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.

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

1 Introduction

The concept of service-oriented computing has transformed the design and implementation of large-scale distributed systems, including online consumer services such as e-commerce sites. It is now common practice to build a system by gluing together the online services of several providers: for example, online travel agents, centralised hotel reservation systems, and online shops. Such systems are characterised by detailed and complex protocols, separate development of components and re-use of existing components, and strict requirements for availability and correctness. In this setting, formal development methods and 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 [14, 20, 15]. In this approach, communication protocols are expressed as types, so that static type-checking can be used to verify that agents observe the correct protocols. For example, the type $S = \&\langle \text{service: } ?\text{Int}!\text{Int}, \text{quit: End} \rangle$ 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 $S^*$, 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 [25] and operating system services [6]. By incorporating correspondence assertions, the behavioural guarantees offered by session types have been strengthened and applied to security analysis [2]. A theory of subtyping for session types has been developed [10] and adapted for specifying distributed software components [21]. 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 [23, 22] was the first proposal for a functional language with session types. Neubauer and Thiemann [17] 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. [5, 4] and is included in the work of Fähndrich et al. [6].

In the present paper we define a multithreaded functional language with session types, unifying and simplifying several strands of previous work and extending the preliminary version [11], and clarifying the relationship between session types and standard functional type theory. The contributions of the paper are as follows.

1. We formalize an operational semantics in which communication is buffered, instead of assuming synchronization between send and receive, as in previous work [22, 23]. This is more realistic, and means that send and select never block. The semantics is similar to, but simpler than, unpublished work by Neubauer and Thiemann [18]. Fähndrich et al. [6] 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. [6]. 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: $T.S \rightarrow T \otimes S$ so that the channel, with its new type, is returned with the received value. This gives a huge simplification of our previous work [22, 23] which instead used a complex system of alias types.

4. We include two forms of subtyping: the standard subtyping relation for session types [10] and a novel form of subtyping between standard and linear function types [8]. The 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 typing system and gives the main results of the paper. Section 5 discusses related and future work.

2 Example: Business Protocol

We present a small example containing typical features of many web service business protocols [4, 25]. A mother and her young son are using an online book shop. The shop implements a simple protocol described by the session type

\[ \text{Shop} = \&\langle \text{add} : ?\text{Book} . \text{Shop} , \text{checkout} : ?\text{Card} . ?\text{Address} . \text{End} \rangle \]

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.

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 \text{Shop}. We express this as \text{shopAccess}: [\text{Shop}]. A name such as \text{shopAccess} is analogous to a URL or an IP address, depending on the kind of service. The shop will contain an expression accept \text{shopAccess} and the shopper will contain an expression request \text{shopAccess}. At runtime these expressions interact to create a new private channel, which in the shop has type \text{Shop} and in the shopper has the dual type (\text{Shopper} = \text{Shop}), where ! means send:

\[ \text{Shopper} = \oplus\langle \text{add} : !\text{Book} . \text{Shopper} , \text{checkout} : !\text{Card} . !\text{Address} . \text{End} \rangle \]

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] → 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 plays the role of a gift voucher. Communication between mother and son is also described by a session type:
Son = ?( Book → Book ) . ! Book . End

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

mother :: Card → Address → [Shop] → [Son] → 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] → 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, typed in a suitable environment (which should also include the types of all of the functions, as well as mCard etc):
shopAccess: [Shop], sonAccess: [Son] ⊢
(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 and 4.

– Channels, such as \( c \) in \( \text{mother} \), are linear values; session types are linear types. The linear function type constructor \( \rightarrow \) appears in the type of \( \text{voucher} \) because applying \( \text{voucher} \) to a channel of type \( \text{Shop} \) 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, \( \text{Son} \) cannot duplicate the voucher and order more than one book.

– Operations on channels, such as \( \text{send} \) and \( \text{select} \), return the channel after communicating on it. Our programming style is to repeatedly re-bind the channel name using \( \text{let} \); each \( c \) is of course a fresh bound variable. The \( \text{receive} \) operation returns the value received and the channel, as a (linear) pair which is split by a \( \text{let} \) construct. In the static type system, the channel type returned by, for example, \( \text{send} \), 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 \( \text{voucher} \) in the example.

Observe that the type \( \text{Shop} \) allows an unbounded sequence of messages in the same direction, alternating between \( \text{add} \) labels and book details. The shop would therefore require a potentially unbounded buffer for incoming messages. However, Fähndrich et al. [6] have pointed out that if the session type does not allow unbounded sequences of messages in the same direction then it is possible to obtain a static upper bound on the size of the buffer. This is also true in our system, and we give a formal proof in Section 4. For example, the type \( S \) in Section 1 yields a bound of 2 because after sending \( \text{service} \) and an \( \text{Int} \), the client must wait to receive an \( \text{Int} \). 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 type-checking would enable a compiler to generate code to deallocate buffer space; the extreme case is that the compiler can also work out when to deallocate buffer space. 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.

Three variations of the example illustrate subtyping and sending channels on channels.

Subtyping function types: Changing the function \( \text{voucher} \). The mother decides that \( \text{voucher} \) should not order the book; she will complete the order herself. She defines

\[ \text{shopAccess: [Shop], sonAccess: [Son]} \]
v ::= c | x | λx.e | fix | (v, v) | request n | accept n | send | send v | receive

e ::= v | ee | (e, e) | let (x, x) = e in e | fork e e | select l e | case e of {li : ei}i∈I

b ::= v | l

C ::= ⟨e⟩ | c ↦→ (c, k, ⃗b) | C ∥ C | (vce)C

E ::= [] | Ee | vE | (E, e) | (v, E) | let (x, x) = E in e | select l E | case E of {li : ei}i∈I

Fig. 1. Syntax.

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 voucher b

is still typable; there is no need to change the type Son.

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 and its dual to

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

We have Shop <: NewShop and NewShopper <: Shopper. If the type of shopAccess in the global environment changes to [NewShop] then expression request shopAccess

in mother returns a channel of type NewShopper. The subtyping relationship means that this channel can still be given to voucher as a parameter.

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. For example, suppose that the shop uses a separate service, shipper, to arrange delivery of the order. When shop has received the customer’s credit card details, it just passes the channel to shipper. When the customer sends her address, it goes directly to shipper. The session type used for communication between shop and shipper is as follows; note the occurrence of the session type ?Address.End as the type of the message.

Shipper = ?⟨Address . End⟩ . End

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

3 Syntax and Operational Semantics
\[(\lambda x.e)v \longrightarrow e[v/x] \quad \text{fix} \quad (\lambda x.e) \longrightarrow e[\text{fix} (\lambda x.e)/x]\] (R-App, R-Fix)

let \((x, y) = (v, u)\) in \(e \longrightarrow e[v/x]u/y\) (R-Split)

**Fig. 2.** Reduction of expressions.

\[
\begin{align*}
&\frac{e \longrightarrow e'}{\langle E[e]\rangle \longrightarrow \langle E[e']\rangle} & \quad (E[\text{fork } e ‘]) \longrightarrow \langle e \rangle \parallel \langle E[e']\rangle & \quad (\text{R-Thread,R-Fork}) \\
&\frac{C \longrightarrow C'}{C \parallel C'' \longrightarrow C'' \parallel C'''} & \quad (\nu c d) C \longrightarrow (\nu c d) C' & \quad \text{if } C \equiv C'' \longrightarrow C' \longrightarrow C''' \equiv C'''' & \quad (\text{R-Par,R-New,R-Struct}) \\
&\frac{E[\text{request } n \ x]}{\langle E[\text{accept } n’ \ x]\rangle} & \quad (\nu c d) (c \mapsto (d, n, \varepsilon) \parallel d \mapsto (c, n’, \varepsilon)) \parallel \langle E[c]\rangle \parallel \langle E'[d]\rangle & \quad (\text{R-Init}) \\
&\frac{c \mapsto (d, n’, \vec{b}) \parallel d \mapsto (c, n, \vec{b})}{c \mapsto (d, n’, \vec{b}) \parallel d \mapsto (c, n, \vec{b})} & \quad \langle E[\text{send } v \ c]\rangle & \quad \text{if } |\vec{b}| < n & \quad (\text{R-Send}) \\
&\frac{c \mapsto (d, n, \vec{b}) \parallel d \mapsto (c, n, \vec{b})}{c \mapsto (d, n, \vec{b}) \parallel d \mapsto (c, n, \vec{b})} & \quad \langle E[\text{select } l \ c]\rangle & \quad \text{if } |\vec{b}| < n & \quad (\text{R-Select}) \\
&\frac{c \mapsto (d, n, v\vec{b}) \parallel d \mapsto (c, n’, \vec{b})}{c \mapsto (d, n, v\vec{b}) \parallel d \mapsto (c, n’, \vec{b})} & \quad \langle E[\text{receive } c]\rangle & \quad \text{if } j \in I & \quad (\text{R-Receive}) \\
&\frac{c \mapsto (d, n, l_j\vec{b}) \parallel d \mapsto (c, n’, \vec{b})}{c \mapsto (d, n, l_j\vec{b}) \parallel d \mapsto (c, n’, \vec{b})} & \quad \langle E[\text{case } c \text{ of } \{l_i : e_i\}_{i \in I}]\rangle & \quad \text{if } j \in I & \quad (\text{R-Branch})
\end{align*}

**Fig. 3.** Reduction of configurations.

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

Figure 2 defines reduction of expressions by means of standard rules. To simplify the presentation of inter-thread reduction, we use evaluation contexts (Figure 1) [26] and structural equivalence on configurations. 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 \(e\). Rules R-Thread and R-Fork in Figure 3 define intra-thread reduction steps.

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).
We write $fc(C)$ for the free channels of a configuration $C$, defined to be the channel endpoints occurring free (i.e. not bound by $\lambda$, $\nu$, or let) in all the threads and queues of $C$. Structural equivalence, the smallest relation satisfying the rules

\begin{align*}
C_1 \parallel C_2 \equiv C_2 \parallel C_1 & \quad \text{(E-Comm)} \\
C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 & \quad \text{(E-Assoc)} \\
C_1 \parallel (\nu cd)C_2 \equiv (\nu cd)(C_1 \parallel C_2) & \quad \text{if } c,d \not\in fc(C_1) \quad \text{(E-Scope)}
\end{align*}

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

Rule R-Init synchronizes two threads trying to start a new connection on a common name $x$, which must be free in the 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 with the buffer size declared by request or accept. Symbol $\varepsilon$ denotes an empty queue. (The example omitted the buffer sizes because they can be inferred; see Sec. 4).

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. 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 $c$. In either case, again, $c$ can be used for further interaction.

4 Typing

We now introduce a static type system for our language. The syntax of types is defined in Figure 4. Session types $S$ will be 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 $S$. $\&\langle l_i : S_i\rangle_{i \in I}$ 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, $\oplus \langle l_i : S_i \rangle_{i \in I}$ 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 $\mu X \cdot S$, which are required to be contractive, i.e. containing no subexpression of the form $\mu X_1 \cdots \mu X_n \cdot X_1$.

We define $\mathit{unfold}(\mu X \cdot S) = \mathit{unfold}(S((\mu X \cdot S)/X))$, and $\mathit{unfold}(T) = T$ for non-recursive types $T$; contractivity guarantees that this definition terminates.

General types are denoted by $T$, including session types $S$ as one case. Data types such as $\operatorname{Int}$ and $\operatorname{Bool}$, or compound data types such as non-linear pairs, or general recursive types, can easily be added. The type $[S]$ describes a name that can be used to establish a session. If a typed name $\alpha : [S]$ occurs in the global environment then a matching request $n \alpha$ and accept $n' \alpha$ 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 $\overline{S}$ (the dual of $S$, defined below).

The type system includes a subtyping relation. This combines the standard definition of subtyping for session types [10], the standard subtyping rules for function types and pairs, and the novel relationship $T \rightarrow U <: T \rightarrow U$ between standard and linear function types [8]. The key features of subtyping for session types are that $?T.S$ is covariant in $T$ and $S$; $!T.S$ is covariant in $S$ and contravariant in $T$; $&\langle l_i : S_i \rangle_{i \in I}$ is covariant in $I$ and contravariant in each $S_i$; $\oplus \langle l_i : S_i \rangle_{i \in I}$ is contravariant in each $S_i$.

**Definition 1.** Let $\operatorname{Type}$ be the set of types. A relation $R \subseteq \operatorname{Type} \times \operatorname{Type}$ is a type simulation if $(T, U) \in R$ implies the following conditions.

1. If $\mathit{unfold}(T) = ?T_1.S_1$ then $\mathit{unfold}(U) = ?U_1.S_2$ and $\{(T_1, U_1), (S_1, S_2)\} \subseteq R$.
2. If $\mathit{unfold}(T) = !T_1.S_1$ then $\mathit{unfold}(U) = U_1.S_2$ and $\{(U_1, T_1), (S_1, S_2)\} \subseteq R$.
3. If $\mathit{unfold}(T) = &\langle l_i : S_i \rangle_{i \in I}$ then $\mathit{unfold}(U) = &\langle l_j : S'_j \rangle_{j \in J}$, $I \subseteq J$ and $\forall i \in I. (S_i, S'_i) \in R$.
4. If $\mathit{unfold}(T) = \oplus \langle l_i : S_i \rangle_{i \in I}$ then $\mathit{unfold}(U) = \oplus \langle l_j : S'_j \rangle_{j \in J}$, $J \subseteq I$ and $\forall i \in J. (S_i, S'_i) \in R$.
5. If $\mathit{unfold}(T) = \operatorname{End}$ then $\mathit{unfold}(U) = \operatorname{End}$.
6. If $\mathit{unfold}(T) = [S]$ then $\mathit{unfold}(U) = [S']$ and $\{(S, S'), (S', S)\} \subseteq R$.
7. If $\mathit{unfold}(T) = T_1 \otimes T_2$ then $\mathit{unfold}(U) = U_1 \otimes U_2$ and $\{(T_1, U_1), (T_2, U_2)\} \subseteq R$.
8. If $\mathit{unfold}(T) = T_1 \rightarrow T_2$ then $\mathit{unfold}(U) = U_1 \rightarrow U_2$ and $\{(U_1, T_1), (T_2, U_2)\} \subseteq R$.
9. If $\mathit{unfold}(T) = T_1 \rightarrow T_2$ then either: (a) $\mathit{unfold}(U) = T_1 \rightarrow T_2$; or (b) $\mathit{unfold}(U) = U_1 \rightarrow U_2$ and $\{(U_1, T_1), (T_2, U_2)\} \subseteq R$.

The subtyping relation is defined by $T <: U$ if there exists a type simulation $R$ such that $(T, U) \in R$.

Further details, including the proof that subtyping is reflexive and transitive and an algorithm for checking subtyping, can be found in [10]. We define equivalence of types $T$ and $U$ as $T <: U$ and $U <: T$. Henceforth types are understood up to type equivalence, so that, in any mathematical context, types $\mu X \cdot S$ and $S((\mu X \cdot S)/X)$ can be used interchangeably, effectively adopting the equi-recursive approach [19, Chapter 21].
Duality is a central concept in the theory of session types. The function $\overline{S}$, defined in Figure 5, yields the canonical dual of a session type $S$. Previous work [10, 21] defined a duality relation coinductively. Here we just write $S = \overline{S}$ on the understanding that we are always working up to equivalence. Details of the relationship between duality and subtyping can be found in reference [10].

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 $T \Rightarrow U$ as well as standard functions $T \to U$. Each type is classified as either linear or unlimited, as defined in Figure 5. $\text{End}$ is unlimited because we do not explicitly close channels.

**Definition 2.** The relation $\mapsto$ on session types is defined by the following rules (as usual, unfolding if necessary).

\[
\begin{align*}
?T.S & \mapsto S & !T.S & \mapsto S & \&\langle\ldots, l : S_i, \ldots\rangle & \mapsto S & \oplus\langle\ldots, l : S_i, \ldots\rangle & \mapsto S
\end{align*}
\]

**Definition 3.** Let $\text{SType}$ be the set of session types. Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. $\mathbb{N}^\infty$ is a complete lattice. $\text{SType} \to \mathbb{N}^\infty$ is a complete lattice if we define $f \sqsubseteq g \iff \forall S \in \text{SType}. f(S) \leq g(S)$ and take meets and joins pointwise. The bottom function maps everything to 0 and the top function maps everything to $\infty$. We also define $\infty + 1 = \infty$ and $\max(n, \infty) = \infty$, for every $n \in \mathbb{N}^\infty$.

Define the monotonic operator $F$ on $\text{SType} \to \mathbb{N}^\infty$, unfolding as necessary:

\[
\begin{align*}
F(f)(\text{End}) &= 0 \\
F(f)(!T.S) &= 0 \\
F(f)(?T.S) &= 1 + f(S) \\
F(f)(\oplus\langle l_i : S_i \rangle_{i \in I}) &= 0 \\
F(f)(\&\langle l_i : S_i \rangle_{i \in I}) &= 1 + \max\{f(S_i)\}_{i \in I}
\end{align*}
\]

The Knaster-Tarski Theorem gives least and greatest fixed points of $F$ (which turn out to coincide, but we do not need this fact). Let $f$ be the least fixed point of $F$. Finally, define $\text{bound}(S) = \max\{f(S') | S \mapsto S'\}$. 

**Fig. 5.** The dual function on session types, and type classification as linear (lin) or unlimited (un).

**Fig. 6.** Type schemas for constants.
If directed graph with until a fixed point is reached: label node \( \alpha \) for a type system with linear types \[24\], we define a partial operation of addition the interactions that are expected after emptying the buffer. In the usual way specifies, for each endpoint, its peer endpoint and a session type; this describes \( \Delta \) environments is disjoint union. contain an implicit condition that the addition must be defined. Addition of binary operation on environments. Typing rules in which environments are added \( \Gamma \quad \Gamma \rhd e: \quad \Gamma \rhd e: T \quad T <: U \)

\[ \begin{align*}
\text{T-Const, T-Var, T-Chan, T-Sub} & \quad \Gamma \vdash e: S \quad \Gamma \vdash e: U \\
\text{T-Pair, T-Split} & \quad \Gamma \vdash e_1: T \quad \Gamma \vdash e_2: U \\
\text{T-Abs, T-AbsL} & \quad \Gamma \vdash \lambda x.e: T \to U \\
\text{T-Fork} & \quad \Gamma \vdash e: \top \quad \Gamma \vdash \text{fork} e_1 e_2: U \\
\text{T-App, T-Case} & \quad \Gamma \vdash e: \top \quad \Gamma \vdash \text{select} l_i e: T_j
\end{align*} \]

Fig. 7. Typing rules for expressions.

The definition yields an algorithm for calculating \( \text{bound}(S) \). Construct a directed graph with \( \{S' \mid S \rhd S'\} \) as the vertices and \( \rhd \) as the edge relation. Label every \( \text{End}, \top, T, S \) and \( \oplus \{l_i: S_i\}_{i \in I} \) 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 \( \langle l_i: S_i\rangle_{i \in I} \) with \( \max \{n_i\}_{i \in I} \) if every \( S_i \) is labelled with \( n_i \). Finally label any unlabelled nodes with \( \infty \). \( \text{bound}(S) \) 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 (Thm 1).

**Lemma 1.** If \( S \rhd S' \) then \( \text{bound}(S') \leq \text{bound}(S) \).

Type environments are defined by the grammar in Figure 4. The order of environment entries is unimportant: regard an environment as a partial function from variables and channels to types. Write \( \text{dom}(\Gamma) \) for the set of variables and channels in \( \Gamma \) and \( \text{cdom}(\Gamma) \) for the set of channels in \( \Gamma \), and say that \( \text{un}(\Gamma) \) is true of an environment in which all types are unlimited. A channel environment \( \Delta \) specifies, for each endpoint, its peer endpoint and a session type; this describes the interactions that are expected after emptying the buffer. In the usual way for a type system with linear types \[24\], we define a partial operation of addition on environments. Let \( \alpha \) be either a variable \( x \) or channel \( c \). If \( \alpha \not\in \text{dom}(\Gamma) \) then \( \Gamma + \alpha: T = \Gamma, \alpha: T \). If \( \alpha: T \in \Gamma \) and \( \text{un}(\Gamma) \) then \( \Gamma + \alpha: T = \Gamma \). In all other cases, \( \Gamma + \alpha: T \) is undefined. Addition is extended inductively to a partial binary operation on environments. Typing rules in which environments are added contain an implicit condition that the addition must be defined. Addition of \( \Delta \) environments is disjoint union.

Typing of expressions is defined in Figures 6 and 7. The typings in Figure 6 are schematics which can be instantiated for any appropriate types. The schemas for \text{send} and \text{receive} capture the essence of the way in which we use linear type constructors to control the use of channels. We treat \text{send} as a curried function which is given a value and a channel and returns the same channel with the
If $\Gamma \vdash \vec{b}$ matches $S$ is defined (by the rules at the top) then we define $S/\vec{b}$ by the rules at the bottom. The diagram illustrates $S/\vec{b}$.

**Fig. 8.** The matches relation.

**Fig. 9.** Typing rules for configurations.
Figure 8 defines two notions. $\Gamma \vdash \vec{b} \text{ matches } S$ means that the sequence of values $\vec{b}$ (which are typed by $\Gamma$) 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 types. These notions are used to characterise the relationship between the types of endpoints and the contents of their buffers. The following result is important for the proof of Type Preservation (Thm 1).

**Lemma 2.** If $\Gamma \vdash \vec{b}$ matches $S$ then $|\vec{b}| \leq \text{bound}(S)$.

Figure 9 defines typing of configurations. T-Thread begins with a single thread (containing an expression), which must have an unlimited type. T-Buffer types a single buffer, requiring the data to match the type of the channel and to fit within the buffer’s capacity. The environment $\theta$ provides read-only access to the types of all channels, including those in different threads. T-Par combines configurations in parallel, both to build a system from threads and to combine buffers. The condition $\Delta_1 \asymp \Delta_2$ ensures that the session types of peer endpoints are dual once the data in their buffers have been taken into account. If the buffers are empty then this condition reduces to the requirement that if $c$ and $c'$ are peers then $\Gamma_1(c) = \Gamma_2(c')$, familiar from previous work with synchronous communication. T-New allows encapsulation of a system and its buffers. The additional hypothesis $S_3 = S_4$ might be expected, but in fact it is a theorem, following ultimately from the hypothesis $\Delta_1 \asymp \Delta_2$ in T-Par. T-TopLevel states that a top-level program has no free channels and provides the complete environment $\Gamma$ to the $\theta$ parameter.

We complete this section by showing that reduction preserves typability of configurations, and then state an explicit runtime safety theorem. We are only interested in configurations that possess buffers for both ends of a channel; furthermore, these buffers cannot be simultaneously non-empty. Any configuration $C$ can be written, up to $\equiv$, as $(\nu c_1c'_1)\ldots(\nu c_mC_m)(C_1 \parallel \ldots \parallel C_n)$, where each $C_i$ is either a thread $(c)$ or a buffer $c \mapsto (c', n, \vec{b})$. We then say that each $C_i \in C$.

**Definition 4.** A configuration $C$ is well-buffered if whenever $c \mapsto (c', n, \vec{b}) \in C$ we have $c' \mapsto (c', n', \vec{b}') \in C$ and at most one of $\vec{b}$ and $\vec{b}'$ is non-empty.

**Theorem 1 (Type Preservation).** If $\Gamma \vdash_\theta C : \Delta$ and $\Gamma \subseteq \theta$ and $C$ is well-buffered and $C \rightarrow C'$, then $C'$ is well-buffered and there exist $\Gamma'$, $\theta'$ and $\Delta'$ such that $\Gamma' \vdash_\theta C' : \Delta'$ and $\Gamma' \subseteq \theta'$ and $\text{dom}(\Gamma') = \text{dom}(\Gamma)$ and $\text{dom}(\Delta') = \text{dom}(\Delta)$.

**Proof.** (Sketch) By induction on the derivation of $C \rightarrow C'$. There are two main points. When receiving (R-Receive/R-Case), the fact that the data in the buffer match the type of the channel guarantees that the first item in the queue is an appropriate value or label. Sending (R-Send/R-Select) is more complex; we discuss R-Send. The matching and compatibility conditions guarantee that if $c : !T.S$ then $c$’s queue is empty. Also, if $d$ is the peer channel of $c$, $\vec{b}$ its queue and $k$ its capacity, we have $\theta(d)/\vec{b} = !T.S$, which implies that $k \geq \text{bound}(\theta(d)) \geq |\vec{b}| + 1$. So the condition $|\vec{b}| < k$ in R-Send is guaranteed by typing.
Runtime Safety is a version of the usual statement that well-typed systems do not get stuck. It implies that there is no buffer overflow (because that would block send or select), that case always receives an appropriate label (otherwise there would be a blocked case with a non-empty buffer), and that receive never receives a label; it also implies the usual runtime safety property for $\nu$.

**Definition 5.** Let $C$ be a configuration with $C_i \in C$. $C_i$ is blocked if $\not\exists c \mapsto (c', n, \overrightarrow{b}) \in C$ such that $C_i \parallel c \mapsto (c', n, \overrightarrow{b}) \rightarrow$.

**Theorem 2 (Runtime Safety).** Let $\Gamma \vdash C: \Delta$ and $C \rightarrow^* C'$. If $C_i \in C'$ is blocked then one of the following applies: (1) $C_i$ is a buffer; (2) $C_i$ is $\langle v \rangle$; (3) $C_i$ is $\langle E[\text{receive } c_j] \rangle$ and $c_j \mapsto (c'_j, n, \varepsilon) \in C'$; (4) $C_i$ is $\langle E[\text{case } c_j \text{ of } \{l_p : e_p\}_{p \in I}] \rangle$ and $c_j \mapsto (c'_j, n, \varepsilon) \in C'$.

Finally, we observe that the expression accept $n$ a can safely be replaced by accept $(\text{bound}(S))$ a where $a: [S]$ 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 $S \rightsquigarrow S'$; Lemmas 1 and 2, 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.

## 5 Related and Future Work

Fähndrich et al. have used session types in the language Sing#, an extension of C# which has been used to implement the Singularity operating system. We compared our work to theirs, and to work by Neubauer and Thiemann [18], in Sections 1 and 2.

Apart from our own previous work [22, 23], the main formal studies of session types in mainstream language paradigms are by Dezani-Ciancaglini et al. [5, 4] for object-oriented languages. They use a synchronous semantics and do not consider subtyping. The language in [4] has 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.

Yoshida and Vasconcelos [27] 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 and Hole [10], 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 ($\nu c'$) to link an endpoint with its peer. Recent work by Honda et al. [16], 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 [12, 13], Vault [3], and adoption and focus [7] are systems based on the C programming language that allow protocols to be statically enforced
by a compiler. They share our goal, but vary greatly in the techniques used. Cyclone [13] 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 [3] and its extension “Adoption and Focus” [7] 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 [9]. We should also investigate the relationship with other forms of static analysis, including type and effect systems [1]. In the longer term we would like 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 object. A thorough understanding of the functional case provides a good foundation for the object-oriented case.

Acknowledgements. Vasco 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.

References


