Session Types for Inter-Process Communication

Simon Gay[1], Vasco Vasconcelos[2], and António Ravara[3]

[1] Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. Email: `<simon@dcs.gla.ac.uk>`
[2] Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, 1749-016 Lisboa, Portugal. Email: `<vv@di.fc.ul.pt>`
[3] Departamento de Matemática, Instituto Superior Técnico, 1049-001 Lisboa, Portugal. Email: `<amar@math.ist.utl.pt>`

Abstract. We define a language whose type system, incorporating session types, allows complex protocols to be specified by types and verified by static typechecking. Although session types are well understood in the context of the π-calculus, our formulation is based on λ-calculus with side-effecting input/output operations and is different in significant ways. Our typing judgements statically describe dynamic changes in the types of channels, our channel types statically track aliasing, and our function types not only specify argument and result types but also describe changes in channel types. After formalising the syntax, semantics and typing rules of our language, and proving a subject reduction theorem, we outline some possibilities for adding references, objects and concurrency.
Keywords: Session types, static typechecking, semantics, distributed programming, specification of communication protocols.

1 Introduction

Communication in distributed systems is typically structured by protocols, which specify the sequence and form of messages passing over communication channels between agents. In order for correct communication to occur, it is essential that protocols are obeyed.

The theory of *session types* [6, 12] allows the specification of a protocol to be expressed as a type; when a communication channel is created, a session type is associated with it. A session type specifies not only the data types of individual messages, but also the state transitions of the protocol and hence the allowable sequences of messages. By extending the standard methodology of static typechecking, it becomes possible to verify, at compile-time, that an agent using the channel do so in accordance with the protocol.

The theory of session types has been developed in the context of the π-calculus [8, 11], an idealised concurrent programming language which focuses on inter-process communication. Session types have not yet been incorporated into a mainstream programmming language, or even studied theoretically in the context of a standard language paradigm: functional, imperative or object-oriented. Vallecillo *et al.* [13] use session types to add behavioural information to the interfaces of CORBA objects, and use Gay and Hole's [3] theory of subtyping to formalise compatibility and substitutability of components, but they have not attempted to design a complete language.

In the absence of session types, current languages do little to assist the programmer in checking that a protocol has been implemented correctly. Although recent developments in programming languages have increasingly emphasised the benefits of static typechecking, programming with communication channels or streams remains largely untyped. In some respects the situation has deteriorated: Pascal provided static typechecking of file input/output, but the modern use of the stream abstraction for both files and network sockets has resulted in the loss of even this level of support.

The aim of this paper is to study session types in a language based on λ-calculus with side-effecting input/output operations, and to establish that compile-time typechecking of session types could feasibly be added to a mainstream imperative language. Although our language is somewhat idealised, we have attempted to organise it around realistic principles, and in particular to address the key differences between a conventional programming style and the programming notation of the π-calculus.

The structure of the paper is as follows. In Section 2 we explain session types in connection with a progressively more sophisticated server for mathematical operations. Sections 3, 4 and 5 define the syntax, operational semantics and type system of our language. In Section 6 we outline the proof of soundness of our type system. Section 7 concludes, and indicates some possibilities for extending the language with references, objects, and concurrency.

2 Session Types and the Maths Server

2.1 Input, Output and Sequencing Types

First consider a server which provides a single operation: addition of integers. A suitable protocol can be defined as follows.

> The client sends two integers. The server sends an integer which is their sum, then closes the connection.

The corresponding session type, from the server's point of view, is

$$S = ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$$

in which ? means *receive*, ! means *send*, dot (.) is sequencing, and End indicates the end of the session. The type does not correspond precisely to the specification, because it does not state that the server calculates the sum. However, the type captures the parts of the specification which we can reasonably expect to verify statically.

The server communicates with a client on a channel called c; we think of the client engaging in a *session* with the server, using the channel c for communication. We can view c as a TCP/IP socket connection, which can be used for bidirectional communication; in practice it would typically be necessary to extract separate input and output streams from c. For the moment we will ignore the mechanism of establishing the connection.

In our language, which uses ML-style let-bindings (as syntactic sugar for applications of λ-abstractions), the server looks like this:

```
let x = receive c
    y = receive c
in  send x + y on c
```

or more concisely:

```
send ((receive c) + (receive c)) on c
```

In a language with assignable variables (which we have not yet included) it might look like this:

```
var    x : Int
       y : Int
begin
       x := receive c
       y := receive c
       send x + y on c
end
```

Interchanging ? and ! yields the type describing the client side of the protocol[1]:

$$\overline{S} = !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End}$$

and a client implementation uses the server to add two particular integers; the *code* may use x but cannot use the channel c except to close it.

```
send 2 on c
send 3 on c
let x = receive c
in  code
```

An alternative possibility is to interpret the specification "the client sends two integers" as a type in which the client sends a single message consisting of a pair of integers (Honda *et al.* [6] call this "piggybacking"). In this case, the type of the server side would be

$$?(\mathsf{Int} \times \mathsf{Int}).!\mathsf{Int}.\mathsf{End}$$

and the type of the client side would again be obtained by exchanging ? and !.

[1] The duality operator $\overline{\cdot} : S \mapsto \overline{S}$ [12, 6, 3, 13] is an important part of the theory of session types, but we do not need to discuss it in this paper because we only consider clients or servers in isolation.

2.2 Branching Types

Now let us modify the protocol and add a negation operation to the server.

> The client selects one of two commands: add or neg. In the case of add the client then sends two integers and the server replies with an integer which is their sum. In the case of neg the client then sends an integer and the server replies with an integer which is its negation. In either case, the server then closes the connection.

The corresponding session type, for the server side, uses the constructor & (*branch*) to indicate that a choice is offered.

$$S = \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\quad \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.\mathsf{End}\rangle$$

Both services must be implemented. We introduce a case construct:

```
case $c$ of {
   add $\Rightarrow$ send ((receive $c$) $+$ (receive $c$)) on $c$
   neg $\Rightarrow$ send ($-$receive $c$) on $c$ }
```

The type of the client side uses the dual constructor ⊕ (*choice*) to indicate that a choice is made.

$$\overline{S} = \oplus\langle \mathsf{add}\colon !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End},\ \ \mathsf{neg}\colon !\mathsf{Int}.?\mathsf{Int}.\mathsf{End}\rangle$$

A particular client implementation makes a particular choice, for example:

```
addclient                          negclient

select add on $c$                  select neg on $c$
send 2 on $c$                      send 4 on $c$
send 3 on $c$                      let $x$ = receive $c$ in $code$
let $x$ = receive $c$ in $code$
```

Note that the type of the subsequent interaction depends on the label which is chosen. In order for typechecking to be decidable, it is essential that the label add or neg appears as a literal name in the program; labels cannot result from computations.

If we add a square root operation, sqrt, then as well as specifying that the argument and result have type Real, we must allow for the possibility of an error (resulting in the end of the session) if the client asks for the square root of a negative number. This is done by using the ⊕ constructor on the server side, with options ok and error. The complete English description of the protocol is starting to become lengthy, so we will omit it and simply show the type of the server side.

$$\begin{aligned} S = \&\langle &\mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\\ &\mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\\ &\mathsf{sqrt}\colon ?\mathsf{Real}\,.\oplus\langle \mathsf{ok}\colon !\mathsf{Real}.\mathsf{End},\quad \mathsf{error}\colon \mathsf{End}\rangle\rangle \end{aligned}$$

In the type of the client side, the occurrence of & indicates that the client must be prepared for both ok and error responses from the server.

$$\begin{aligned}\overline{S} = \oplus\langle &\mathsf{add}\colon \mathsf{!Int.!Int.?Int.End},\\ &\mathsf{neg}\colon \mathsf{!Int.?Int.End},\\ &\mathsf{sqrt}\colon \mathsf{!Real}\,.\,\&\langle \mathsf{ok}\colon \mathsf{?Real.End},\quad \mathsf{error}\colon \mathsf{End}\rangle\rangle\end{aligned}$$

More realistically we might like the operations of the server to allow both integer and real arguments and return results of the appropriate type. This is supported by the theory of subtyping for session types [3, 5] but we have not yet incorporated it into our present language.

2.3 Establishing a Connection

We have not yet considered the question of how the client and the server reach a state in which they both know about the channel c. In the pi calculus, it is natural to define a complete system consisting of a client and a server in parallel. Previous studies of session types in the π-calculus have suggested two mechanisms for creating a connection. Takeuchi, Kubo and Honda [12] propose a pair of constructs: request c in P for use by clients, and accept c in Q for use by servers. In use, request and accept occur in separate parallel processes, and interact with each other to create a new channel; this channel is bound to the name c in both P and Q. Gay and Hole [3] use the standard π-calculus new construct; the client creates a new channel and sends one end of it to the server along a public channel.

In both cases, the creation and naming of a connection are combined into a single operation. If we want to use session types in an imperative or object-oriented language, it is more realistic for the connection to be a value which is returned by an operator and which can then be assigned to a variable. Furthermore, we have not yet added concurrency to our language, and at this stage we are just considering clients or servers as isolated programs. We therefore use chan S to create a channel with type S, and view this operation as an abstraction of both requesting and accepting network connections. A complete client or server will have the form let $x =$ chan S in $\ldots$; close x

2.4 Function Types

Our maths server is bound to a particular channel, c. If we need the server to answer on different channels, we may abstract the channel c, transforming the maths server into a *function.*

$$\mathsf{fun}\ \mathit{serve}\ c = \mathsf{case}\ c\ \mathsf{of}\ \{\ldots\}$$

Then we may have a server answering on *myChannel*, by applying the function *serve* to the channel *myChannel*, as in *serve myChannel*. The type of our server now reflects, not only the fact that it accepts a channel and returns nothing (that is, the type Unit), but also information on how the function uses the channel:

$$c\colon \&\langle \mathsf{add}\colon \ldots, \mathsf{neg}\colon \ldots, \mathsf{sqrt} \ldots\colon \rangle \triangleright \mathsf{chan}\ c \to \mathsf{Unit} \triangleleft c\colon \mathsf{End}$$

It can also be useful to send functions on channels. For example we could add

$$\mathsf{eval}\colon ?(\triangleright \mathsf{Int} \to \mathsf{Bool} \triangleleft).?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}$$

to the type S, with corresponding server code

$$\mathsf{eval} \Rightarrow \mathsf{send}\ ((\mathsf{receive}\ c)\ (\mathsf{receive}\ c))\ \mathsf{on}\ c$$

and a client which requires a primality test service (perhaps the server has fast hardware):

```
select eval on $c$
send $isPrime$ on $c$
send $bignumber$ on $c$
let $x$ = receive $c$ in $code$
```

If it is possible to send channels on channels (we have not yet transferred this feature from the π-calculus to our language) then the server can release c earlier by accepting a channel on which to conduct the function application:

```
eval: ?(Chan ?( $\triangleright$ Int $\to$ Bool $\triangleleft$ ).?Int.!Bool.End).End
eval $\Rightarrow$ let $d$ = receive $c$ in send ((receive $d$) (receive $d$)) on $d$
```

and the client would create a channel (d, here). Eventually the client uses $\mathsf{close}\ d$ to close the channel, changing its type from End to $\perp$.

```
select eval on $c$
let $d$ = chan ?( $\triangleright$ Int $\to$ Bool $\triangleleft$ ).?Int.!Bool.End
in  send $d$ on $c$
    send $isPrime$ on $d$
    send $bignumber$ on $d$
    let $x$ = receive $d$
    in  close $d$
        $code$
```

2.5 Recursive Types

A more realistic server would allow a session to consist of a sequence of commands and responses. The corresponding type must be defined recursively, and it is useful to include a quit command. Here is the type of the server side:

$$\begin{aligned} S = \&\langle & \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.S, \\ & \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.S, \\ & \mathsf{sqrt}\colon ?\mathsf{Real}\,.\oplus\langle \mathsf{ok}\colon !\mathsf{Real}.S,\quad \mathsf{error}\colon S\rangle, \\ & \mathsf{quit}\colon \mathsf{End}\rangle \end{aligned}$$

The server is now implemented by a recursive function, in which the positions of the recursive calls correspond to the recursive occurrences of S in the type definition. Our type system supports the use of recursive functions with any recursive type and any combination of terminating and recursive branches, and is flexible enough to allow some branches to work through the body of the recursive type more than once if desired.

```
fun serve c =
    case c of {
        add ⇒ send ((receive c) + (receive c)) on c
              serve c
        neg ⇒ send (−(receive c)) on c
              serve c
        sqrt ⇒ let x = receive c
               in  if x < 0 then select error on c
                   else select ok on c
                        send √x on c
                   serve c
        quit ⇒ close c }
```

2.6 Aliasing of Channels

As soon as we separate creation and naming of channels, aliasing becomes an issue. In this program:

```
let x = chan !Int.!Int.End
    y = x
in  send 1 on x
    send 2 on y
```

x and y are aliases for a single underlying channel, and the two send operations reduce the type to End. Clearly our type system must track aliases in order to be able to correctly typecheck programs such as this. Our approach is to introduce indirection into type environments. In this example, chan creates a channel with some identity, c say, and the types of both x and y are Chan c. The session type of c, initially !Int.!Int.End, is recorded separately.

3 Syntax

Most of the syntax of our language has been illustrated in the previous sections; here we define it formally by the grammar in Figure 1.

We define data types D, session types S, channel environments Σ, term types T, values v and terms e. We use channel identifiers $c, \ldots$, term identifiers $x, \ldots$, label identifiers $l, \ldots$, and type variables $X, \ldots$. Evaluation contexts E are used in the definition of the operational semantics; an evaluation context contains a single hole $[\ ]$ and we write $E[e]$ for substitution of term e into the hole. Free

$$
\begin{array}{rl}
D & ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Unit} \\
S & ::= \bot \mid \mathsf{End} \mid ?D.S \mid !D.S \mid \&\langle\, l_i : S_i \,\rangle_{i \in I} \mid \oplus\langle\, l_i : S_i \,\rangle_{i \in I} \mid X \mid \mu X \cdot S \\
 & \quad \mid \Sigma \triangleright T \to T \triangleleft \Sigma \\
\Sigma & ::= c_1 : S_1, \ldots, c_n : S_n \quad \text{(the } c_i \text{ are pairwise distinct)} \\
T & ::= D \mid \mathsf{Chan}\ c \\
v & ::= x \mid \lambda x.e \mid c \mid \mathsf{unit} \mid \mathsf{fix} \mid \mathsf{unfold} \mid \mathsf{receive} \mid \mathsf{send} \mid \mathsf{close} \\
 & \quad \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid -1 \mid \ldots \\
e & ::= v \mid ee \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mid \mathsf{select}\ l\ \mathsf{on}\ e \mid \mathsf{chan}\ S \\
E & ::= [\,] \mid Ee \mid vE \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{case}\ E\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mid \mathsf{select}\ l\ \mathsf{on}\ E
\end{array}
$$

Fig. 1. Syntax of types, terms, and contexts

and bound variables are defined as usual and we work up to α-equivalence; the binding occurrences are x in $\lambda x.e$, and X in $\mu X \cdot S$. Substitution is defined as expected.

We do not allow channels to be transmitted on channels. It should be possible to remove this restriction, as the necessary typing techniques are well-understood in the π-calculus, but we leave it for future work. The type $\bot$ describes a channel which has been fully used and explicitly closed. The type $\mathsf{Chan}\ c$ represents the type of the channel with identity c; the actual session type is associated with c in the typing environment, as will become clear in Section 5. Channel identifiers c are not available in the top-level syntax of terms; they arise only during reduction of terms.

In Section 2 we use two *derived constructors*. A term $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ stands for $(\lambda x.e')e$, and $e; e'$ (implied in our examples by the indentation) is an abbreviation for $(\lambda y.e')e$, provided y does not occur free in e'.

4 Operational Semantics

We define a small step operational semantics, making use of evaluation contexts to simplify the definitions. In order to be able to prove a Subject Reduction theorem, we include channel environments Σ in reductions. This is simply to track the changing types of channels; there are no runtime type checks.

We first define the relation $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma', e'$ by the axiom schemas in Figure 2. We abbreviate an axiom schema $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma, e'$ by $e \longrightarrow_{\mathsf{v}} e'$. The $+$ operation is disjoint union on maps. We then define the relation $\Sigma, e \longrightarrow \Sigma', e'$ by the rule R-Context. Some comments on the reduction rules:

– R-Chan creates a new channel with an arbitrary identity. This models both requesting and accepting a connection.
– R-Case allows any permitted label to be chosen. This use of Σ in order to define reductions is only necessary because we are working with programs

$$\begin{array}{lr}
\mathsf{if\ true\ then}\ e\ \mathsf{else}\ e' \longrightarrow_{\mathsf{v}} e & \text{(R-IfT)} \\
\mathsf{if\ false\ then}\ e\ \mathsf{else}\ e' \longrightarrow_{\mathsf{v}} e' & \text{(R-IfF)} \\
(\lambda x.e)v \longrightarrow_{\mathsf{v}} e\{v/x\} & \text{(R-Beta)} \\
\mathsf{fix}\ v \longrightarrow_{\mathsf{v}} v(\mathsf{fix}\ v) & \text{(R-Fix)} \\
\Sigma + c : \mu X \cdot S, \mathsf{unfold}\ c \longrightarrow_{\mathsf{v}} \Sigma + c : S\{(\mu X \cdot S)/X\}, \mathsf{unit} & \text{(R-Unfold)} \\
\Sigma, \mathsf{chan}\ S \longrightarrow_{\mathsf{v}} \Sigma + c : S, c & \text{(R-Chan)} \\
\Sigma + c : !D.S, \mathsf{send}\ v\ \mathsf{on}\ c \longrightarrow_{\mathsf{v}} \Sigma + c : S, \mathsf{unit} & \text{(R-Send)} \\
\Sigma + c : ?D.S, \mathsf{receive}\ c \longrightarrow_{\mathsf{v}} \Sigma + c : S, v \quad \mathsf{if}\ \mathit{typeof}(v) = D & \text{(R-Receive)} \\
\Sigma + c : \mathsf{End}, \mathsf{close}\ c \longrightarrow_{\mathsf{v}} \Sigma + c : \bot, \mathsf{unit} & \text{(R-Close)} \\
\Sigma + c : \&\langle\ l_i : S_i\ \rangle_{i \in I}, \mathsf{case}\ c\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \longrightarrow_{\mathsf{v}} \Sigma + c : S_j, e_j \quad \mathsf{if}\ j \in I & \text{(R-Case)} \\
\Sigma + c : \oplus\langle\ l_i : S_i\ \rangle_{i \in I}, \mathsf{select}\ l_j\ \mathsf{on}\ c \longrightarrow_{\mathsf{v}} \Sigma + c : S_j, \mathsf{unit} \quad \mathsf{if}\ j \in I & \text{(R-Select)} \\
\dfrac{\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma', e'}{\Sigma, E[e] \longrightarrow \Sigma', E[e']} & \text{(R-Context)}
\end{array}$$

Fig. 2. Reduction rules

which only use one end of a channel. In effect, our Subject Reduction theorem (Section 6) is proved relative to the assumption that the program at the other end of the channel does not introduce any type violations.

– Similarly R-Receive allows receive c to evaluate to any value permitted by the type of c.

5 The Type System

Typing judgements are of the form

$$\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$$

where Γ is a map from variables to types and Σ, Σ' are channel environments as in Section 3. The difference between Σ and Σ' reflects the effect of a term on the types of channels, for example

$$x : \mathsf{Chan}\ c \vdash c : !\mathsf{Int}.\mathsf{End} \triangleright \mathsf{send}\ 2\ \mathsf{on}\ x : \mathsf{Unit} \triangleleft c : \mathsf{End}$$

The assignment of types to values is shown in Figure 3. The values send, receive, close, unfold and fix have multiple types: for example, the type of receive is a type schema representing the set of all types of the form c: $?D.S \triangleright \mathsf{Chan}\ c \rightarrow D \triangleleft c$: S where D is an arbitrary data type, S is an arbitrary session type and c is an arbitrary channel identifier. We write $\mathit{typeof}(v)$ for the set of types assigned to value v. To simplify the theory we explicitly unfold recursive types; an implementation would insert unfold automatically where necessary.

$$
\begin{aligned}
\mathsf{true}, \mathsf{false} &: \mathsf{Bool} \\
0, 1, -1, \ldots &: \mathsf{Int} \\
\mathsf{unit} &: \mathsf{Unit} \\
\mathsf{close} &: c\colon \mathsf{End} \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c\colon \bot \\
\mathsf{receive} &: c\colon ?D.S \triangleright \mathsf{Chan}\ c \to D \triangleleft c\colon S \\
\mathsf{send} &: D \to (c\colon !D.S \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c\colon S) \\
\mathsf{fix} &: \emptyset \triangleright (T \to T) \to T \triangleleft \Sigma \\
\mathsf{unfold} &: c\colon \mu X \cdot S \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c\colon S\{(\mu X \cdot S)/X\} \\
+ &: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}
\end{aligned}
$$

Fig. 3. Types for constants

The typing rules are shown in Figure 4. The rules with no typing judgements in their hypotheses have side conditions of the form $\Sigma \vdash \Gamma$ (reading: Γ is well-formed with respect to Σ), defined by

$$\Sigma \vdash \Gamma \iff x : \mathsf{Chan}\ c \in \Gamma \implies c \in dom(\Sigma)$$

which guarantees that Γ contains no references to non-existent channels. Some comments on the typing rules:

- T-Chan links the identity of a channel to its type in Σ.
- In T-New, a new channel is created with an arbitrary identity.
- In T-Abs, the initial and final channel environments of the function body go directly into the function type. The function itself, being a value, cannot affect channels, hence the empty environments both on the left and on the right.
- In T-App, the final channel environment Σ'' of function e must match the initial environment of the argument e'. The final environment of the argument is spilt into two: Σ_1, that satisfies the channel requirements in the function type; and Σ' that must go (together with the final environment in the function type) into the final environment of the application.
- In T-Case, all branches must produce the same final channel environment. This enables us to know the environment for any code following the case, independently of which branch is chosen at runtime. The same applies to the two branches of the conditional in T-If.

A complete program e is well-typed if there exist T and Σ such that

$$\emptyset \vdash \emptyset \triangleright e : T \triangleleft \Sigma$$

Unclosed channels can be detected by examining Σ for types which are not $\bot$.

$$\Gamma \vdash \Sigma \triangleright v : T \triangleleft \Sigma \quad \text{if } \Sigma \vdash \Gamma \text{ and } T \in \mathit{typeof}(v) \tag{T-Const}$$

$$\Gamma + x : T \vdash \Sigma \triangleright x : T \triangleleft \Sigma \quad \text{if } \Sigma \vdash \Gamma + x : T \tag{T-Var}$$

$$\Gamma \vdash \Sigma + c : S \triangleright c : \mathsf{Chan}\ c \triangleleft \Sigma + c : S \quad \text{if } \Sigma + c : S \vdash \Gamma \tag{T-Chan}$$

$$\Gamma \vdash \Sigma \triangleright \mathsf{chan}\ S : \mathsf{Chan}\ c \triangleleft \Sigma + c : S \quad \text{if } \Sigma \vdash \Gamma \tag{T-New}$$

$$\frac{\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma + \Sigma'' \triangleright e : T \triangleleft \Sigma' + \Sigma''} \tag{T-Weak}$$

$$\frac{\Gamma \vdash \Sigma \triangleright e : \mathsf{Bool} \triangleleft \Sigma' \quad \Gamma \vdash \Sigma' \triangleright e' : T \triangleleft \Sigma'' \quad \Gamma \vdash \Sigma' \triangleright e'' : T \triangleleft \Sigma''}{\Gamma \vdash \Sigma \triangleright \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : T \triangleleft \Sigma''} \tag{T-If}$$

$$\frac{\Gamma + x \colon T \vdash \Sigma \triangleright e : U \triangleleft \Sigma' \quad \emptyset \vdash \Gamma}{\Gamma \vdash \emptyset \triangleright \lambda x.e : (\Sigma \triangleright T \to U \triangleleft \Sigma') \triangleleft \emptyset} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \Sigma \triangleright e : (\Sigma_1 \triangleright T \to U \triangleleft \Sigma_2) \triangleleft \Sigma'' \qquad \Gamma \vdash \Sigma'' \triangleright e' : T \triangleleft \Sigma_1 + \Sigma'}{\Gamma \vdash \Sigma \triangleright ee' : U \triangleleft \Sigma_2 + \Sigma'} \tag{T-App}$$

$$\frac{\Gamma \vdash \Sigma \triangleright e : \mathsf{Chan}\ c \triangleleft \Sigma' + c : \&\langle\ l_i : S_i\ \rangle_{i \in I} \quad \forall i.(\Gamma \vdash \Sigma' + c : S_i \triangleright e_i : T \triangleleft \Sigma'')}{\Gamma \vdash \Sigma \triangleright \mathsf{case}\ e\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} : T \triangleleft \Sigma''} \tag{T-Case}$$

$$\frac{\Gamma \vdash \Sigma \triangleright e : \mathsf{Chan}\ c \triangleleft \Sigma' + c : \oplus\langle\ l_i : S_i\ \rangle_{i \in I}}{\Gamma \vdash \Sigma \triangleright \mathsf{select}\ l_i\ \mathsf{on}\ e : T \triangleleft \Sigma' + c : S_i} \tag{T-Select}$$

Fig. 4. Typing rules

5.1 Example: Typing the Maths Server

We simplify the maths server from Section 2.5, so that its channel type is

$$T = \mu X \cdot \&\langle \mathsf{neg} \colon ?\mathsf{Int}.!\mathsf{Int}.X, \quad \mathsf{quit} \colon \mathsf{End}\rangle$$

and the program, removing some syntactic sugar, is

```
(fix (λf.λx.
    unfold x
    case x of {
        neg ⇒ send (−(receive x)) on x
              f x
        quit ⇒ close x })) (chan T)
```

Writing $F = c : T \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c : \perp$ and setting $\Gamma = f : F,\ x : \mathsf{Chan}\ c$, we apply T-App to the following judgements

$$\Gamma \vdash c : \mathsf{End} \triangleright \mathsf{close} : (c \colon \mathsf{End} \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c \colon \perp) \triangleleft c : \mathsf{End}$$

$$\Gamma \vdash c : \mathsf{End} \triangleright x : \mathsf{Chan}\ c \triangleleft c : \mathsf{End}$$

to obtain:

$$\Gamma \vdash c : \mathsf{End} \triangleright \mathsf{close}\ x : \mathsf{Unit} \triangleleft c : \perp \tag{1}$$

Similarly the types of receive and send, and two applications of T-App, give:

$$\Gamma \vdash c : \mathsf{?Int.!Int}.T \triangleright \mathsf{send}\ (-(\mathsf{receive}\ x))\ \mathsf{on}\ x : \mathsf{Unit} \triangleleft c : T \tag{2}$$

We use T-App to type the recursive call; note that the type F was chosen to make (1) and (3) have the same final channel environment:

$$\Gamma \vdash c : T \triangleright f\ x : \mathsf{Unit} \triangleleft c : \bot \tag{3}$$

Applying the expansion $e;e' \mapsto (\lambda y.e')e$ we can use T-Abs, T-App and T-Weak with (2) and (3) to obtain: (call these steps T-Seq)

$$\Gamma \vdash c : \mathsf{?Int.!Int}.T \triangleright \mathsf{send}\ (-(\mathsf{receive}\ x))\ \mathsf{on}\ x\ ;\ f\ x : \mathsf{Unit} \triangleleft c : \bot \tag{4}$$

and now (4) and (1) are the branches of the case, so we have

$$\Gamma \vdash c : \&\langle \mathsf{neg}\colon \mathsf{?Int.!Int}.T,\ \mathsf{quit}\colon \mathsf{End}\rangle \triangleright \mathsf{case}\ x\ \mathsf{of}\ \{\ldots\} : \mathsf{Unit} \triangleleft c : \bot \tag{5}$$

The typing derivation for fix is completed as follows:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash c : T \triangleright \mathsf{unfold}\ x : \mathsf{Unit} \triangleleft c : \&\langle \mathsf{neg}\colon \mathsf{?Int.!Int}.T,\ \mathsf{quit}\colon \mathsf{End}\rangle \qquad (5)}{\Gamma \vdash c : T \triangleright \mathsf{unfold}\ x\ ;\mathsf{case}\ x\ \mathsf{of}\ \ldots : \mathsf{Unit} \triangleleft c : \bot}\ \text{T-Seq}}{\vdash \emptyset \triangleright \lambda f.\lambda x.\ldots : (\emptyset \triangleright F \to F \triangleleft \emptyset) \triangleleft \emptyset}\ \text{T-Abs (twice)}}{\vdash \emptyset \triangleright \mathsf{fix}\ldots : (c : T \triangleright \mathsf{Chan}\ c \to \mathsf{Unit} \triangleleft c : \bot) \triangleleft \emptyset}\ \text{T-App}, \mathsf{fix}$$

T-New gives $\emptyset \vdash \emptyset \triangleright \mathsf{chan}\ T : \mathsf{Chan}\ c \triangleleft c : T$ and using T-App again we have

$$\emptyset \vdash \emptyset \triangleright (\mathsf{fix}\ldots)\ (\mathsf{chan}\ T) : \mathsf{Unit} \triangleleft c : \bot$$

as the typing of the complete program, showing that a channel is created and eventually closed.

6 Type Safety

We state the key lemmas leading to the Subject Reduction theorem, and sketch their proofs.

Lemma 1 (Typability of Subterms in E). *If $\Gamma \vdash \Sigma \triangleright E[e] : T \triangleleft \Sigma'$ then there exist Σ'' and U such that $\Gamma \vdash \Sigma \triangleright e : U \triangleleft \Sigma''$.*

Proof. By induction on the structure of E. The possible positions of the hole mean that in a derivation of $\Gamma \vdash \Sigma \triangleright E[e] : T \triangleleft \Sigma'$, there is a typing derivation for e at the left of the tree (with the same Σ).

Lemma 2 (Replacement in E). *If*

1. *$\mathcal{D}$ is a typing derivation concluding $\Gamma \vdash \Sigma \triangleright E[e] : T \triangleleft \Sigma'$*
2. *$\mathcal{D}'$ is a subderivation of $\mathcal{D}$ concluding $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma_2$*

3. $\mathcal{D}'$ occurs in $\mathcal{D}$ in the position corresponding to the hole in E
4. $\Gamma \vdash \Sigma_1 \triangleright e' : T \triangleleft \Sigma_2$

then $\Gamma \vdash \Sigma_1 \triangleright E[e'] : T \triangleleft \Sigma'$.

Proof. Replace $\mathcal{D}'$ in $\mathcal{D}$ by a derivation of $\Gamma \vdash \Sigma_1 \triangleright e' : T \triangleleft \Sigma_2$. The structure of E is essential; it means that the typings of e' and $E[e']$ have the same leftmost environment Σ'.

Lemma 3. (Values do not use channels) *If $\Gamma \vdash \Sigma \triangleright v : T \triangleleft \Sigma'$, then $\Sigma' = \Sigma$.*
(Well-formedness) *If $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$, then $\Sigma \vdash \Gamma$ and $\Sigma' \vdash \Gamma$.*
(Narrowing) *If $\Gamma \vdash \Sigma + \Sigma'' \triangleright e : T \triangleleft \Sigma' + \Sigma''$ and $\Sigma \vdash \Gamma$, then $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$.*
(Γ-Weakening) *If $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$ and $\Gamma' \vdash \Sigma$, then $\Gamma + \Gamma' \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$.*

Proof. The first item follows from the typing rules for values; the others require induction on typing derivations.

Lemma 4 (Substitution). *If $\Gamma \vdash \Sigma \triangleright v : T \triangleleft \Sigma$ and $\Gamma + x : T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'$, then $\Gamma \vdash \Sigma \triangleright e\{v/x\} : U \triangleleft \Sigma'$.*

Proof. By induction on the derivation of $\Gamma + x : T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'$.

Our Subject Reduction theorem describes the evolution of the channel environment as a program is executed. The invariance of Σ' during reduction steps reflects the fact that Σ' is the final channel environment of a program.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$ and $\Sigma, e \longrightarrow \Sigma'', e'$, then $\Gamma \vdash \Sigma'' \triangleright e' : T \triangleleft \Sigma'$.*

Proof. By induction on the derivation of $\Sigma, e \longrightarrow \Sigma'', e'$ (which means induction on the structure of evaluation contexts E[ ]), first considering $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma'', e'$ by a simple case analysis.

7 Conclusions

We have transferred the concept of session types from the π-calculus to a language based on the λ-calculus. This is a first step towards the application of session types to the specification and verification of protocols implemented in mainstream programming languages. The main differences between the π-calculus with session types, and our language, are as follows. The operations on channels are now independent terms, rather than prefixes of processes, so we have introduced a new form of typing judgement which describes the effect of a term on the channel environment. We have separated creation and naming of channels, and because this introduces the possibility of aliasing, we represent the types of channels by indirection from the main type environment to the channel environment. We have introduced a new form of function type, which in addition to specifying parameter and result types, describes changes in the channel types of both parameters and free variables.

We have defined a static type system which guarantees that channels are not misused, even in the presence of aliasing, and we have proved this property with respect to a formal operational semantics of the language. We have therefore established a sound basis for the design of a more complete language with session types.

Related Work. In the *Vault* system [2] annotations are added to C programs, in order to describe protocols that a compiler can statically enforce. Similarly to our approach, individual runtime objects are tracked by associating keys (channels, in our terminology) with resources, and function types describe the effect of the function on the keys. Although incorporating a form of selection ($\oplus$) the language of types seems quite simple in comparison with session types, and the annotations to C types can appear awkward.

A somewhat related line of research uses *flow-sensitive type qualifiers* to prove that programs access resources in a disciplined manner. Walker, Crary, and Morrisett [14] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [7] present a general framework comprising a language with primitives for creating and accessing resources, and a type inference algorithm that checks whether programs are resource-safe. Although it might be possible to formulate operations on channels as resource use in these frameworks, our work focuses on a more specific problem: we transfer the concept of session types into a language which is closer to programming practice, and we do it purely with a straightforward type system.

Type and effect systems can be used to prove properties of protocols. Gordon and Jeffrey [4] use one such system to prove progress properties of communication protocols written in π-calculus. Rajamani *et al.*'s *Behave* [1, 9] uses CCS to describe properties of π-calculus programs, verified via a combination of type and model checking. In contrast, our system, while embodying more sophisticated protocols (using branch and select, for example), does not attempt to prove correctness of the contents of messages; only correctness of the types and sequence of messages.

Future Work. Finally, we outline some of the issues involved in extending our language to include a wider range of standard features.

Adding ML-style references and assignment seems straightforward, as we have already defined the infrastructure for typechecking in the presence of aliasing. It is important to dereference types sufficiently: we should have

$$x : \mathsf{Ref\ Chan}\ c,\ y : \mathsf{Chan}\ d \vdash c : T,\ d : T \triangleright x := y : \mathsf{Unit} \triangleleft c : T,\ d : T$$

even though the types of x and y do not match directly.

Our function types have a limitation: we are not able to type functions which use free channel variables. We do not yet know how to generalize the rule T-Abs so that a channel identifier can enter the function type and also remain in the environment. This creates a problem with curried functions, and so we cannot type this program:

```
fun $sendTwice\ x\ y =$
    send 1 on $x$
    send 2 on $y$
```

It might be possible to make this program typable by allowing multi-argument functions (via tuple types, for example). However, there are more serious implications for the use of session types in an object oriented language: it is typical and important for the methods of an object to use variables which, although bound in the object, are free in the method.

If we can extend our type system so that functions can use free channel variables, we would expect a function type to describe the effect of a function on the types of any free channel variables which it uses. This will induce constraints on the sequence of function calls within a program: calling a function might change the channel environment in such a way that the same function cannot be called again immediately. Now consider, in an object-oriented language with session types, an object containing a channel which its methods can use. The session type of the channel would constrain the use of the methods, perhaps imposing a particular sequence of method calls. This opens up the possibility of a connection with type systems for non-uniform objects [10]: maybe a non-uniform object type could be generated by the session type of a hidden channel. If we add objects to our language then it will also be interesting to investigate how the existing theory of subtyping for session types [3] relates to subclasses and inheritance.

Our language has no concurrency constructs, and our examples are individual components of a distributed system. By introducing a parallel composition operator we could describe complete systems, and strengthen our type safety results. This would involve transferring more of the machinery of session types from the π-calculus to our language. An alternative way of introducing concurrency within our present language would be to allow certain functions to execute as separate threads; we could then implement a multi-threaded server, for example. If $f\ c$ starts a new thread then from the caller's point of view, f returns immediately, but the caller cannot use c. So the type of f must be $c : S \triangleright T \rightarrow \mathsf{Unit} \triangleleft c : \bot$. More generally we could introduce a new typing rule for thread-starting (*runnable*, in Java's terminology) abstraction:

$$\frac{\Gamma + x\colon T \vdash \Sigma + \Sigma_1 \triangleright e : \mathsf{Unit} \triangleleft \Sigma_2 + \Sigma \quad \Sigma \vdash \Gamma \quad \Sigma_1 \vdash x\colon T}{\Gamma \vdash \Sigma \triangleright \lambda x.e : (\Sigma_1 \triangleright T \rightarrow \mathsf{Unit} \triangleleft \bot) \triangleleft \Sigma} \quad \text{(T-ABS')}$$

where $\bot$ in the conclusion means Σ_1 with all types replaced by $\bot$.

We would like to allow channels to be first-class values which can be sent and received. It should not be too difficult to transfer the necessary machinery from the theory of session types in the pi calculus. This would allow us to implement delegation of sessions, which is necessary in a multi-threaded server.

Acknowledgements. The research reported in this paper was supported by a Treaty of Windsor grant from the British Council in Portugal and the Portuguese

Council of University Rectors. Work partially supported by the EU FEDER, the EU projects of the proactive initiative FET-Global Computing (Mikado and Profundis), and the Portuguese Fundação para a Ciência e Tecnologia, namely via the project MIMO (POSI/CHS/39789/2001).

References

1. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proceedings, 29th ACM Symposium on Principles of Programming Languages*, pages 45–57. ACM Press, 2002.
2. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the PLDI conference*, pages 59–69. ACM Press, 2001.
3. S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In S. D. Swierstra, editor, *ESOP'99: Proceedings of the European Symposium on Programming Languages and Systems*, number 1576 in Lecture Notes in Computer Science, pages 74–90. Springer-Verlag, 1999.
4. A. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In Elsevier, editor, *17th Conference on the Mathematical Foundations of Programming Semantics*, number 45 in ENTCS, 2001.
5. M. J. Hole and S. J. Gay. Bounded polymorphism in session types, 2002. Submitted for publication.
6. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
7. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings, 29th ACM Symposium on Principles of Programming Languages*, pages 331–342. ACM Press, 2002.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.
9. S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proceedings of the Static Analysis Symposium*, volume 2126 of *LNCS*, pages 375–394. Springer-Verlag, 2001.
10. A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488. Springer-Verlag, 2000.
11. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.
12. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.
13. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures (Foclasa 2002)*, Electronic Notes in Theoretical Computer Science. Elsevier, August 2002.
14. D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.