdash c \mapsto (d, n, \vec{b}) \triangleright c : (d, n, \vec{B})} \quad \Gamma_1 + \Gamma_3 + \Gamma_4 + c : S \vdash \langle E[(v, c)]\rangle \triangleright \emptyset}{\Gamma_1 + \Gamma_2 + \Gamma_3 + \Gamma_4 + c : S \vdash c \mapsto (d, n, \vec{b}) \parallel \langle E[(v, c)]\rangle \triangleright c : (d, n, \vec{B})}$$

in which T-Par uses $\vec{B}\,\mathrm{matches}\,S$. Because there is only one buffer, T-Par has no compatibility condition, but we do need $\mathrm{bound}(S) \leqslant n$, which follows from $\mathrm{bound}(\Gamma'(c)) \leqslant n$ and Lemma 5.

- R-Branch. Similar to the previous case. $\square$

Theorem 25 (*Runtime Safety*)

Let $\Gamma \vdash C \triangleright \Delta$ be fully buffered and assume that $C \longrightarrow^* C'$. If C'' is a blocked thread in C', then one of the following applies:

1. C'' is $\langle v\rangle$ or $\langle \mathsf{send}\ v\rangle$ or $\langle \mathsf{request}\ n\ x\rangle$ or $\langle \mathsf{accept}\ n\ x\rangle$;
2. C'' is $\langle E[\mathsf{receive}\ c]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$;
3. C'' is $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i : e_i\}_{i \in I}]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$.

Proof
By Theorem 24 and Lemma 22, we know that $\Gamma' \vdash C' \triangleright \Delta'$ and C' is fully buffered. Suppose that C'' is a blocked thread of none of the above forms. Analysing the reduction rules in Figures 3 and 4, we find six cases to consider:

1. C'' is $\langle E[\mathsf{receive}\ c]\rangle$ and $c \mapsto (\_, \_, l\vec{b}) \in C'$;
2. C'' is $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i : e_i\}_{i \in I}]\rangle$ and $c \mapsto (\_, \_, v\vec{b}) \in C'$;
3. C'' is $\langle E[\mathsf{send}\ v\ c]\rangle$, and $c \mapsto (d, \_, \_), d \mapsto (\_, n, \vec{b}) \in C'$, and $|\vec{b}| \geqslant n$;
4. C'' is $\langle E[\mathsf{select}\ l\ c]\rangle$, and $c \mapsto (d, \_, \_), d \mapsto (\_, n, \vec{b}) \in C'$, and $|\vec{b}| \geqslant n$;
5. C'' is $\langle E[\mathsf{let}\ x, y = v\ \mathsf{in}\ e]\rangle$ and v is not of the form (v_1, v_2);
6. C'' is $\langle E[\mathsf{fix}\ v]\rangle$ and v is not of the form $\lambda x.e$.

We outline the argument for each case.

1. Build the typing derivation for $\Gamma' \vdash C' \triangleright \Delta'$. We know that the typing environment that types C'' contains an entry $c : ?T.S$. The derivation includes an application of rule T-Par which combines the buffer $c \mapsto (\_, \_, l\vec{b})$ and the rest of the configuration. We therefore have $l\vec{b}\,\mathrm{matches}\,\Gamma'(c)$, from which we conclude that $\Gamma'(c)$ is of the form $\&\langle ..l : S..\rangle$; hence contradiction.
2. Similar to the previous case.
3. The main point is to show that the assumption $|\vec{b}| \geqslant n$ leads to a contradiction. Consider the information in case R-Send of the proof of type preservation. We have $\mathrm{bound}(\Gamma'(d)) \leqslant n$. From $\vec{B}T\,\mathrm{matches}\,\Gamma'(d)$ and Lemma 17 we have $\mathrm{bound}(\Gamma'(d)) \geqslant |\vec{B}T| = |\vec{b}| + 1$; hence $|\vec{b}| < n$.
4. Similar to the previous case.
5. The typing derivation shows that the type of v must be of the form $T_1 \otimes T_2$, and hence v must be of the form (v_1, v_2).
6. Similar to the previous case. □

To apply the runtime safety theorem we consider the initial state of a system to be a parallel collection of threads, with no buffers, so that it is trivially fully buffered.

Finally, we observe that the expression '$\mathsf{accept}\ n\ a$' can safely be replaced by '$\mathsf{accept}\ \mathrm{bound}(S)\ a$' where $a : \langle S\rangle^{\mathsf{a}}$ in the current environment, and similarly for $\mathsf{request}$. In other words, the compiler can infer the necessary buffer sizes. Also, when a channel of type S is used, for example by send, its subsequent type is S' with $S \mapsto S'$; Lemmas 5 and 17, and rule T-Buffer, imply that information available during typechecking can be used to generate code to reduce the size of a buffer and ultimately to deallocate the buffer of a channel of type end.

7 Related and future work

Apart from our own previous work (Vasconcelos *et al.* 2004, 2006), the main formal studies of session types in mainstream language paradigms are by Dezani-Ciancaglini *et al.* (2005, 2006), Coppo *et al.* (2007), Capecchi *et al.* (2009) and ourselves (Gay *et al.* 2010), all for object-oriented languages. The languages studied by Dezani-Ciancaglini *et al.* (2006) and Coppo *et al.* (2007) have an interesting progress property, whereby well-typed programs do not starve at communication points, once a session is established; however, in contrast to our language, they do not allow a single thread to interleave communications on different channels.

As mentioned in the introduction, work on session types for functional languages started with our own work (Gay *et al.* 2003; Vasconcelos *et al.* 2004, 2006). Neubauer and Thiemann (2004a) showed how to implement session types on top of the Haskell programming language; furthermore, Neubauer and Thiemann (2004b) modelled software components as concurrent functional processes and used session types to extract the smallest protocol required by each process; in addition, they addressed the problem of program transformation, from sequential to multi-tier, guided by session types (Neubauer and Thiemann 2005).

Asynchronous semantics for session types can be traced back to the unpublished work of Neubauer and Thiemann (2004c). Fähndrich *et al.* (2006) chose an asynchronous semantics for Sing# but without formal semantics. The present formulation is based on our previous work (Gay & Vasconcelos 2007). Some recent work uses asynchronous semantics, including the work of Coppo *et al.* (2007) and Capecchi *et al.* (2009) in the context of OO languages and that of Honda *et al.* (2008) in the context of a π-calculus like language with multiparty session types.

Yoshida and Vasconcelos (2007) showed that to model 'true' channel passing, where one thread may acquire both ends of a communication channel, the two endpoints of the channel must be treated separately. Like Gay and Hole (2005), they referred to the endpoints of channel c as c^+ and c^-. The present paper achieves true channel passing by storing the peer endpoint c' of c in c's buffer and using the double binder $(\nu cc')$ to link an endpoint with its peer. Recent work by Honda *et al.* (2008), although using asynchronous semantics and generalising session types to multi-party protocols, does not allow a thread to acquire both endpoints of a channel.

Cyclone (Grossman *et al.* 2002; Grossman 2003), *Vault* (DeLine & Fähndrich 2001) and *adoption and focus* (Fähndrich & DeLine 2002) are systems based on the C programming language that allow protocols to be statically enforced by a compiler. They share our goal but vary greatly in the techniques used. *Cyclone* (Grossman *et al.* 2002) adds many benefits to C, but its support for protocols is limited to enforcing locking of resources. Between acquiring and releasing a lock, there are no restrictions on how a thread may use a resource. In contrast, our system uses types both to enforce locking of channels (via linearity) and to enforce protocols on channels. In the *Vault* system (DeLine & Fähndrich 2001) and its extension 'adoption and focus' (Fähndrich & DeLine 2002) annotations are added to C programs, in order to describe protocols that a compiler can statically enforce. Objects on which protocols may be specified are not limited to communication channels. However, in the case of communication channels, session types allow more detailed specification of protocols. Also, being based on C, these systems do not support higher-order functional programming.

In terms of session types in functional languages, the main area of future work is to study type inference and polymorphism, either in a simple ML style or along the lines proposed by Gay (2008). We should also investigate the relationship with other forms of static analysis, including type and effect systems (Amtoft *et al.* 1999).

We have mentioned that a thorough understanding of session types in functional languages provides a good foundation for the development of object-oriented session types. In recent work (Gay *et al.* 2010) we have been developing a more general theory of object-oriented session types than exists at present, including inheritance and subtyping and integrating with more general notions of non-uniform objects. The main idea, distinct from the work of Dezani-Ciancaglini *et al.* (2005, 2006), Coppo *et al.* (2007) and Capecchi *et al.* (2009), is to use session types to control the sequence of method calls on objects and to check that the session type of an object is consistent with the session types of the objects that it uses. In this setting, a session-typed communication channel is just a particular kind of object, and communication operations are method calls. Although not directly based on the functional language of the present paper, that work continues the theme of controlling linear entities, including session-typed channels, in a simple and uniform way.

We would like to investigate whether communication on session-typed channels can be formulated in terms of monads (Peyton Jones & Wadler 1993), along the lines of input–output effects in Haskell. Ideally, for example, the son from Section 2

```
son sonAccess book =
  let s = accept sonAccess in
  let (f,s) = receive s in
  let s = send (f book) s in s
```

would be written in a form of **do**-notation

```
son sonAccess book =
  do s ← accept sonAccess
     f ← receive s
     return (send (f book) s)
```

in order to hide the re-binding of s. Such a translation could be defined easily enough as syntactic sugar, but it is not an instance of the standard translation of **do**-notation. Indeed, the standard translation does not respect linearity of the resource that is threaded through the sequence of calls. Neubauer and Thiemann (2004a) used a monad in their Haskell implementation of session types. Because their setting is somewhat different, with a continuation-passing style and restriction to a single channel, we have not yet understood whether it can be adapted to our language.

Our own previous work (Vasconcelos *et al.* 2006) avoided the re-binding by using two-sided typing judgements such as

$$x : \,!T.S \triangleright \mathsf{send}\ v\ x : \mathsf{Unit} \triangleleft x : S$$

but the type system was rather complex, partly because of the inclusion of a form of alias type. The aim of the present paper has been to simplify the underlying type theory by using more standard techniques and eliminating aliasing completely. A more detailed comparison of the two approaches, as well as a comparison of both with monadic approaches, is a subject for future work.

Acknowledgments

Vasco T. Vasconcelos 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 LASIGE). Simon Gay was partially supported by Instituto de Telecomunicações, Portugal, and by the EPSRC grant 'Engineering Foundations of Web Services: Theories and Tool Support' (EP/E065708/1). We thank the anonymous referees for many helpful comments.

References

Amtoft, T., Nielson, F. & Nielson, H. R. (1999) *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, London.

Bonelli, E., Compagnoni, A. & Gunter, E. (2005) Correspondence assertions for process synchronization in concurrent communication, *J. Funct. Program.* **15**(2), 219–247.

Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S. & Giachino, E. (2009) Amalgamating sessions and methods in object-oriented languages with generics, *Theoret. Comp. Sci.* **410**, 142–167.

Coppo, M., Dezani-Ciancaglini, M. & Yoshida, N. (2007) Asynchronous session types and progress for object-oriented languages. In *Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, M. M. Bonsangue and E. B. Johnsen (eds.), Lecture Notes in Computer Science, vol. 4468. Springer, Berlin, pp. 1–31.

DeLine, R. & Fähndrich, M. (2001) Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, vol. 36, no. 5. ACM Press, pp. 59–69.

Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N. & Drossopoulou, S. (2006) Session types for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Languages (ECOOP)*, D. Thomas (ed.), Lecture Notes in Computer Science, vol. 4067. Springer, Berlin, pp. 328–352.

Dezani-Ciancaglini, M., Yoshida, N., Ahern, A. & Drossopolou, S. (2005) A distributed object-oriented language with session types. In *Proceedings of the International Symposium on Trustworthy Global Computing (TGC)*, R. DeNicola and D. Sangiorgi (eds.), Lecture Notes in Computer Science, vol. 3705. Springer, Berlin, pp. 299–318.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R. & Levi, S. (2006) Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys Conference*, SIGOPS Operating Systems Review, vol. 40, no. 4. ACM Press, pp. 177–190.

Fähndrich, M. & DeLine, R. (2002) Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, vol. 37, no. 5. ACM Press, pp. 13–24.

Gay, S. J. (2006) *Subtyping Between Standard and Linear Function Types*, Technical Report 2006-305. Glasgow, Scotland: Department of Computing Science, University of Glasgow.

Gay, S. J. (2008) Bounded polymorphism in session types, *Math. Struct. Comp. Sci.* **18**(5), 895–930.

Gay, S. J. & Hole, M. J. (2005) Subtyping for session types in the pi calculus, *Acta Inform.* **42**(2/3), 191–225.

Gay, S. J. & Vasconcelos, V. T. (2007) *Asynchronous Functional Session Types*, Technical Report 2007-251. Glasgow, Scotland: Department of Computing Science, University of Glasgow.

Gay, S. J., Vasconcelos, V. T. & Ravara, A. (2003) *Session Types for Inter-Process Communication*, Technical Report 2003-133. Glasgow, Scotland: Department of Computing Science, University of Glasgow.

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. & Caldeira, A. Z. (2010) Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM Press.

Grossman, D. (2003) Type-safe multithreading in Cyclone. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, SIGPLAN Notices, vol. 38, no. 3. ACM Press, pp. 13–25.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y. & Cheney, J. (2002) Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, vol. 37, no. 5. ACM Press, pp. 282–293.

Honda, K. (1993) Types for dyadic interaction. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, E. Best (ed.), Lecture Notes in Computer Science, vol. 715. Springer, Berlin, pp. 509–523.

Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*, C. Honkin (ed.), Lecture Notes in Computer Science, vol. 1381. Springer, Berlin, pp. 122–138.

Honda, K., Yoshida, N. & Carbone, M. (2008) Multiparty asynchronous session types. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, SIGPLAN Notices, vol. 43, no. 1. ACM Press, pp. 273–284.

Lanese, I., Vasconcelos, V. T., Martins, F. & Ravara, A. (2007) Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods.* IEEE, pp. 305–314.

Neubauer, M. & Thiemann, P. (2004a) An implementation of session types. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, B. Jayaraman (ed.), Lecture Notes in Computer Science, vol. 3057. Springer, Berlin, pp. 56–70.

Neubauer, M. & Thiemann, P. (2004b) Protocol specialization. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, W.-N. Chin (ed.), Lecture Notes in Computer Science, vol. 3302. Springer, Berlin, pp. 246–261.

Neubauer, M. & Thiemann, P. (2004c) *Session Types for Asynchronous Communication.* [online] Available at: http://www.informatik.uni-freiburg.de/∼thiemann/papers/stac.ps.gz Accessed 17 November 2009.

Neubauer, M. & Thiemann, P. (2005) From sequential programs to multi-tier applications by program transformation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, SIGPLAN Notices, vol. 40, no. 1. ACM Press, pp. 221–232.

Peyton Jones, S. & Wadler, P. (1993). Imperative functional programming. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM Press, pp. 71–84.

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

Takeuchi, K., Honda, K. & Kubo, M. (1994) An interaction-based language and its typing system. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, C. Halatsis, D. Maritsas, G. Philokyprou and S. Theodoridis (eds.), Lecture Notes in Computer Science, vol. 817. Springer, Berlin, pp. 398–413.

Vallecillo, A., Vasconcelos, V. T. & Ravara, A. (2006) Typing the behavior of software components using session types, *Fundam. Inform.* **73**(4), 583–598.

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

Vasconcelos, V. T., Ravara, A. & Gay, S. J. (2004) Session types for functional multithreading. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, P. Gardner and N. Yoshida (eds.), Lecture Notes in Computer Science, vol. 3170. Springer, Berlin, pp. 497–511.

W3C. (2005) Services choreography description language version 1.0 [online]. Available at: `http://www.w3.org/TR/2005/CR-ws-cdl-10/` Accessed 17 November 2009.

Walker, D. (2005) Substructural type systems. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed.). MIT Press, Cambridge, Massachusetts, Chapter 1, pp. 3–43.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inform. Comput.* **115**(1), 38–94.

Yoshida, N. & Vasconcelos, V. T. (2007) Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication, *ENTCS* **171**(4), 73–93.