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 type checking. Applications include network protocols, business processes, and operating system services. In this paper we define a multithreaded functional language with session types, which unifies, simplifies and extends previous work. There are four main contributions. First: 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 re-use 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.

This 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 ob-

serve the correct protocols. For example, the type $\mathsf{S} = \&\langle \mathsf{service}\colon\ ?\mathbf{Int}.!\,\mathbf{Int}.\mathsf{S},\ \ \mathsf{quit}:\mathsf{End}\rangle$ 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, 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 & 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); Capecchi *et al.* (2008) 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 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 formalize an operational semantics in which communication is buffered, instead of assuming synchronization 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, unpublished work by Neubauer & Thiemann (2004c). Fähndrich *et al.* (2006) also use buffered communication but have not published a formal semantics.
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, receive$\colon\ ?T.S \to T \otimes S$

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 5 defines the typing system 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 (Dezani-Ciancaglini *et al.*, 2006; W3C, 2005). A mother and her young son are using an online book shop. The shop implements a simple protocol described by the session type

$$\mathsf{Shop} = \&\langle \mathsf{add} : ?\,\mathsf{Book}\,.\,\mathsf{Shop}\,,\ \ \mathsf{checkout} : ?\,\mathsf{Card}\,.\,?\,\mathsf{Address}\,.\,\mathbf{end}\rangle$$

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 a delivery address. We write all this as:

$$\mathsf{Shopper} = \oplus\langle \mathsf{add} : !\,\mathsf{Book}\,.\,\mathsf{Shopper}\,,\ \ \mathsf{checkout} : !\,\mathsf{Card}\,.\,!\,\mathsf{Address}\,.\,\mathbf{end}\rangle$$

Notice that protocols for shops and shoppers are *compatible*, in the sense an interaction between the two will not terminate prematurely due to a mismatch in the expectations of one of the partners. In fact, 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 $\oplus$ and &; we have that duality that 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) chose 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 the possession of the accept capability, the shop contains an expression **accept** shopAccess; whereas the shopper exercises its 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 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.

The mother intends to choose a book for herself, then allow her son to choose a book. She does not want to give him free access to the channel which accesses the shop, so instead she gives him a function which allows him to choose exactly one book (of an appropriate kind) and then completes the transaction. This function, 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

```
Recipient = !(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 → ⟨Recipient⟩^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

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 $\langle \mathsf{Shop} \rangle^{\mathsf{a}}$, and that of mother with type $\langle \mathsf{Shopper} \rangle^{\mathsf{r}}$) are reconciled by giving shopAccess the type $\langle \mathsf{Shop}, \mathsf{Shopper} \rangle$, which turns out to be a super type 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 etc.

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

The example illustrates the following general points about our language, its semantics and its type system; the details are presented in Sections 3 to 5.

- Channels, such as c in mother, are *linear* values; session types are linear types. The linear function type constructor $\multimap$ appears in the type of voucher because applying voucher to a channel of type Shop yields a function closure which contains a channel—hence this function closure must itself be treated as a linear value and given a linear type. Because of linearity, Son cannot duplicate the voucher and order more than one book.
- Operations on channels, such as **send** and **select**, return the channel after communicating on it. Our programming style is to repeatedly re-bind the channel name using **let**; each c is of course a fresh bound variable. The **receive** operation returns the value received and the channel, as a (linear) pair which is split by a **let** construct. In the static type system, the channel type returned by, for example, **send**, is not the same as the channel type given to it; this reflects the fact that part of the session types is consumed by a communication operation.
- The type system supports programming with higher-order functions on channels in a very natural way, as illustrated by the function voucher in the example.

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.

A few variations of the example illustrating subtyping and (explicit) channel sending complete this section.

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 type inference 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, for the channel s returned by son is not of type **end** anymore. Since the new version of the voucher does not "proceed" to checkout, the type of channel s returned by the son is still Shopper. 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 card address c) s in
  let (sonBook, s) = receive s in
  ...
  let s = select checkout s in
  let s = send card s in
  let s = send address s in s
```

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 NewShop is still *compatible* with the old Shooper; 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. Stingy mother changed her mind. After starting a session with shop, she decides her budget has no room for more books. In order to comply 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:

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

and her code as below.

```
mother :: Card → Address → ⟨StingyShopper⟩^r → end
stingyMother 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
```

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

Can StingyShopper safely interact with NewShop? Reasoning as for the case of StingyMother 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 StingyShopper, hence NewShop and StingyShopper are compatible. Section 4 discusses the relationship between subtyping, duality and compatibility.

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

```
shopAccess:⟨NewShop,StingyShopper⟩, sonAccess:⟨Son,Recipient⟩ ⊢
  ⟨newShop shopAccess⟩ ‖ ⟨son sonAccess sBook⟩ ‖
  ⟨stingyMother 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-visuals, and electronic gadgets. The new protocol for the shop reflects the change,

```
ProductShop = &⟨add: ?Product.NewShop,
                remove: ?Product.NewShop,
```

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

Fig. 1. Syntax.

$$\mathsf{checkout}\colon\ ?\mathsf{Card}.?\mathsf{Address}.\mathbf{end}\rangle$$

where the shop as made sure that old Book is a subtype of new Product. Once again, the new shop is still compatible with the old shopper, that 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

$$\mathsf{LunchersShopper} = \oplus\langle \mathsf{checkout}:!\mathsf{Lunchers}.!\mathsf{Address}.\mathbf{end}\rangle$$

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 the order. When shop has received the customer's credit card details, it just passes the channel to shipper. When the customer sends her address, it goes directly to shipper. The session type used for communication between shop and shipper is as follows; note the occurrence of the session type ?Address.**end** as the type of the message.

$$\mathsf{Shipper} = ?(?\mathsf{Address}.\mathbf{end}).\mathbf{end}$$

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

3 Syntax and Operational Semantics

Most of the syntax of our language was described in the previous section. We rely on a countable set of *term variables* x, and on a disjoint countable set of (runtime) *channel endpoints* c, and use α to range over both kinds of *identifiers*. We also rely on a set of *labels* l, 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.

$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 \qquad \text{(E-COMM,E-ASSOC)}$$
$$C_1 \parallel (\nu cd)C_2 \equiv (\nu cd)(C_1 \parallel C_2) \text{ if } c,d \notin fc(C_1) \qquad \text{(E-SCOPE)}$$

Fig. 2. Structural congruence.

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

Fig. 3. Reduction of expressions.

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. We write $fc(C)$ for the *free channels* of a configuration C. Structural equivalence, the smallest relation satisfying the rules in Figure 2, 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. 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*. Items in $\vec{b}$ are values v (written and read by send and receive expressions) and labels l (written and read by select and case expressions).

Rule R-INIT synchronizes 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. Two new endpoints are created, c and d, one for each thread. Also, two new buffers are created, 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 $\{l_i\colon e_i\}_{i\in I}$ is the application of the function e_j,

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

$$\frac{C \longrightarrow C'}{C \parallel C'' \longrightarrow C' \parallel C''} \qquad \frac{C \longrightarrow C'}{(\nu cd)C \longrightarrow (\nu cd)C'} \qquad \frac{C \equiv C' \quad C' \longrightarrow C'' \quad C'' \equiv C'''}{C \longrightarrow C'''} \qquad \text{(R-PAR,R-NEW,R-STRUCT)}$$

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

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

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

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

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

Fig. 4. Reduction of configurations.

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

Fig. 5. Syntax of types and session types.

the body of the branch labelled by l_j, to channel c. 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. end is the type of a channel which cannot be used for further communication. $?T.S$ is the type of a channel from which a message of type T can be received; subsequently the channel is described by type S. Dually, $!T.S$ is the type of a channel on which a message of type T can be sent; subsequently the type of the channel is S. $\&\langle l_i\colon S_i\rangle_{i\in I}$ is the type of a channel from which a message can be received, which will be one of the labels l_i. The subsequent behaviour of the channel is described by the corresponding type S_i. Dually, $\oplus\langle l_i\colon S_i\rangle_{i\in I}$ is the type of a channel on which one of the labels l_i can be sent, with subsequent behaviour described by S_i.

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 by 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 $T \to U$ 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 or not the parameter must be used exactly once depends on whether or not 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 and access point which 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 $\mathsf{request}\ n\ a$ and $\mathsf{accept}\ n'\ a$ 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'. Data types such as Int and Bool, or compound data types such as non-linear pairs, or general recursive types, can easily be added.

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 $T \to U <: T \multimap U$ 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; $\&\langle l_i \colon S_i \rangle_{i \in I}$ is covariant in I; $\oplus\langle l_i \colon S_i \rangle_{i \in I}$ is contravariant in I, while 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.

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

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

Fig. 6. The dual function on session types.

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 $\mu X.ST$ 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 sub-typing.

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 21.3.6–7 of (Pierce, 2002). Reflexivity and transitivity of subtyping on session types are proved explicitly in (Gay & Hole, 2005), and transitivity of a similar (equivalence) relation on session types is proved explicitly in (Yoshida & 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 & Hole (2005); 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, e.g., $\overline{\mu X.\&\langle l\colon X\rangle} = \oplus\langle l\colon \mu Y.\oplus\langle l\colon Y\rangle\rangle$.

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, are from Vallecillo *et al.* (2006).[1]

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

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

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

Fig. 7. The relation $\mapsto$ on session types.

Describing protocols, session types "advance" during computation. The reduction relation on session types, Figure 7, makes this notion precise. The bound of a session type S describes the minimum size of the buffer required to hold the values received on a channel of type S. Notice that S and $\overline{S}$ will have in general different bounds.

Definition 4 (*Bound of a session type*)
The set of maps $\mathscr{S} \to \mathbb{N}^\infty$ is a complete lattice if we define $f \sqsubseteq g$ to mean $f(S) \leqslant g(S)$, $\forall S \in \mathscr{S}$, 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 $F \in (\mathscr{S} \to \mathbb{N}^\infty) \to \mathscr{S} \to \mathbb{N}^\infty$ as follows.

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

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

The definition yields an algorithm for calculating $\text{bound}(S)$. Construct a directed graph with $\{S' | S \mapsto^* S'\}$ as the vertices and $\mapsto$ as the edge relation. Label every end, $!T.S$ and $\oplus\langle l_i\colon S_i\rangle_{i\in I}$ node with 0. Iterate the following steps until a fixed point is reached: label node $?T.S$ with $n+1$ if S is labelled with n, and label node $\&\langle l_i\colon S_i\rangle_{i\in I}$ with $\max\{n_i\}_{i\in I}$ if every S_i is labelled with n_i. Finally label any unlabelled nodes with ∞. $\text{bound}(S)$ 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 23).

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

Proof
Let f be the function defined in Definition 4. $\text{bound}(S) = max\{f(T) | S \mapsto^* T\}$. $\text{bound}(S') = max\{f(T) | S' \mapsto^* T\}$. Because $S \mapsto S'$, $\{f(T) | S' \mapsto^* T\} \subseteq \{f(T) | S \mapsto^* T\}$. The result follows. □

5 Typing

This section introduces a static type system for our language.

[2] Which turn out to coincide, but we do not need this fact.

$$\begin{array}{ccccccc} & & \mathsf{lin}(S \neq \mathsf{end}) & & \mathsf{lin}(T \otimes T) & & \mathsf{lin}(T \multimap T) \\ \mathsf{un}(\mathsf{end}) & & \mathsf{un}(T \to T) & & \mathsf{un}(\langle S, S \rangle) & \mathsf{un}(\langle S \rangle^{\mathsf{a}}) & \mathsf{un}(\langle S \rangle^{\mathsf{r}}) \end{array}$$

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

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

Fig. 9. Type schemas for constants k.

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 $\text{dom}(\Gamma)$ for the set of variables and channels in Γ and $\text{cdom}(\Gamma)$ for the set of channels in Γ, and say that $\mathsf{un}(\Gamma)$ 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 types. The schemas for send and receive capture the essence of the way in which we use linear type constructors to control the use of channels. We treat send as a curried function which is given a value and a channel and returns the same channel with the type that remains after sending the specified value. There are two versions of this schema, because the partial application send v contains v in its closure and therefore we must use a linear function type if v has a linear type. 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.

Most of the rules in Figure 10 are standard. 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-FORK describes spawning a new thread, whose type is required to be unlimited in order

$$\frac{\mathsf{un}(\Gamma)\quad k\colon T}{\Gamma\vdash k\colon T}\qquad \frac{\mathsf{un}(\Gamma)}{\Gamma,\alpha\colon T\vdash\alpha\colon T}\qquad \frac{\Gamma\vdash e\colon T\quad T<\colon U}{\Gamma\vdash e\colon U} \qquad \text{(T-Const,T-Id,T-Sub)}$$

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

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

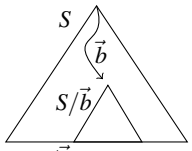
$$\frac{\Gamma_1\vdash e_1\colon T\multimap U\quad \Gamma_2\vdash e_2\colon T}{\Gamma_1+\Gamma_2\vdash e_1e_2\colon U}\qquad \frac{\Gamma_1\vdash e_1\colon T\quad \Gamma_2\vdash e_2\colon U\quad \mathsf{un}(T)}{\Gamma_1+\Gamma_2\vdash \mathsf{fork}\ e_1\ e_2\colon U} \qquad \text{(T-App,T-Fork)}$$

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

Fig. 10. Typing rules for expressions.

$$\frac{\mathsf{un}(\Gamma)}{\Gamma\vdash\varepsilon\,\mathsf{matches}\,S}\qquad \frac{\Gamma_1\vdash v\colon T\quad \Gamma_2\vdash\vec{b}\,\mathsf{matches}\,S}{\Gamma_1+\Gamma_2\vdash v\vec{b}\,\mathsf{matches}\,?T.S}\qquad \frac{\Gamma\vdash\vec{b}\,\mathsf{matches}\,S}{\Gamma\vdash l\vec{b}\,\mathsf{matches}\,\&\langle..l\colon S..\rangle}$$

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



If $\Gamma\vdash\vec{b}\,\mathsf{matches}\,S$ 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.

to ensure that the thread completely consumes any channels that it uses. T-Select is like the typing schema for send, but 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. $\Gamma\vdash\vec{b}\,\mathsf{matches}\,S$ means that the sequence of values $\vec{b}$ (which are typed by Γ) is suitable to be received by 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 is contains the buffer entries in C. More precisely, Δ is a map from channels into *buffer contents* $(d,n,\vec{b},S)$, endowed with a partial operation $+$ of disjoint union.

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 single buffer, by checking that the buffer contents $\vec{b}$ matches a given session type S, and by copying the buffer (together with S) to Δ. T-Par combines configurations in parallel, by combining the environments and the buffers in each configuration. Finally, rule T-New checks that the buffer allocated for each channel c_i is large enough to accommodate the values to be received by the channel, and that the session types associated to each channel, after the buffers have been emptied, are dual.

$$\frac{\Gamma \vdash e\colon T \quad \mathsf{un}(T)}{\Gamma \vdash \langle e\rangle \triangleright \emptyset} \qquad \frac{\Gamma \vdash \vec{b}\,\mathrm{matches}\,S}{\Gamma \vdash c \mapsto (d,n,\vec{b}) \triangleright c\colon (d,n,\vec{b},S)} \qquad \frac{\Gamma_1 \vdash C_1 \triangleright \Delta_1 \quad \Gamma_2 \vdash C_2 \triangleright \Delta_2}{\Gamma_1+\Gamma_2 \vdash C_1 \parallel C_2 \triangleright \Delta_1+\Delta_2}$$

(T-THREAD,T-BUFFER,T-PAR)

$$\frac{\begin{array}{c}\mathrm{bound}(S_i) \leqslant n_i \qquad S_1/\vec{b}_1 \asymp S_2/\vec{b}_2 \\ \Gamma + c_1\colon S_1 + c_2\colon S_2 \vdash C \triangleright \Delta + c_1\colon (c_2,n_1,\vec{b}_1,S_1) + c_2\colon (c_1,n_2,\vec{b}_2,S_2)\end{array}}{\Gamma \vdash (\nu c_1 c_2)C \triangleright \Delta} \qquad \text{(T-NEW)}$$

Fig. 12. Typing rules for configurations.

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 $\nexists c_1 \mapsto (c_2,n_1,\vec{b}_1), c_2 \mapsto (c_1,n_2,\vec{b}_2) \in C$ such that $C' \parallel c_1 \mapsto (c_2,n_1,\vec{b}_1) \parallel c_2 \mapsto (c_1,n_2,\vec{b}_2) \longrightarrow$.

By analyzing 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.

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 due to 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). As a last example consider the typable configuration C of the form $\langle \mathsf{send}\ x\ c\rangle \parallel c \mapsto (d,\_,\_) \parallel d \mapsto (\_,n,\vec{b})$, with $|\vec{b}| = n$. C cannot progress due to lack of buffer space. Notice however that $(\nu cd)C$ is not typable, for the T-NEW rule (Figure 12) makes sure there is enough space in buffers. We are only interested in configurations that are *well-buffered*.

Definition 8 (*Well-buffered configuration*)
A typable configuration $\Gamma \vdash C \triangleright \Delta$ is *well-buffered* if whenever $c_1 \mapsto (c_2,n_1,\vec{b}_1)$ and $c_2 \mapsto (c_1,n_2,\vec{b}_2)$ are free in C, then $c_1\colon S_1, c_2\colon S_2 \in \Gamma, c_1\colon (c_2,n_1,\vec{b}_1,S_1) \in \Delta, c_2\colon (c_1,n_2,\vec{b}_2,S_2) \in \Delta$, $\mathrm{bound}(S_i) \leq n_i$ for $i = 1,2$, and $S_1/\vec{b}_1 \asymp S_2/\vec{b}_2$,

This enables the runtime safety theorem to be stated as follows.

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

1. C'' is $\langle v\rangle$ or $\langle \mathsf{send}\ v\rangle$;

2. C'' is $\langle E[\mathsf{receive}\ c]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$;
3. C'' is $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}]\rangle$ and $c \mapsto (\_, \_, \varepsilon) \in C'$.

As usual, we make use of a type preservation theorem; the interesting part of the theorem is stated below, although we will see later that a stronger statement is needed in order to complete an inductive proof.

Theorem (Type Preservation)
If $\Gamma \vdash C \triangleright \Delta$ is well-buffered and $C \longrightarrow C'$ then there exist Γ' and Δ' such that $\Gamma' \vdash C' \triangleright \Delta'$ is well-buffered.

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 & Felleisen (1994). We omit the proofs of most lemmas, which are either 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 $\Gamma_1 \vdash e\colon T$ and $\mathsf{un}(\Gamma_2)$ and $\Gamma_1 + \Gamma_2$ is defined then $\Gamma_1 + \Gamma_2 \vdash e\colon T$.

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

Lemma 12 (*Typability of Subterms*)
If $\mathscr{D}$ is a derivation of $\Gamma \vdash E[e]\colon T$ or $\Gamma \vdash \langle E[e]\rangle \triangleright \Delta$ then there exist Γ_1, Γ_2 and U such that $\Gamma = \Gamma_1 + \Gamma_2$, $\mathscr{D}$ has a subderivation $\mathscr{D}'$ concluding $\Gamma_1 \vdash e\colon U$ 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 $\Gamma_1 + \Delta_1 + \Delta_2 \vdash E[e]\colon T$ or $\Gamma_1 + \Delta_1 + \Delta_2 \vdash \langle E[e]\rangle \triangleright \Delta$
2. $\mathscr{D}'$ is a subderivation of $\mathscr{D}$ concluding $\Delta_1 + \Delta_2 \vdash e\colon U$
3. the position of $\mathscr{D}'$ in $\mathscr{D}$ corresponds to the position of the hole in $E[\,]$
4. $\Delta_1 + \Delta_3 \vdash e'\colon U$
5. $\Gamma_1 + \Delta_1 + \Delta_3$ is defined

then $\Gamma_1 + \Delta_1 + \Delta_3 \vdash E[e']\colon T$ or $\Gamma_1 + \Delta_1 + \Delta_3 \vdash \langle E[e']\rangle \triangleright \Delta$ 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.

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.

Lemma 17

For all Γ, $\vec{b}$ and S, if $\Gamma \vdash \vec{b}\,\text{matches}\,S$ then $|\vec{b}| \leqslant \text{bound}(S)$.

Proof
By induction on the derivation of $\Gamma \vdash \vec{b}\,\text{matches}\,S$ with a case-analysis on the last rule (equivalently, on the form of $\vec{b}$). Let f and F be as defined in Definition 4.

- $\vec{b} = \varepsilon$. This case is trivial, as $|\varepsilon| = 0$.
- $\vec{b} = v\vec{b'}$. From the derivation, $S = ?U.S'$ and there exists Γ' such that $\Gamma' \vdash \vec{b'}\,\text{matches}\,S'$. Because $f = F(f)$, $f(S) = 1 + f(S')$. By the induction hypothesis, $|\vec{b'}| \leqslant f(S')$. Therefore $|\vec{b}| \leqslant f(S)$. This reasoning is valid even if $f(S') = \infty$.
- $\vec{b} = l\vec{b'}$. From the derivation, $S = \&\langle l_i \colon S_i\rangle_{i\in I}$ with $l = l_j$ for some $j \in I$, and there exists Γ' such that $\Gamma' \vdash \vec{b'}\,\text{matches}\,S_j$. Because $f = F(f)$, $f(S) = 1 + max_{i\in I}\{f(S_i)\}$. By the induction hypothesis, $|\vec{b'}| \leqslant f(S_j) \leqslant max_{i\in I}\{f(S_i)\}$. Therefore $|\vec{b}| \leqslant f(S)$. Again, this reasoning is valid even if some of the $f(S_i)$ are ∞.

□

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

Lemma 19
If $C \equiv C'$ then C and C' have exactly the same buffers, in the sense of Definition 6.

Lemma 20
If $\Gamma \vdash C \triangleright \Delta$ is well-buffered and $C \equiv C'$ then $\Gamma \vdash C' \triangleright \Delta$ is well-buffered.

Lemma 21
1. If $\Gamma \vdash \vec{b}\,\text{matches}\,S$ and $\Gamma' \vdash v \colon T$ and $S/\vec{b} = ?T.S'$ then $\Gamma + \Gamma' \vdash \vec{b}v\,\text{matches}\,S$.
2. If $\Gamma \vdash \vec{b}\,\text{matches}\,S$ and $S/\vec{b} = \&\{\ldots, l \colon S, \ldots\}$ then $\Gamma \vdash \vec{b}l\,\text{matches}\,S$.

Definition 22
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 buffer $c \mapsto (d, n, \vec{b})$ whose contents are changed by the reduction.

Theorem 23 (*Type Preservation*)
If $\Gamma \vdash C \triangleright \Delta$ is well-buffered and $C \longrightarrow C'$ then there exist Γ' and Δ' such that $\Gamma' \vdash C' \triangleright \Delta'$ is well-buffered and $\text{dom}(\Gamma') = \text{dom}(\Gamma)$ and $\text{dom}(\Delta') = \text{dom}(\Delta)$. Furthermore, if the reduction originates from R-RECEIVE or R-BRANCH, then for every $c \colon (d, n, \vec{b}, S) \in \Delta$ and corresponding $c \colon (d, n, \vec{b'}, S') \in \Delta'$, $S/\vec{b} = S'/\vec{b'}$. Finally, for every c, if the reduction does not change the contents of buffer c, then $\Gamma'(c) = \Gamma(c)$ and $\Delta'(c) = \Delta(c)$.

Proof
By induction on the derivation of $C \longrightarrow C'$, with a case-analysis on the last rule.

First note that in each case, if $c \colon S \in \Gamma$ and $c \colon S' \in \Gamma'$, either $S' = S$ or $S \mapsto S'$. So Lemma 5 guarantees that the conditions $\text{bound}(S_i) \leq n_i$ in the definition of well-buffering are satisfied. We will not discuss these condition in the analysis of each case below.

- 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 $\Gamma \vdash \langle E[e']\rangle \triangleright \emptyset$. The remaining conditions are trivially satisfied because the configuration contains no buffers.
- R-FORK. We have $\langle E[\mathsf{fork}\ e\ e']\rangle \longrightarrow \langle e\rangle \parallel \langle E[e']\rangle$ and $\Gamma \vdash \langle E[\mathsf{fork}\ e\ e']\rangle \triangleright \emptyset$. 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 $\Gamma_1 \vdash \mathsf{fork}\ e\ e' : T$. The end of $\mathscr{D}'$ has the form

$$\frac{\Gamma_3 \vdash e : U \quad \Gamma_4 \vdash e' : T \quad \mathsf{un}(U)}{\Gamma_1 \vdash \mathsf{fork}\ e\ e' : T}$$

  where $\Gamma_3 + \Gamma_4 = \Gamma_1$. By Lemma 13 we have $\Gamma_2 + \Gamma_4 \vdash \langle E[e']\rangle \triangleright \emptyset$ and so we can construct the derivation

$$\frac{\Gamma_3 \vdash \langle e\rangle \triangleright \emptyset \qquad \Gamma_2 + \Gamma_4 \vdash \langle E[e']\rangle \triangleright \emptyset}{\Gamma \vdash \langle e\rangle \parallel \langle E[e']\rangle \triangleright \emptyset}$$

  The remaining conditions are trivially satisfied because the configuration contains no buffers.
- 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

$$\frac{\Gamma_1 \vdash C \triangleright \Delta_1 \qquad \Gamma_2 \vdash C'' \triangleright \Delta_2}{\Gamma_1 + \Gamma_2 \vdash C \parallel C'' \triangleright \Delta_1 + \Delta_2}\ \text{T-PAR}$$

  where $\Gamma_1 + \Gamma_2 = \Gamma$ and $\Delta_1 + \Delta_2 = \Delta$.
  Because $\Gamma \vdash C \parallel C'' \triangleright \Delta$ is well-buffered, $\Gamma_1 \vdash C \triangleright \Delta_1$ is well-buffered. By induction, $\Gamma_1' \vdash C' \triangleright \Delta_1'$ is well-buffered with $\mathrm{dom}(\Gamma_1') = \mathrm{dom}(\Gamma_1)$ and $\mathrm{dom}(\Delta_1') = \mathrm{dom}(\Delta_1)$. Hence $\Gamma_1' + \Gamma_2$ and $\Delta_1' + \Delta_2$ are defined and we can derive $\Gamma_1' + \Gamma_2 \vdash C' \parallel C'' \triangleright \Delta_1' + \Delta_2$, with $\mathrm{dom}(\Gamma_1' + \Gamma_2) = \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta_1' + \Delta_2) = \mathrm{dom}(\Delta_1 + \Delta_2)$.
  We now show that $\Gamma_1' + \Gamma_2 \vdash C' \parallel C'' \triangleright \Delta_1' + \Delta_2$ is well-buffered. Consider $c_1 \mapsto (c_2, n_1, \vec{b_1})$ and $c_2 \mapsto (c_1, n_2, \vec{b_2})$ in $C \parallel C''$. If c_1 and c_2 are both in C then preservation of well-buffering is sufficient. Otherwise suppose that only c_1 is in C. If the reduction changes c_1 then the original rule must be R-RECEIVE or R-BRANCH, and so preservation of $S/\vec{b}$ guarantees well-buffering. If the reduction does not change c_1 then by induction its type is also unchanged and hence the part of the well-buffering condition concerning c_1 is satisfied.
  The remaining conditions follow directly by induction.
- R-STRUCT. Follows from the induction hypothesis and Lemmas 9 and 20.
- R-NEW. We have $(\nu c_1 c_2)C \longrightarrow (\nu c_1 c_2)C'$ because $C \longrightarrow C'$. We have $\Gamma \vdash (\nu c_1 c_2)C \triangleright \Delta$ with the derivation

$$\frac{\begin{array}{c}\mathrm{bound}(S_i) \leqslant n_i \qquad S_1/\vec{b_1} \asymp S_2/\vec{b_2} \\ \Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \triangleright \Delta + c_1 : (c_2, n_1, \vec{b_1}, S_1) + c_2 : (c_1, n_2, \vec{b_2}, S_2)\end{array}}{\Gamma \vdash (\nu c_1 c_2)C \triangleright \Delta}$$

  By induction we have $\Gamma' + c : S_1' + d : S_2' \vdash C' \triangleright \Delta' + c : (d, n_1, \vec{b_1'}, S_1') + d : (c, n_2, \vec{b_2'}, S_2')$ well-buffered with $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta') = \mathrm{dom}(\Delta)$. The definition of

well-buffering gives the hypotheses of the derivation

$$\frac{\mathrm{bound}(S_i') \leqslant n_i \qquad S_1'/\vec{b}_1' \asymp S_2'/\vec{b}_2' \qquad \Gamma' + c_1 \colon S_1' + c_2 \colon S_2' \vdash C \triangleright \Delta + c_1 \colon (c_2, n_1, \vec{b}_1') + c_2 \colon (c_1, n_2, \vec{b}_2')}{\Gamma' \vdash (\nu c_1 c_2) C' \triangleright \Delta'}$$

and $\Gamma' \vdash (\nu c_1 c_2) C' \triangleright \Delta'$ is also well-buffered. 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 $\Gamma \vdash \langle E[\mathsf{request}\ n\ x] \rangle \parallel \langle E'[\mathsf{accept}\ n'\ x] \rangle \triangleright \Delta$ well-buffered with the derivation

  $$\frac{\dfrac{\Gamma_1 \vdash E[\mathsf{request}\ n\ x] \colon T_1}{\Gamma_1 \vdash \langle E[\mathsf{request}\ n\ x] \rangle \triangleright \Delta_1} \qquad \dfrac{\Gamma_2 \vdash E'[\mathsf{accept}\ n'\ x] \colon T_2}{\Gamma_2 \vdash \langle E'[\mathsf{accept}\ n'\ x] \rangle \triangleright \Delta_2}}{\Gamma_1 + \Gamma_2 \vdash \langle E[\mathsf{request}\ n\ x] \rangle \parallel \langle E'[\mathsf{accept}\ n'\ x] \rangle \triangleright \Delta_1 + \Delta_2}$$

  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 $\Gamma_3 \vdash \mathsf{request}\ n\ x \colon \overline{S}$, with $\Gamma_3 \vdash x \colon \langle S \rangle^{\mathsf{r}}$ and $\mathrm{bound}(\overline{S}) \leqslant n$. Similarly there exist Γ_5 and Γ_6 such that $\Gamma_2 = \Gamma_5 + \Gamma_6$ and $\Gamma_5 \vdash \mathsf{accept}\ n'\ x \colon S$, with $\Gamma_5 \vdash x \colon \langle S \rangle^{\mathsf{a}}$ and $\mathrm{bound}(S) \leqslant n'$.
  Taking c and d to be fresh channels, Lemma 13 gives $\Gamma_1 + c \colon \overline{S} \vdash \langle E[c] \rangle \triangleright \Delta_1$ and $\Gamma_2 + d \colon S \vdash \langle E[d] \rangle \triangleright \Delta_2$. We also have $\vdash c \mapsto (d, n, \varepsilon) \triangleright c \colon (d, n, \varepsilon, \mathsf{end})$ and $\vdash d \mapsto (c, n', \varepsilon) \triangleright d \colon (c, n', \varepsilon, \mathsf{end})$ from which we use T-PAR and T-NEW to derive

  $$\Gamma_1 + \Gamma_2 \vdash (\nu cd)(c \mapsto (d, n, \varepsilon) \parallel d \mapsto (c, n', \varepsilon)\} \parallel \langle E[c] \rangle \parallel \langle E'[d] \rangle) \triangleright \Delta_1 + \Delta_2$$

  which is well-buffered by the original assumption. T-NEW requires $\mathrm{bound}(\overline{S}) \leqslant n$ and $\mathrm{bound}(S) \leqslant n'$, which are among the data above. The remaining conditions are trivially satisfied because there are no buffers in the initial configuration and the rule is not R-RECEIVE or R-BRANCH.
- 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 $\Gamma \vdash c \mapsto (d, n', \vec{b}') \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\mathsf{send}\ v\ c] \rangle \triangleright \Delta$ well-buffered with the derivation

  $$\frac{\dfrac{\Gamma_1 \vdash \vec{b}' \,\mathrm{matches}\, S_1}{\Gamma_1 \vdash c \mapsto (d, n', \vec{b}') \triangleright c \colon (d, n', \vec{b}', S_1)} \qquad \dfrac{\Gamma_2 \vdash \vec{b} \,\mathrm{matches}\, S_2}{\Gamma_2 \vdash d \mapsto (c, n, \vec{b}) \triangleright d \colon (c, n, \vec{b}, S_2)} \qquad \Gamma' \vdash \langle E[\mathsf{send}\ v\ c] \rangle \triangleright \Delta'}{\Gamma' + \Gamma_1 + \Gamma_2 \vdash c \mapsto (d, n', \vec{b}') \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\mathsf{send}\ v\ c] \rangle \triangleright \Delta' + c \colon (d, n', \vec{b}', S_1) + d \colon (c, n, \vec{b}, S_2)}$$

  where $\Gamma' + \Gamma_1 + \Gamma_2 = \Gamma$ and $\Delta' + c \colon (d, n', \vec{b}', S_1) + d \colon (c, n, \vec{b}, S_2) = \Delta$.
  By Lemma 12 there exist Γ_3 and Γ_4 such that we have the subderivation

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

  with $\Gamma' = \Gamma_3 + \Gamma_4$ and $\Gamma_3 = \Gamma_5 + \Gamma_6 + c \colon !T.S$. Lemma 13 gives $\Gamma_4 + \Gamma_6 + c \colon S \vdash \langle E[c] \rangle \triangleright \Delta$.

Because the original typing judgement is well-buffered, we have $S_1 = !T.S$ and $!T.S/\vec{b}' \asymp S_2/\vec{b}$. Because $\Gamma_1 \vdash \vec{b}'$ matches $!T.S$ we have $\vec{b}' = \varepsilon$ and hence $!T.S/\vec{b}' = !T.S$. Therefore $?T.\overline{S} <: S_2/\vec{b}$, so $S_2/\vec{b} = ?T'.S'$ with $\overline{S} <: S'$ and $T <: T'$. Lemma 21 gives $\Gamma_2 + \Gamma_5 \vdash \vec{b}v$ matches S_2 and we can build the derivation

$$\frac{\dfrac{\Gamma_1 \vdash \vec{b}' \text{ matches } S}{\Gamma_1 \vdash c \mapsto (d,n',\vec{b}') \triangleright c\colon (d,n',\vec{b}',S)} \quad \dfrac{\Gamma_2+\Gamma_5 \vdash \vec{b}v \text{ matches } S_2}{\Gamma_2+\Gamma_5 \vdash d \mapsto (c,n,\vec{b}v) \triangleright d\colon (c,n,\vec{b}v,S_2)} \quad \Gamma_4+\Gamma_6+c\colon S \vdash \langle E[c]\rangle \triangleright \Delta'}{\Gamma_1+\Gamma_2+\Gamma_4+\Gamma_5+\Gamma_6+c\colon S \vdash c \mapsto (d,n',\vec{b}') \parallel d \mapsto (c,n,\vec{b}v) \parallel \langle E[c]\rangle \triangleright \Delta' + c\colon (d,n',\vec{b}',S) + d\colon (c,n,\vec{b}v,S_2)}$$

It remains to show that the conclusion is well-buffered. This requires $S/\vec{b}' \asymp S_2/\vec{b}v$. We have $S/\vec{b}' = S/\varepsilon = S$ and $S_2/\vec{b}v = S'$. Finally, by definition, $S \asymp S'$ because $\overline{S} <: S'$.

The condition on unchanged buffers is satisfied, as it can be seen from the derivation above that their types do not change.

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

$$\frac{\dfrac{\dfrac{\Gamma_1 \vdash v\colon T \quad \Gamma_2 \vdash \vec{b} \text{ matches } S}{\Gamma_1+\Gamma_2 \vdash v\vec{b} \text{ matches } ?T.S}}{\Gamma_1+\Gamma_2 \vdash c \mapsto (d,n,v\vec{b}) \triangleright c\colon (d,n,v\vec{b},?T.S)} \quad \Gamma' \vdash \langle E[\mathsf{receive}\ c]\rangle \triangleright \Delta'}{\Gamma \vdash c \mapsto (d,n,v\vec{b}) \parallel \langle E[\mathsf{receive}\ c]\rangle \triangleright \Delta}$$

where $\Gamma' + \Gamma_1 + \Gamma_2 = \Gamma$ and $\Delta' + c\colon (d,n,v\vec{b},?T.S) = \Delta$.
By Lemma 12 there exist Γ_3 and Γ_4 such that we have the subderivation

$$\frac{\Gamma_3 + c\colon ?T.S \vdash c\colon ?T.S}{\Gamma_3 + c\colon ?T.S \vdash \mathsf{receive}\ c\colon T \otimes S}$$

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

$$\frac{\dfrac{\Gamma_2 \vdash \vec{b} \text{ matches } S}{\Gamma_2 \vdash c \mapsto (d,n,\vec{b}) \triangleright c\colon (d,n,\vec{b},S)} \quad \Gamma_1+\Gamma_3+\Gamma_4 \vdash \langle E[(v,c)]\rangle \triangleright \Delta'}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_4+c\colon S \vdash c \mapsto (d,n,\vec{b}) \parallel \langle E[(v,c)]\rangle \triangleright \Delta' + c\colon (d,n,\vec{b},S)}$$

Well-buffering is trivial because there is only one buffer. The remaining condition we need is that $S/\vec{b} = ?T.S/v\vec{b}$, which follows from the definitions.

- R-BRANCH. Similar to the previous case.

□

Theorem 24 (*Runtime Safety*)
Let $\Gamma \vdash C \triangleright \Delta$ be well-buffered, and $C \longrightarrow^* C'$. 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 $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}]\rangle$ and $c \mapsto (\_,\_,\varepsilon) \in C'$.

Proof
By Type Preservation, Theorem 23, we know that $\Gamma' \vdash C' \triangleright \Delta'$ is well-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 $\langle E[\mathsf{case}\ c\ \mathsf{of}\ \{l_i\colon e_i\}_{i\in I}]\rangle$ 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 $|\vec{b}| \geq n$;
4. C'' is $\langle E[\mathsf{select}\ l\ c]\rangle$, and $c \mapsto (d,\_,\_), d \mapsto (\_,n,\vec{b}) \in C'$, and $|\vec{b}| \geq n$;
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 $c\colon ?T.S$. If c is bound in C', then the derivation includes an application of rule T-NEW to a sub-configuration $\Gamma'' \vdash (\nu cd)C''' \triangleright \Delta''$, at which point we know that $\Gamma''' \vdash l\vec{b}\,\mathsf{matches}\,\Gamma''(c)$, from which we conclude that $\Gamma''(c)$ is of the form $\&\langle..l\colon S..\rangle$, hence contradiction. If c is free in C', then we reach the same contradiction based on the fact that $\Gamma' \vdash C' \triangleright \Delta'$ is well-buffered.
2. Similar to the previous case.
3. The main point is to show that the assumption $|\vec{b}| \geq n$ leads to a contradiction. Consider the information in case R-SEND of the proof of Type Preservation. From $\Gamma_2 + \Gamma_5 \vdash \vec{b}v\,\mathsf{matches}\,S_2$ and Lemma 17 we have $\mathrm{bound}(S_2) \geqslant |\vec{b}v| = |\vec{b}| + 1$. By well-buffering we have $\mathrm{bound}(S_2) \leqslant n$, 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.

□

Finally, we observe that the expression "$\mathsf{accept}\ n\ a$" can safely be replaced by "$\mathsf{accept}\ \mathrm{bound}(S)\ a$" where $a\colon \langle S\rangle^{\mathsf{a}}$ in the current environment, and similarly for request. In other words, the compiler can infer the necessary buffer sizes. Also, when a channel of type S is used, e.g. 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.*, 2006, 2004), 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.* (2008) and our own (Gay *et al.*, 2008)

for object-oriented languages. The languages in (Dezani-Ciancaglini *et al.*, 2006; 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, a single thread cannot 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 & Thiemann (2004a) show how to implement session types on top of the Haskell programming language; Neubauer & Thiemann (2004b) model software components as concurrent functional processes, and use session types to extract the smallest protocol required by each process; Neubauer & Thiemann (2005) address the problem of program transformation, from sequential to multi-tier, guided by session types.

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

Yoshida & Vasconcelos (2007) show 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 & Hole (2005), they refer 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 generalizing session types to multi-party protocols, does not allow a thread to acquire both endpoints of a channel.

Cyclone (Grossman, 2003; Grossman *et al.*, 2002), *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 of Gay (2008). We should also investigate the relationship with other forms of static analysis, including type and effect systems (Amtoft *et al.*, 1999). In the longer term we intend to formalize 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. A thorough understanding of the functional case provides a good foundation for the object-oriented case.

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) use 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.

Acknowledgements

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 CITI, and project Space–Time–Types, POSC/EIA/55582/2004). 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).

References

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

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

Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S., & Giachino, E. (2008). Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science*. to appear.

Coppo, M., Dezani-Ciancaglini, M., & Yoshida, N. (2007). Asynchronous Session Types and Progress for Object-Oriented Languages. *Pages 1–31 of:* Bonsangue, M., & Johnsen, E. B. (eds), *FMOODS'07*. LNCS, vol. 4468. Springer.

DeLine, R., & Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. *Pages 59–69 of: PLDI*. SIGPLAN Notices, vol. 36(5). ACM Press.
Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., & Drossopolou, S. (2005). A distributed object-oriented language with session types. *Pages 299–318 of: TGC*. LNCS, vol. 3705. Springer.
Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., & Drossopoulou, S. (2006). Session types for object-oriented languages. *Pages 328–352 of: ECOOP*. LNCS, vol. 4067. Springer.
Fähndrich, M., & DeLine, R. (2002). Adoption and focus: practical linear types for imperative programming. *Pages 13–24 of: PLDI*. SIGPLAN Notices, vol. 37(5). ACM Press.
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. *SIGOPS Oper. Syst. Rev.*, **40**(4), 177–190.
Gay, S., Vasconcelos, V. T., & Ravara, A. 2003 (Mar.). *Session types for inter-process communication*. TR 2003–133. Department of Computing, University of Glasgow.
Gay, S. J. (2006). *Subtyping between standard and linear function types*. `http://www.dcs.gla.ac.uk/~simon/publications/subslf.pdf`.
Gay, S. J. (2008). Bounded polymorphism in session types. *Mathematical Structures in Computer Science*. To appear.
Gay, S. J., & Hole, M. J. (2005). Subtyping for session types in the pi calculus. *Acta Informatica*, **42**(2/3), 191–225.
Gay, S. J., & Vasconcelos, V. T. (2007). *Asynchronous functional session types*. Tech. rept. TR-2007-251. Department of Computing Science, University of Glasgow.
Gay, S. J., Vasconcelos, V. T., & Ravara, A. (2008). *Dynamic interfaces*. Submitted.
Grossman, D. (2003). Type-safe multithreading in Cyclone. *Pages 13–25 of: TLDI*. SIGPLAN Notices, vol. 38(3). ACM Press.
Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., & Cheney, J. (2002). Region-based memory management in Cyclone. *Pages 282–293 of: PLDI*. SIGPLAN Notices, vol. 37(5). ACM Press.
Honda, K. (1993). Types for dyadic interaction. *Pages 509–523 of: CONCUR*. LNCS, vol. 715. Springer.
Honda, K., Vasconcelos, V. T., & Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. *Pages 122–138 of: ESOP*. LNCS, vol. 1381. Springer.
Honda, K., Yoshida, N., & Carbone, M. (2008). Multiparty asynchronous session types. *Pages 273–284 of: POPL*. SIGPLAN Notices, vol. 43(1). ACM Press.
Neubauer, M., & Thiemann, P. (2004a). An implementation of session types. *Pages 56–70 of: PADL*. LNCS, vol. 3057. Springer.
Neubauer, M., & Thiemann, P. (2004b). Protocol specialization. *Pages 246–261 of:* Chin, W.-N. (ed), *Aplas*. Lecture Notes in Computer Science, vol. 3302. Springer.
Neubauer, M., & Thiemann, P. (2004c). *Session types for asynchronous communication*. Unpublished.
Neubauer, M., & Thiemann, P. (2005). From sequential programs to multi-tier applications by program transformation. *Pages 221–232 of:* Palsberg, J., & Abadi, M. (eds), *Popl*. ACM.

Peyton Jones, S., & Wadler, P. (1993). Imperative functional programming. *Popl*. ACM.
Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.
Takeuchi, K., Honda, K., & Kubo, M. (1994). An interaction-based language and its typing system. *Pages 398–413 of: PARLE*. LNCS, vol. 817. Springer.
Vallecillo, A., Vasconcelos, V. T., & Ravara, A. (2006). Typing the behavior of software components using session types. *Fundamenta Informaticae*, **73**(4), 583–598.
Vasconcelos, V. T., Ravara, A., & Gay, S. J. (2004). Session types for functional multithreading. *Pages 497–511 of: CONCUR*. LNCS, vol. 3170. Springer.
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.
W3C. (2005). *Web Services Choreography Description Language Version 1.0*. `http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/`.
Walker, D. (2005). Substructural type systems. *Pages 3–43 of:* Pierce, B. C. (ed), *Advanced Topics in Types and Programming Languages*. MIT Press.
Wright, A. K., & Felleisen, M. (1994). A syntactic approach to type soundness. *Information and Computation*, **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.