Session Types for Inter-Process Communication

Simon Gay, Vasco Vasconcelos and António Ravara

UNIVERSITY *of* GLASGOW

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

TR-2003-133
March 2003

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>

March 26, 2003

Abstract

We define a language whose type system, incorporating session types, allows complex protocols to be specified by types and verified by static typechecking. A session type, associated with a communication channel, specifies not only the data types of individual messages, but also the state transitions of a protocol and hence the allowable sequences of messages. 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 extending this work to a concurrent object-oriented language.

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, 7, 16] 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 does so in accordance with the protocol.

The theory of session types has been developed in the context of the π-calculus [9, 15], 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.* [17] 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 mainly-functional language such as an ML dialect. 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. This work is a first step towards incorporating channels and session types into a concurrent object-oriented language.

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. Section 3 presents a more substantial example, the POP3 protocol. Sections 4, 5 and 6 define the syntax, operational semantics and type system of our language. In Section 7 we outline the proof of soundness of our type system. In Section 8 we discuss related work. Section 9 concludes, and indicates some possibilities for extending the language with objects and concurrency. The appendices contain more details of the soundness proofs.

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:

$$\mathsf{send}\ ((\mathsf{receive}\ c) + (\mathsf{receive}\ c))\ \mathsf{on}\ c$$

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.* [7] 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 $\overline{\cdot}: S \mapsto \overline{S}$ [3, 6, 7, 16, 17] 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.

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

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

```
case $c$ of {
   add ⇒ send ((receive $c$) + (receive $c$)) on $c$
   neg ⇒ 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.

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

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.

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

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 π-calculus, it is natural to define a complete system consisting of a client and a server running in parallel. Previous studies of session types in the π-calculus have suggested two mechanisms for creating a connection. Takeuchi, Kubo and Honda [16] propose a pair of constructs: $\mathsf{request}\ c\ \mathsf{in}\ P$ for use by clients, and $\mathsf{accept}\ c\ \mathsf{in}\ Q$ for use by servers. In use, $\mathsf{request}$ 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 a more conventional programming language, it is more realistic for the connection to be a value which is returned by an operator and which can then be bound to a name. 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 $\mathsf{new}\ S$ 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 $\mathsf{let}\ x = \mathsf{new}\ S\ \mathsf{in}\ \ldots ; \mathsf{close}\ x$

2.4 Function Types

Our maths server is bound to a particular channel, c. In order to define recursive behaviour via fix (Sections 2.5 and 6.3) we must abstract the channel c, transforming the maths server into a *function*.

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

The type of our server now reflects, not only the fact that it accepts a channel and returns nothing (that is, the constant unit), but also information on how the function uses the channel:

$$c\colon \&\langle \mathsf{add}\colon \ldots, \mathsf{neg}\colon \ldots, \mathsf{sqrt}\ldots\colon \rangle; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End}$$

This function type is specific to the channel c. In order to achieve true abstraction over channels we would have to modify the type system to allow generalization of the channel identifier. However, this restricted form of function type is sufficient for the definition of recursive functions in Section 2.5. We return to this point in Section 6. It can also be useful to send functions on channels. For example we could add

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

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 $code$
```

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:

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

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 $\sqrt{x}$ on $c$
                   $serve\ c$
        eval ⇒ send (receive $c$)(receive $c$) on $c$
               $serve\ c$
        quit ⇒ close $c$
    }
```

2.6 Input and Output of Channels

The eval method proposed above may compromise the throughput of the server, for it may now become unavailable for long periods (while computing hard predicates), or may even fail (from the point of view of the client), when asked to evaluate a partial function.

A better approach involves delegating predicate evaluation to a different thread, thus releasing the server at an earlier stage. The eval method now creates a predicate-evaluation channel that it sends back to the client.

$$\mathsf{eval}\colon !(!(\mathsf{Int} \to \mathsf{Bool}).!\mathsf{Int}.?\mathsf{Bool}.\mathsf{End}).S$$
$$\mathsf{eval} \Rightarrow \mathsf{send\ new}\ !(\mathsf{Int} \to \mathsf{Bool}).!\mathsf{Int}.?\mathsf{Bool}.\mathsf{End}\ \mathsf{on}\ c$$

In order to establish connection with method eval, we assume that the server comprises a pool of threads capable of evaluating the predicate.

```
loop
  let $d$ = new ?(Int → Bool).?Int.!Bool.End
  in send (receive $c$)(receive $c$) on $d$
    close $d$
```

Our client now requests from the server a channel, d, on which to perform predicate evaluation, and goes on with the code for primality test, as in the previous section, only that this time, the session is conducted on the received channel, d.

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

After sending a channel, no further interaction on the channel is possible. Returning to the server, one might be tempted to write the code for the eval method as:

```
eval ⇒ let $d$ = new ?(Int → Bool).?Int.!Bool.End
       in send $d$ on $c$
         close $d$
```

but then, channel d would be closed twice: in the server and in one of the threads in the pool. Our type system guarantees that there is no further interaction on a sent channel.

2.7 Aliasing of Channels

As soon as we separate creation and naming of channels, aliasing becomes an issue. In the program below, x and y are aliases for a single underlying channel, and the two send operations reduce the type of this channel to End.

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

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, new creates a channel with some identity, say c, and the types of both x and y are Chan c. The state of c, initially !Int.!Int.End, is recorded separately. Section 6.4 contains more details, explaining how a function of two arguments may be typed in the presence or absence of aliasing.

3 The POP3 Protocol as a Session Type

POP3 [10] is one of the standard Internet protocols, and specifies the way in which email messages may be manipulated on a remote mail server. A typical client would be an email application running on a personal computer; a typical server would be an organisation's mail gateway. Figure 1 shows the states of the protocol, the possible messages, and the state transitions caused by messages. The main states of the protocol are START, from which the server simply sends a welcoming message; AUTHORIZATION, from which the client must authenticate itself by means of a user name (command USER) and password (command PASS); and TRANSACTION, from which the client may issue a range of commands. Only the STAT (status), RETR (retrieve) and QUIT commands are shown in the diagram. The other commands are similar.

The protocol specifies that all messages are strings, based on the underlying assumption that communication channels transmit sequences of characters. However, higher level type information can be extracted from the specified format of these strings. For example, the response to a STAT command contains a pair of integers (representing the number of messages and the total size of the mailbox). Every response from the server is prefixed by either +OK or -ERR, and these strings can be viewed as labels in a branch type. In most cases the server also sends a string containing additional information. Similarly the protocol specifies the set of commands available to the client, which can be incorporated into another branch type. Some client commands are structured, for example USER and PASS which carry strings and RETR which carries an integer. In some cases there are alternative possibilities for imposing type structure on the messages. The server's OK responses to the STAT and RETR commands illustrate two views of the transmission of two pieces of information: as a pair, or as a sequence.

Figure 2 shows the definitions of session types corresponding to the specification of the POP3 protocol. The named types S, A and T correspond to the START, AUTHORIZATION and TRANSACTION states; the definitions of these types are mutually recursive. The types describe the protocol from the server's point of view. Notice the nested alternation of & and $\oplus$, corresponding to the alternation of choices made by the client and the server, and the occurrences of $\oplus$ with only one option, specifying the server's response to commands which always succeed.

4 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 3.

We define data types D, session types S, channel environments Σ, term types T, values v and terms e. We use channel variables $c, \ldots$, term variables $x, \ldots$, labels $l, \ldots$, 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 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.

The type $\mathsf{Chan}\ c$ represents the type of the channel with identity c; the actual session type associated with c is recorded in a typing environment Γ, as will become clear in Section 6.

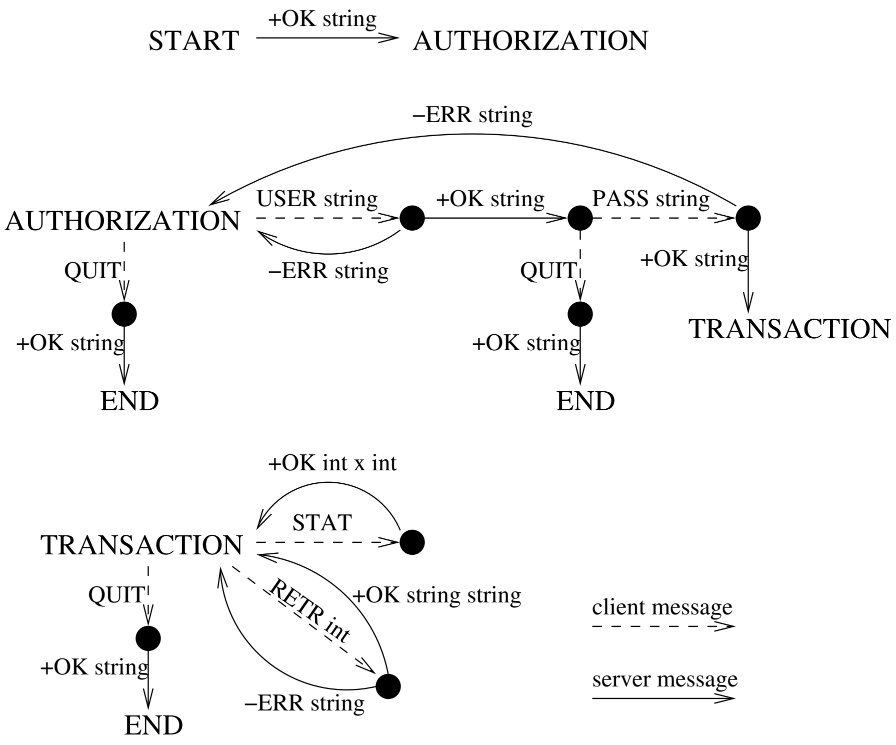

Figure 1: States and transitions of the POP3 protocol

$$
\begin{aligned}
\mathsf{S} &= \oplus\langle \mathsf{ok}\colon !\mathsf{String}\,.\,\mathsf{A}\rangle\\
\mathsf{A} &= \&\langle \mathsf{quit}\colon \oplus\langle \mathsf{ok}\colon !\mathsf{String}\,.\,\mathsf{End}\rangle,\\
&\qquad \mathsf{user}\colon ?\mathsf{String}\,.\,\oplus\langle \mathsf{error}\colon !\mathsf{String}\,.\,\mathsf{A},\\
&\qquad\qquad \mathsf{ok}\colon !\mathsf{String}\,.\,\&\langle \mathsf{quit}\colon \oplus\langle \mathsf{ok}\colon !\mathsf{String}\,.\,\mathsf{End}\rangle,\\
&\qquad\qquad\qquad \mathsf{pass}\colon ?\mathsf{String}\,.\,\oplus\langle \mathsf{error}\colon !\mathsf{String}\,.\,\mathsf{A},\quad \mathsf{ok}\colon !\mathsf{String}\,.\,\mathsf{T}\rangle\rangle\rangle\rangle\\
\mathsf{T} &= \&\langle \mathsf{stat}\colon \oplus\langle \mathsf{ok}\colon !(\mathsf{Int}\times\mathsf{Int})\,.\,\mathsf{T}\rangle,\\
&\qquad \mathsf{retr}\colon ?\mathsf{Int}\,.\,\oplus\langle \mathsf{ok}\colon !\mathsf{String}\,.\,!\mathsf{String}\,.\,\mathsf{T},\quad \mathsf{error}\colon !\mathsf{String}\,.\,\mathsf{T}\rangle,\\
&\qquad \mathsf{quit}\colon \oplus\langle \mathsf{ok}\colon !\mathsf{String}\,.\,\mathsf{End}\rangle\rangle
\end{aligned}
$$

Figure 2: Types for the POP3 protocol

$$
\begin{aligned}
D &::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Unit} \mid \Sigma; T \rightarrow T; \Sigma \\
S &::= ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \&\langle\, l_i : S_i \,\rangle_{i \in I} \mid \oplus \langle\, l_i : S_i \,\rangle_{i \in I} \mid \mathsf{End} \mid \\
&\qquad X \mid \mu X \cdot S \\
\Sigma &::= \emptyset \mid \Sigma + c\colon S \quad (c\colon S' \text{ not in } \Sigma) \\
T &::= D \mid \mathsf{Chan}\ c \\
v &::= x \mid \lambda x.e \mid c \mid \mathsf{unit} \mid \mathsf{fix} \mid \mathsf{unfold} \mid \mathsf{receive} \mid \mathsf{receive}\ v \mid \mathsf{send} \mid \mathsf{close} \mid \\
&\qquad \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid -1 \mid \ldots \\
e &::= v \mid ee \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mid \mathsf{select}\ l\ \mathsf{on}\ e \mid \mathsf{new}\ S \\
E &::= [\,] \mid Ee \mid vE \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{case}\ E\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mid \mathsf{select}\ l\ \mathsf{on}\ E
\end{aligned}
$$

Figure 3: Syntax of types, terms, and contexts

Channel identifiers c are not available in the top-level syntax of terms; they arise only during reduction of terms. We often write $T \rightarrow U$ as an abbreviation for $\emptyset; T \rightarrow U; \emptyset$.

In Section 2 we use several *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'. Finally, a term $\mathsf{loop}\ e$ stands for $\mathsf{fix}\,(\lambda f.(e; f))\,\mathsf{unit}$, provided f does not occur free in e.

5 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 on typable programs.

We refer to a pair composed of a channel environment Σ and a term e as a *configuration*. The use of configurations gives our semantics an imperative flavour: the channel environment is analogous to state information. The reduction relation $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma', e'$ is defined on configurations, by the axiom schemas in Figure 4, where channel environments Σ are considered up to reordering of their components $c\colon S$. We abbreviate an axiom schema $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma, e'$ by $e \longrightarrow_{\mathsf{v}} e'$. We then define the relation $\Sigma, e \longrightarrow \Sigma', e'$ by the rule R-Context.

Some comments on the reduction rules:

- R-Fix introduces an abstraction around ($\mathsf{fix}\ v$) as v must applied to a value [19].
- R-New creates a new channel with an arbitrary identity. This models both requesting and accepting a connection.
- Similarly R-ReceiveD allows $\mathsf{receive}\ c$ to evaluate to any value permitted by the type of c.
- R-SendS ensures that no further interaction is possible at a transmited channel, by removing it from the channel environment.

$$\text{if true then } e \text{ else } e' \longrightarrow_{\mathsf{v}} e \qquad \text{(R-IfT)}$$

$$\text{if false then } e \text{ else } e' \longrightarrow_{\mathsf{v}} e' \qquad \text{(R-IfF)}$$

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

$$\mathsf{fix}\ v \longrightarrow_{\mathsf{v}} v(\lambda x.\mathsf{fix}\, v\, x) \quad \text{for some } x \text{ not free in } v \qquad \text{(R-Fix)}$$

$$\Sigma, \mathsf{new}\ S \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, c \qquad \text{(R-New)}$$

$$\Sigma + c\colon !D.S, \mathsf{send}\ v \text{ on } c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, \mathsf{unit} \qquad \text{(R-SendD)}$$

$$\Sigma + c\colon !S'.S + d\colon S', \mathsf{send}\ d \text{ on } c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, \mathsf{unit} \qquad \text{(R-SendS)}$$

$$\Sigma + c\colon ?D.S, \mathsf{receive}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, v \quad \text{for some closed } v \text{ of type } D \qquad \text{(R-ReceiveD)}$$

$$\Sigma + c\colon ?S'.S, \mathsf{receive}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S + d\colon S', d \qquad \text{(R-ReceiveS)}$$

$$\Sigma + c\colon \mathsf{End}, \mathsf{close}\ c \longrightarrow_{\mathsf{v}} \Sigma, \mathsf{unit} \qquad \text{(R-Close)}$$

$$\Sigma + c\colon \&\langle\, l_i : S_i \,\rangle_{i\in I}, \mathsf{case}\ c \text{ of } \{l_i \Rightarrow e_i\}_{i\in I} \longrightarrow_{\mathsf{v}} \Sigma + c\colon S_j, e_j \quad \text{for some } j \in I \qquad \text{(R-Case)}$$

$$\Sigma + c\colon \oplus\langle\, l_i : S_i \,\rangle_{i\in I}, \mathsf{select}\ l_j \text{ on } c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S_j, \mathsf{unit} \quad \text{if } j \in I \qquad \text{(R-Select)}$$

$$\Sigma + c\colon \mu X \cdot S, \mathsf{unfold}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S\{(\mu X \cdot S)/X\}, \mathsf{unit} \qquad \text{(R-Unfold)}$$

$$\frac{\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma', e'}{\Sigma, E[e] \longrightarrow \Sigma', E[e']} \qquad \text{(R-Context)}$$

Figure 4: Reduction rules

- 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 which only use one end of a channel. In effect, our Subject Reduction theorem (Section 7) is proved relative to the assumption that the program at the other end of the channel does not introduce any type violations.

6 Typing

6.1 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 4. The difference between Σ and Σ' reflects the effect of a term on the types of channels, for example

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

The assignment of types to constants is shown in Figure 5. 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 $c\colon ?D.S; \mathsf{Chan}\ c \to D; c\colon S$ or of the form $c\colon ?S'.S; \mathsf{Chan}\ c \to \mathsf{Chan}\ d; c\colon S + d\colon S'$ where D is an arbitrary data type, S, S' are arbitrary session types, and c, d are arbitrary channel identifiers. We write *typeof*(v) for the set of types assigned to constant v. The type of fix allows recursive functions to create new

$$
\begin{aligned}
\mathsf{true}, \mathsf{false} &\colon \mathsf{Bool}\\
0, 1, -1, \ldots &\colon \mathsf{Int}\\
\mathsf{unit} &\colon \mathsf{Unit}\\
\mathsf{close} &\colon c\colon \mathsf{End}; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; \emptyset\\
\mathsf{receive} &\colon c\colon ?D.S; \mathsf{Chan}\ c \rightarrow D; c\colon S\\
\mathsf{receive} &\colon c\colon ?S'.S; \mathsf{Chan}\ c \rightarrow \mathsf{Chan}\ d; c\colon S + d\colon S'\\
\mathsf{send} &\colon D \rightarrow (c\colon !D.S; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; c\colon S)\\
\mathsf{send} &\colon \mathsf{Chan}\ d \rightarrow (c\colon !S'.S + d\colon S'; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; c\colon S)\\
\mathsf{fix} &\colon (T \rightarrow T) \rightarrow T \qquad \text{where } T = \Sigma_1; T_1 \rightarrow T_2; \Sigma_2\\
\mathsf{unfold} &\colon c\colon \mu X \cdot S; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; c\colon S\{(\mu X \cdot S)/X\}
\end{aligned}
$$

Figure 5: Types for constants

channels, or to use channels supplied as arguments, but not to use global channels. To simplify the theory we explicitly unfold recursive types; an implementation would insert unfold automatically where necessary.

The typing rules are shown in Figure 6. Some comments:

- The type of a channel c is Chan c. The state (or current type) of c, if needed, may be added to the channel environment via the T-Weak rule.

- 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 split 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.

  Also, T and Σ_1 in the function type and the argument typing must match exactly, including equality of channel identifiers. Although we are able to construct the functions necessary to express recursive behaviour—the main concern of the present paper—a realistic programming language would require a more general form of function type in which channel identifiers are generalized. We expect that this would be similar to standard formulations of polymorphism in typed λ-calculus, with rules for abstraction and instantiation of channel identifiers.

- 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.

$$\frac{T \in \mathit{typeof}(v)}{\Gamma \vdash \emptyset \triangleright v : T \triangleleft \emptyset} \quad \text{(T-Const)}$$

$$\Gamma + x \colon T \vdash \emptyset \triangleright x : T \triangleleft \emptyset \quad \text{(T-Var)}$$

$$\Gamma \vdash \emptyset \triangleright c : \mathsf{Chan}\ c \triangleleft \emptyset \quad \text{(T-Chan)}$$

$$\Gamma \vdash \emptyset \triangleright \mathsf{new}\ S : \mathsf{Chan}\ c \triangleleft c \colon S \quad \text{(T-New)}$$

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

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

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

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

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

$$\frac{\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma + c \colon S \triangleright e : T \triangleleft \Sigma' + c \colon S} \quad \text{(T-Weak)}$$

Figure 6: Typing rules

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

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

Any entry in Σ denotes an unclosed channel.

6.2 Derived rules

We have used a number of useful term abbreviations in Section 2. Figure 7 gathers the typing rules for these abbreviations.

Recall from Section 4 that term $e; e'$ is an abbreviation for $(\lambda y.e')e$, provided y does not occur free in e'. The corresponding rule, T-Seq, can be derived as follows, where rule T-Weak is applied as many times as there are entries in Σ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \Sigma' \triangleright e' : U \triangleleft \Sigma''}{\Gamma + y \colon T \vdash \Sigma' \triangleright e' : U \triangleleft \Sigma''}\ \text{Lemma 4}}{\Gamma \vdash \emptyset \triangleright \lambda y.e' : (\Sigma'; T \to U; \Sigma'') \triangleleft \emptyset}\ \text{T-Abs}}{\Gamma \vdash \Sigma \triangleright \lambda y.e' : (\Sigma'; T \to U; \Sigma'') \triangleleft \Sigma}\ \text{T-Weak}^* \qquad \Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright (\lambda y.e')e : U \triangleleft \Sigma''}\ \text{T-App}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : T \lhd \Sigma' \quad \Gamma \vdash \Sigma' \rhd e' : U \lhd \Sigma''}{\Gamma \vdash \Sigma \rhd e; e' : U \lhd \Sigma''} \tag{T-Seq}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : T \lhd \Sigma' \quad \Gamma' + x\colon T \vdash \Sigma'' \rhd e' : U \lhd \Sigma'}{\Gamma \vdash \Sigma \rhd \mathsf{let}\ x = e\ \mathsf{in}\ e' : U \lhd \Gamma''} \tag{T-Let}$$

$$\frac{\Gamma \vdash \emptyset \rhd e : T \lhd \emptyset}{\Gamma \vdash \Sigma \rhd \mathsf{loop}\ e : \mathsf{Unit} \lhd \Sigma'} \tag{T-Loop}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : \mathsf{Chan}\ c \lhd \Sigma' + c\colon ?D.S}{\Gamma \vdash \Sigma \rhd \mathsf{receive}\ e : D \lhd \Sigma' + c\colon S} \tag{T-ReceiveD}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : D \lhd \Sigma' \qquad \Gamma \vdash \Sigma' \rhd e' : \mathsf{Chan}\ c \lhd \Sigma'' + c\colon !D.S}{\Gamma \vdash \Sigma \rhd \mathsf{send}\ e\ \mathsf{on}\ e' : \mathsf{Unit} \lhd \Sigma'' + c\colon S} \tag{T-SendD}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : \mathsf{Chan}\ c \lhd \Sigma' + c\colon ?S'.S}{\Gamma \vdash \Sigma \rhd \mathsf{receive}\ e : \mathsf{Chan}\ d \lhd \Sigma' + c\colon S + d\colon S'} \tag{T-ReceiveS}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : \mathsf{Chan}\ d \lhd \Sigma' \qquad \Gamma \vdash \Sigma' \rhd e' : \mathsf{Chan}\ c \lhd \Sigma'' + c\colon !S'.S + d\colon S}{\Gamma \vdash \Sigma \rhd \mathsf{send}\ e\ \mathsf{on}\ e' : \mathsf{Unit} \lhd \Sigma'' + c\colon S} \tag{T-SendS}$$

$$\frac{\Gamma \vdash \Sigma \rhd e : \mathsf{Chan}\ c \lhd \Sigma' + c\colon \mathsf{End}}{\Gamma \vdash \Sigma \rhd \mathsf{close}\ e : \mathsf{Unit} \lhd \Sigma'} \tag{T-Close}$$

Figure 7: Derived typing rules

A similar derivation yields rule T-Let for term let $x = e$ in e', which, recall, stands for $(\lambda x.e')e$. Rule T-SendS is also easy to derive. Let $T = c\colon !S'.S + d\colon S'; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon S$; then the type of send is $\mathsf{Chan}\ d \to T$.

$$\dfrac{\dfrac{}{\Gamma \vdash \Sigma \rhd \mathsf{send} : \mathsf{Chan}\ d \to T \lhd \Sigma}\ \text{T-Const, T-Weak}^* \qquad \Gamma \vdash \Sigma \rhd e : \mathsf{Chan}\ d \lhd \Sigma'}{\Gamma \vdash \Sigma \rhd \mathsf{send}\ e : T \lhd \Sigma'}\ \text{T-App} \tag{1}$$

$$\dfrac{(1) \qquad \Gamma \vdash \Sigma' \rhd e' : \mathsf{Chan}\ c \lhd \Sigma'' + c\colon !S'.S + d\colon S'}{\Gamma \vdash \Sigma \rhd \mathsf{send}\ e\ \mathsf{on}\ e' : \mathsf{Unit} \lhd \Sigma'' + c\colon S}\ \text{T-App}$$

Rule T-Loop delivers a term that asks for any channel environment Σ and delivers a possibly unrelated channel environment Σ'. Intuitively, this is justified by the fact that loop does not terminate. The derivation below illustrates this point. For the type of fix we take $(F \to F) \to F$, with F of the form $\Sigma; \mathsf{Unit} \to \mathsf{Unit}; \Sigma'$; this is where we choose the desired initial and final channel environments.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \emptyset \rhd e : T \lhd \emptyset}{\Gamma + f\colon F \vdash \emptyset \rhd e : T \lhd \emptyset}\ \text{Lemma 4} \qquad \dfrac{}{\Gamma + f\colon F \vdash \emptyset \rhd f : F \lhd \emptyset}\ \text{T-Var}}{\Gamma + f\colon F \vdash \emptyset \rhd e; f : F \lhd \emptyset}\ \text{T-Seq}}{\Gamma \vdash \emptyset \rhd \lambda f.e; f : F \to F \lhd \emptyset}\ \text{T-Abs} \tag{2}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \emptyset \triangleright \mathsf{fix} : (F \to F) \to F \triangleleft \emptyset} \quad (2)}{\Gamma \vdash \Sigma \triangleright \mathsf{fix}(\lambda f.e; f) : F \triangleleft \Sigma}\ \text{T-App, T-Weak}^* \qquad \dfrac{}{\Gamma \vdash \Sigma \triangleright \mathsf{unit} : \mathsf{Unit} \triangleleft \Sigma}}{\Gamma \vdash \Sigma \triangleright \mathsf{loop}\ e : \mathsf{Unit} \triangleleft \Sigma'}\ \text{T-App}$$

The first part of the derivation also shows why the initial and final channel environments of the loop body must be empty: because the type of fix requires it. For a loop of type Int, one would pick for F the type $\Sigma; \mathsf{Int} \to \mathsf{Int}; \Sigma'$ and apply $\mathsf{fix}(\lambda f.e; f)$ to some integer constant.

6.3 Typing the Maths Server

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

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

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 })) (new S)
```

Writing $F = c\colon S; \mathsf{Chan}\ c \to \mathsf{Unit}; \emptyset$, and setting $\Gamma = f\colon F,\ x\colon \mathsf{Chan}\ c$, we start with the derived T-Close rule (Figure 7).

$$\frac{\Gamma \vdash c\colon \mathsf{End} \triangleright x : \mathsf{Chan}\ c \triangleleft c\colon \mathsf{End}}{\Gamma \vdash c\colon \mathsf{End} \triangleright \mathsf{close}\ x : \mathsf{Unit} \triangleleft \emptyset}\ \text{T-Close} \tag{3}$$

Similarly, the rules T-ReceiveD, T-App (with constant $-$ of type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$), and T-SendD give:

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

We use T-App to type the recursive call; note that the type F was chosen to make (3) and (5) have the same final channel environment:

$$\Gamma \vdash c\colon S \triangleright f\ x : \mathsf{Unit} \triangleleft \emptyset \tag{5}$$

We then apply rule T-Seq to (4) and (5) to obtain:

$$\Gamma \vdash c\colon ?\mathsf{Int}.!\mathsf{Int}.S \triangleright \mathsf{send}\ (-(\mathsf{receive}\ x))\ \mathsf{on}\ x\ ; f\ x : \mathsf{Unit} \triangleleft \emptyset \tag{6}$$

and now (6) and (3) are the branches of the case, so we have

$$\Gamma \vdash c\colon \&\langle \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.S,\ \mathsf{quit}\colon \mathsf{End}\rangle \triangleright \mathsf{case}\ x\ \mathsf{of}\ \{\ldots\} : \mathsf{Unit} \triangleleft \emptyset \tag{7}$$

The typing derivation for fix is completed as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash c\colon S \triangleright \mathsf{unfold}\ x : \mathsf{Unit} \triangleleft c\colon \&\langle \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.S,\ \mathsf{quit}\colon \mathsf{End}\rangle \quad (7)}{\Gamma \vdash c\colon S \triangleright \mathsf{unfold}\ x\,;\mathsf{case}\ x\ \mathsf{of}\ \ldots : \mathsf{Unit} \triangleleft \emptyset}\ \text{T-Seq}}{\vdash \emptyset \triangleright \lambda f.\lambda x.\ldots : F \to F \triangleleft \emptyset}\ \text{T-Abs (twice)}}{\vdash \emptyset \triangleright \mathsf{fix}\ldots : F \triangleleft \emptyset}\ \text{T-App}, \mathsf{fix}$$

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

$$\vdash \emptyset \triangleright (\mathsf{fix}\ldots)\ (\mathsf{new}\ S) : \mathsf{Unit} \triangleleft \emptyset$$

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

6.4 Typing Channel Aliasing

The expansion of the let-term in Section 2.7 is $(\lambda x.Ax)(\mathsf{new}\ S)$ where S is $!\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$ and A is $\lambda y.\mathsf{send}\ 1\ \mathsf{on}\ x; \mathsf{send}\ 2\ \mathsf{on}\ y$. A suitable derivation for A ends as follows, where we have chosen x and y to share the same channel c.

$$\dfrac{\dfrac{\ldots \qquad\qquad \ldots}{x\colon \mathsf{Chan}\ c + y\colon \mathsf{Chan}\ c \vdash c\colon S \triangleright \mathsf{send}\ 1\ \mathsf{on}\ x; \mathsf{send}\ 2\ \mathsf{on}\ y : \mathsf{Unit} \triangleleft c\colon \mathsf{End}}\ \text{T-Seq}}{x\colon \mathsf{Chan}\ c \vdash \emptyset \triangleright A : (c\colon S; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End}) \triangleleft \emptyset}\ \text{T-Abs} \qquad (8)$$

We then apply A to x, thus aliasing x and y, extracting the (initial and final) state of channel c, from A's function type, into the sequent.

$$\dfrac{(8) \qquad \dfrac{}{x\colon \mathsf{Chan}\ c \vdash \emptyset \triangleright x : \mathsf{Chan}\ c \triangleleft \emptyset}\ \text{T-Var}}{x\colon \mathsf{Chan}\ c \vdash c\colon S \triangleright Ax : \mathsf{Unit} \triangleleft c\colon \mathsf{End}}\ \text{T-App}$$

Notice that A is typed in such a way that it can only be applied to a term of type $\mathsf{Chan}\ c$, where c is linked both to both variables x and y.

If the aliasing of x and y in A is not sought, then we must type A as follows, where S' is now $!\mathsf{Int}.\mathsf{End}$.

$$\dfrac{\dfrac{\ldots \qquad\qquad \ldots}{x\colon \mathsf{Chan}\ c + y\colon \mathsf{Chan}\ d \vdash c\colon S + d\colon S \triangleright \mathsf{send} \cdots : \mathsf{Unit} \triangleleft c\colon S' + d\colon S'}\ \text{T-Seq}}{x\colon \mathsf{Chan}\ c \vdash \emptyset \triangleright A : (c\colon S + d\colon S; \mathsf{Chan}\ d \to \mathsf{Unit}; c\colon S' + d\colon S') \triangleleft \emptyset}\ \text{T-Abs}$$

Function A must now be applied to an expression of type $\mathsf{Chan}\ d$, different from x's type $\mathsf{Chan}\ c$.

7 Type Safety

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

Lemma 1 (Values do not use channels) *If v is a value and $\Gamma \vdash \Sigma \triangleright v : T \triangleleft \Sigma'$ then $\Sigma' = \Sigma$.*

Proof: A derivation of $\Gamma \vdash \Sigma \rhd v : T \lhd \Sigma'$ involves an application of T-Const, T-Abs, T-Var or T-New, possibly followed by applications of T-Weak which preserve equality of the left and right environments. $\square$

Lemmas 2 and 3 are similar to lemmas used by Wright and Felleisen [19].

Lemma 2 (Typability of Subterms in E) *If $\mathcal{D}$ is a typing derivation concluding $\Gamma \vdash \Sigma \rhd E[e] : T \lhd \Sigma'$ then there exist Σ'' and U such that $\mathcal{D}$ has a subderivation $\mathcal{D}'$ concluding $\Gamma \vdash \Sigma \rhd e : U \lhd \Sigma''$.*

Proof: By induction on the structure of $E[\,]$. The possible positions of the hole mean that in a subderivation of $\mathcal{D}$ which types e, the leftmost environment is equal to Σ. For example, consider the case $E[e] = v(F[e])$. The derivation $\mathcal{D}$ has the form

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash \Sigma_1 \rhd v : (\Sigma_3; V \to T; \Sigma_3) \lhd \Sigma''\end{array} \qquad \mathcal{D}_1 \left\{ \begin{array}{c}\vdots\\ \Gamma \vdash \Sigma'' \rhd F[e] : V \lhd \Sigma_2 + \Sigma_3\end{array}\right.}{\Gamma \vdash \Sigma_1 \rhd v(F[e]) : T \lhd \Sigma_2 + \Sigma_4}\ \textsc{T-App}}{\Gamma \vdash \Sigma \rhd v(F[e]) : T \lhd \Sigma'}\ \textsc{T-Weak}^*$$

By Lemma 1, $\Sigma'' = \Sigma_1$. By the induction hypothesis applied to $\mathcal{D}_1$, there exist Σ_5 and U such that $\mathcal{D}_1$ has a subderivation $\mathcal{D}'$ concluding $\Gamma \vdash \Sigma_1 \rhd e : U \lhd \Sigma_5$. $\mathcal{D}'$ is the desired subderivation of $\mathcal{D}$. $\square$

Lemma 3 (Replacement in E) *If*

1. *$\mathcal{D}$ is a typing derivation concluding $\Gamma \vdash \Sigma \rhd E[e] : T \lhd \Sigma'$*
2. *$\mathcal{D}'$ is a subderivation of $\mathcal{D}$ concluding $\Gamma \vdash \Sigma \rhd e : T \lhd \Sigma_2$*
3. *$\mathcal{D}'$ occurs in $\mathcal{D}$ in the position corresponding to the hole in E*
4. *$\Gamma \vdash \Sigma_1 \rhd e' : T \lhd \Sigma_2$*

then $\Gamma \vdash \Sigma_1 \rhd E[e'] : T \lhd \Sigma'$.

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

Lemma 4 (Γ-Weakening) *If $\Gamma \vdash \Sigma \rhd e : T \lhd \Sigma'$, then $\Gamma + x\colon T \vdash \Sigma \rhd e : T \lhd \Sigma'$.*

Proof: By induction on the derivation of $\Gamma \vdash \Sigma \rhd e : T \lhd \Sigma'$. $\square$

Lemma 5 (Narrowing for values) *If $\Gamma \vdash \Sigma \rhd v : T \lhd \Sigma'$, then $\Gamma \vdash \emptyset \rhd v : T \lhd \emptyset$.*

Proof: The structure of a derivation of $\Gamma \vdash \Sigma \rhd v : T \lhd \Sigma'$ is as described in the proof of Lemma 1. Removing the applications of T-Weak yields a derivation of $\Gamma \vdash \emptyset \rhd v : T \lhd \emptyset$. $\square$

Lemma 6 (Substitution) *If $\Gamma + x\colon T \vdash \Sigma \rhd e : U \lhd \Sigma'$ and $\Gamma \vdash \emptyset \rhd v : T \lhd \emptyset$ then $\Gamma \vdash \Sigma \rhd e\{v/x\} : U \lhd \Sigma'$.*

Proof: By induction on the derivation of $\Gamma + x\colon T \vdash \Sigma \rhd e : U \lhd \Sigma'$. Details of the proof can be found in Appendix A. $\square$

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 7 (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. Details of the proof can be found in Appendix B. □

Definition 1 *A configuration* (Σ, e) *is* faulty *when* e *is not a value and* $\Sigma, e \not\to$. *We can also give an extensional definition of faulty environments, by looking at the reduction rules in Figure 4. Below are some examples of faulty environments; the remaining involve* unfold, send, close, *and* case.

$$\begin{array}{rl}
(\_, \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e') & v \neq \mathsf{true}, \mathsf{false} \\
(\_, \mathsf{receive}\ v) & v \textit{ not a channel} \\
(\Sigma, \mathsf{receive}\ c) & c\colon S \textit{ not in } \Sigma \\
(\Sigma + c\colon S, \mathsf{receive}\ c) & S \neq ?D.S', ?S''.S' \\
(\_, \mathsf{select}\ l_j\ \mathsf{on}\ v) & v \textit{ not a channel} \\
(\Sigma, \mathsf{select}\ l_j\ \mathsf{on}\ c) & c\colon S \textit{ not in } \Sigma \\
(\Sigma + c\colon S, \mathsf{select}\ l_j\ \mathsf{on}\ c) & S \neq \oplus\langle\ l_i : S_i\ \rangle_{i\in I} \\
(\Sigma + c\colon \oplus\langle\ l_i : S_i\ \rangle_{i\in I}, \mathsf{select}\ l_j\ \mathsf{on}\ c) & j \textit{ not in } I
\end{array}$$

Well typed closed environments are not faulty.

Theorem 8 *If* $\vdash \Sigma \triangleright e : \_ \triangleleft \_$ *then* (Σ, e) *is not faulty.*

Proof: By a case analysis on the extensional definition of faulty programs.

Case $(\_, \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e')$. We inspect all possible derivations of

$$\_ \vdash \Sigma + \Sigma' \triangleright \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e' : \_ \triangleleft \_.$$

They are of the form:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \Sigma \triangleright v : \mathsf{Bool} \triangleleft \Sigma} \qquad \ldots}{\Gamma \vdash \Sigma + \Sigma' \triangleright \mathsf{if}\ v \cdots : \_ \triangleleft \Sigma'' + \Sigma'}\ \textsc{T-If}, \textsc{T-Weak}^*$$

There are four kinds of values: variables (ruled out, since configurations under consideration are closed); abstractions and channels (ruled out, since the type in the conclusion of rule T-Abs, respectively T-Chan, cannot be Bool); and constants. Of all the constants in the language, only true and false have type Bool; hence v must be one of these.

Case $(\_, \mathsf{receive}\ v)$. All the derivation trees of $\_ \vdash \Sigma + \Sigma' \triangleright \mathsf{receive}\ v : \_ \triangleleft \_$ are of the form below, where T-Receive stands for T-ReceiveD or T-ReceiveS (Figure 7), and S is $?D.S'$ or $?S''.S'$, respectively.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \Sigma + c\colon S \triangleright v : \mathsf{Chan}\ c \triangleleft \_}}{\Gamma \vdash \Sigma + \Sigma' + c\colon S \triangleright \mathsf{receive}\ v : \_ \triangleleft \_}\ \textsc{T-Receive}, \textsc{T-Weak}^*$$

Reasoning as above, we conclude that the only values of type $\mathsf{Chan}\ c$ are channels. We can easily see that $c\colon S'$ is in the conclusion; and that S' is $S' = ?D.S$ or $S' = ?S''.S$.

The remaining cases (unfold, send, close, case, and select) are similar. □

Our final result states that well-typed programs do not reduce to faulty configurations. This eliminates the need for runtime checks.

Corollary 9 *If* $\vdash \Sigma \triangleright e : \_ \triangleleft \_$ *and* $\Sigma, e \to^* \Sigma', e'$, *then* (Σ', e') *is not faulty.*

Proof: By induction on the length of the derivation of $\Sigma, e \to^* \Sigma', e'$. If the length is zero then use Lemma 2, otherwise use Theorem 7 followed by induction. □

8 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 type system describes protocols in less detail than we can achieve with session types.

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 [18] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [8] 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, 13] 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.

9 Conclusions and Future Work

We have transferred the concept of session types from the π-calculus to a language based on λ-calculus with side-effecting input/output operations. 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 express recursion (essential for the implementation of protocols whose session types are recursive) by means of functions and a fixed point combinator, rather than through explicitly recursive process definitions or the replication operator of the π-calculus.

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.

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

- We have already mentioned (Section 2.4) that to take full advantage of functions, we need to add some form of polymorphism which allows generalization of channel identifiers in function types.

- The next major step is to incorporate channels and session types into a core object-oriented language. The relationship between session types, subtyping and inheritance will introduce complexity, but the object-oriented paradigm will allow us to achieve a technical simplification by restricting attention to first-order functions. We anticipate a programming style in which an object contains a private channel, and its public methods use this channel. The session type of the channel will constrain the use of those methods, perhaps imposing a particular sequence of method calls. This opens up the possibility of a connection with type systems for non-uniform objects [11, 12, 14]: perhaps a non-uniform object type could be generated by the session type of a private channel.

- The purpose of our language is to support typed programming with inter-process communication channels, but we have only considered individual processes in isolation; we do not have concurrency. We could add a parallel composition operator along the lines of the π-calculus, in order to be able to define systems of communicating processes. However, in line with our aim of working within a conventional programming paradigm, we would prefer to allow certain functions to execute as separate threads; we would then be able to define a multi-threaded server, for example.

- We could consider adding ML-style references and assignment. This would introduce further issues of aliasing. We do not yet know whether our present infrastructure for typechecking in the presence of aliasing would be sufficient for this extension.

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. The work was also 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, via project MIMO (POSI/CHS/39789/2001).

The authors would like to thank M. Hennessy, L. Caires and A. Sernadas for important feedback and suggestions.

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 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (SIGPLAN Notices 36(5))*, 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*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 1999.

[4] A. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In S. Brooks and M. Mislove, editors, *MFPS 2001: 17th Conference on the Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[5] M. J. Hole and S. J. Gay. Bounded polymorphism in session types. Technical Report TR-2003-132, Department of Computing Science, University of Glasgow, March 2003.

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

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

[8] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings, 29th ACM Symposium on Principles of Programming Languages*, pages 331–342. ACM Press, 2002.

[9] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.

[10] J. Myers and M. Rose. Post office protocol version 3, May 1996. Internet Standards RFC1939.

[11] O. Nierstrasz. Regular types for active objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993.

[12] F. Puntigam. Coordination requirements expressed in types for active objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[13] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In P. Cousot, editor, *Static Analysis: 8th International Symposium, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, pages 375–394. Springer-Verlag, 2001.

[14] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00: Proceedings of the International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2000.

[15] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[16] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Proceedings*, volume 817 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[17] 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)*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier, August 2002.

[18] 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.

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

A Proof of Lemma 6 (Substitution)

Lemma 6 (Substitution) *If* $\Gamma + x\colon T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'$ *and* $\Gamma \vdash \emptyset \triangleright v : T \triangleleft \emptyset$ *then* $\Gamma \vdash \Sigma \triangleright e\{v/x\} : U \triangleleft \Sigma'$.

Proof: By induction on the derivation of $\Gamma + x\colon T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'$, with a case-analysis on the last rule. We describe each case in the following format.

Given The last step of a derivation of $\Gamma + x\colon T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'$, perhaps with more specific values of Γ, Σ and Σ' deduced from the last typing rule.

Substitution The calculation of $e\{v/x\}$.

Conclusion A derivation of $\Gamma \vdash \Sigma \triangleright e\{v/x\} : U \triangleleft \Sigma'$, which can be constructed from the given information and the use of the induction hypothesis.

Case T-Const

Given

$$\frac{U \in \mathit{typeof}(u)}{\Gamma + x\colon T \vdash \emptyset \triangleright u : U \triangleleft \emptyset}\ \text{T-C{\scriptsize ONST}}$$

Substitution

$$u\{v/x\} = u$$

Conclusion

$$\frac{U \in \mathit{typeof}(u)}{\Gamma + x\colon T \vdash \emptyset \triangleright u : U \triangleleft \emptyset}\ \text{T-C{\scriptsize ONST}}$$

Case T-Var, same variable

Given

$$\frac{}{\Gamma + x\colon T \vdash \emptyset \triangleright x : T \triangleleft \emptyset}\ \text{T-V{\scriptsize AR}}$$

Substitution

$$x\{v/x\} = v$$

Conclusion

$$\frac{}{\Gamma \vdash \emptyset \triangleright v : T \triangleleft \emptyset}\ \text{T-V{\scriptsize AR}}$$

Case T-Var, different variable

Given

$$\frac{}{\Gamma + x\colon T + y\colon U \vdash \emptyset \triangleright y : U \triangleleft \emptyset}\ \text{T-V{\scriptsize AR}}$$

Substitution

$$y\{v/x\} = y$$

Conclusion

$$\frac{}{\Gamma + y\colon U \vdash \emptyset \triangleright y : U \triangleleft \emptyset}\ \textrm{T-V{\scriptsize AR}}$$

Case T-Chan

Given

$$\frac{}{\Gamma + x\colon T \vdash \emptyset \triangleright c : \mathsf{Chan}\ c \triangleleft \emptyset}\ \textrm{T-C{\scriptsize HAN}}$$

Substitution

$$c\{v/x\} = c$$

Conclusion

$$\frac{}{\Gamma \vdash \emptyset \triangleright c : \mathsf{Chan}\ c \triangleleft \emptyset}\ \textrm{T-C{\scriptsize HAN}}$$

Case T-New

Given

$$\frac{}{\Gamma + x\colon T \vdash \emptyset \triangleright \mathsf{new}\ S : \mathsf{Chan}\ c \triangleleft c\colon S}\ \textrm{T-N{\scriptsize EW}}$$

Substitution

$$(\mathsf{new}\ S)\{v/x\} = \mathsf{new}\ S$$

Conclusion

$$\frac{}{\Gamma \vdash \emptyset \triangleright \mathsf{new}\ S : \mathsf{Chan}\ c \triangleleft c\colon S}\ \textrm{T-N{\scriptsize EW}}$$

Case T-If

Given

$$\frac{\Gamma + x\colon T \vdash \Sigma_1 \triangleright e : \mathsf{Bool} \triangleleft \Sigma' \qquad \begin{array}{c}\Gamma + x\colon T \vdash \Sigma' \triangleright e' : U \triangleleft \Sigma_2 \\ \Gamma + x\colon T \vdash \Sigma' \triangleright e'' : U \triangleleft \Sigma_2\end{array}}{\Gamma + x\colon T \vdash \Sigma_1 \triangleright \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : U \triangleleft \Sigma_2}\ \textrm{T-I{\scriptsize F}}$$

Substitution

$$(\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'')\{v/x\} = \mathsf{if}\ e\{v/x\}\ \mathsf{then}\ e'\{v/x\}\ \mathsf{else}\ e''\{v/x\}$$

Conclusion

$$\dfrac{\Gamma \vdash \Sigma_1 \triangleright e\{v/x\} : \mathsf{Bool} \triangleleft \Sigma' \qquad \begin{array}{l}\Gamma \vdash \Sigma' \triangleright e'\{v/x\} : U \triangleleft \Sigma_2 \\ \Gamma \vdash \Sigma' \triangleright e''\{v/x\} : U \triangleleft \Sigma_2\end{array}}{\Gamma \vdash \Sigma_1 \triangleright \mathsf{if}\ e\{v/x\}\ \mathsf{then}\ e'\{v/x\}\ \mathsf{else}\ e''\{v/x\} : U \triangleleft \Sigma_2}\ \text{T-IF}$$

Case T-Abs

Given

$$\dfrac{\Gamma + x\colon T + y\colon V \vdash \Sigma \triangleright e : U \triangleleft \Sigma'}{\Gamma + x\colon T \vdash \emptyset \triangleright \lambda y.e : \Sigma; V \to U; \Sigma' \triangleleft \emptyset}\ \text{T-ABS}$$

Substitution

$$(\lambda y.e)\{v/x\} = \lambda y.e\{v/x\}$$

Conclusion

$$\dfrac{\Gamma + y\colon V \vdash \Sigma \triangleright e\{v/x\} : U \triangleleft \Sigma'}{\Gamma \vdash \emptyset \triangleright \lambda y.e\{v/x\} : \Sigma; V \to U; \Sigma' \triangleleft \emptyset}\ \text{T-ABS}$$

Case T-App

Given

$$\dfrac{\Gamma + x\colon T \vdash \Sigma \triangleright e : (\Sigma_1; V \to U; \Sigma_2) \triangleleft \Sigma'' \qquad \Gamma + x\colon T \vdash \Sigma'' \triangleright e' : V \triangleleft \Sigma_1 + \Sigma'}{\Gamma + x\colon T \vdash \Sigma \triangleright ee' : U \triangleleft \Sigma_2 + \Sigma'}\ \text{T-APP}$$

Substitution

$$(ee')\{v/x\} = e\{v/x\}e'\{v/x\}$$

Conclusion

$$\dfrac{\Gamma \vdash \Sigma \triangleright e\{v/x\} : (\Sigma_1; V \to U; \Sigma_2) \triangleleft \Sigma'' \qquad \Gamma \vdash \Sigma'' \triangleright e'\{v/x\} : V \triangleleft \Sigma_1 + \Sigma'}{\Gamma \vdash \Sigma \triangleright e\{v/x\}e'\{v/x\} : U \triangleleft \Sigma_2 + \Sigma'}\ \text{T-APP}$$

Case T-Case

Given

$$\dfrac{\begin{array}{l}\Gamma + x\colon T \vdash \Sigma \triangleright e : \mathsf{Chan}\ c \triangleleft \Sigma'' + c\colon \&\langle\ l_i : S_i\ \rangle_{i \in I} \\ \forall i.(\Gamma + x\colon T \vdash \Sigma'' + c\colon S_i \triangleright e_i : U \triangleleft \Sigma')\end{array}}{\Gamma + x\colon T \vdash \Sigma \triangleright \mathsf{case}\ e\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} : U \triangleleft \Sigma'}\ \text{T-CASE}$$

Substitution

$$(\mathsf{case}\ e\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I})\{v/x\} = \mathsf{case}\ e\{v/x\}\ \mathsf{of}\ \{l_i \Rightarrow e_i\{v/x\}\}_{i\in I}$$

Conclusion

$$\dfrac{\begin{array}{l}\Gamma \vdash \Sigma \triangleright e\{v/x\} : \mathsf{Chan}\ c \triangleleft \Sigma'' + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I}\\ \forall i.(\Gamma \vdash \Sigma'' + c\colon S_i \triangleright e_i\{v/x\} : U \triangleleft \Sigma')\end{array}}{\Gamma \vdash \Sigma \triangleright \mathsf{case}\ e\{v/x\}\ \mathsf{of}\ \{l_i \Rightarrow e_i\{v/x\}\}_{i\in I} : U \triangleleft \Sigma'}\ \text{T-Case}$$

Case T-Select

Given

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

Substitution

$$(\mathsf{select}\ l_i\ \mathsf{on}\ e)\{v/x\} = \mathsf{select}\ l_i\ \mathsf{on}\ e\{v/x\}$$

Conclusion

$$\dfrac{\Gamma \vdash \Sigma \triangleright e\{v/x\} : \mathsf{Chan}\ c \triangleleft \Sigma'' + c\colon \oplus\langle\ l_i : S_i\ \rangle_{i\in I}}{\Gamma \vdash \Sigma \triangleright \mathsf{select}\ l_i\ \mathsf{on}\ e\{v/x\} : U \triangleleft \Sigma' + c\colon S_i}\ \text{T-Select}$$

Case T-Weak

Given

$$\dfrac{\Gamma + x\colon T \vdash \Sigma \triangleright e : U \triangleleft \Sigma'}{\Gamma + x\colon T \vdash \Sigma + c\colon S \triangleright e : U \triangleleft \Sigma' + c\colon S}\ \text{T-Weak}$$

Substitution

$$e\{v/x\}$$

Conclusion

$$\dfrac{\Gamma \vdash \Sigma \triangleright e\{v/x\} : U \triangleleft \Sigma'}{\Gamma \vdash \Sigma + c\colon S \triangleright e\{v/x\} : U \triangleleft \Sigma' + c\colon S}\ \text{T-Weak}$$

□

B Proof of Theorem 7 (Subject Reduction)

Lemma 7 (Subject Reduction for $\longrightarrow_{\mathsf{v}}$) *If* $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$ *and* $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma'', e'$ *then* $\Gamma \vdash \Sigma'' \triangleright e' : T \triangleleft \Sigma'$.

Proof: By case-analysis on the rule used to derive $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma'', e'$. In each case we show part of the derivation of $\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'$, the reduction $\Sigma, e \longrightarrow_{\mathsf{v}} \Sigma'', e'$, and the derivation of the desired conclusion $\Gamma \vdash \Sigma'' \triangleright e' : T \triangleleft \Sigma'$ from the components of the given derivation.

Case R-IfT, R-IfF

Given

$$\cfrac{\cfrac{\Gamma \vdash \Sigma \triangleright \mathsf{true} : \mathsf{Bool} \triangleleft \Sigma \qquad \Gamma \vdash \Sigma \triangleright e_1 : T \triangleleft \Sigma' \qquad \Gamma \vdash \Sigma \triangleright e_2 : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 : T \triangleleft \Sigma'}\ \text{T-If}}{\Gamma \vdash \Sigma + \Sigma_1 \triangleright \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 : T \triangleleft \Sigma' + \Sigma_1}\ \text{T-Weak}^*$$

Reduction

$$\Sigma + \Sigma_1, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow_{\mathsf{v}} \Sigma + \Sigma_1, e_1$$

Conclusion

$$\cfrac{\Gamma \vdash \Sigma \triangleright e_1 : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma + \Sigma_1 \triangleright e_1 : T \triangleleft \Sigma' + \Sigma_1}\ \text{T-Weak}^*$$

The case of R-IfF is similar.

Case R-Beta

Given

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma + x\colon T \vdash \Sigma_1 \triangleright e : U \triangleleft \Sigma_2}{\Gamma \vdash \emptyset \triangleright \lambda x.e : (\Sigma_1; T \to U; \Sigma_2) \triangleleft \emptyset}\ \text{T-Abs}}{\Gamma \vdash \Sigma \triangleright \lambda x.e : (\Sigma_1; T \to U; \Sigma_2) \triangleleft \Sigma}\ \text{T-Weak}^* \qquad \cfrac{\Gamma \vdash \emptyset \triangleright v : T \triangleleft \emptyset}{\Gamma \vdash \Sigma \triangleright v : T \triangleleft \Sigma = \Sigma_1 + \Sigma'}\ \text{T-Weak}^*}{\Gamma \vdash \Sigma \triangleright (\lambda x.e)v : U \triangleleft \Sigma_2 + \Sigma'}\ \text{T-App}}{\Gamma \vdash \Sigma + \Sigma_3 \triangleright (\lambda x.e)v : U \triangleleft \Sigma_2 + \Sigma' + \Sigma_3}\ \text{T-Weak}^*$$

Reduction

$$\Sigma + \Sigma_3, (\lambda x.e)v \longrightarrow_{\mathsf{v}} \Sigma + \Sigma_3, e\{v/x\}$$

Conclusion

$$\cfrac{\cfrac{\Gamma + x\colon T \vdash \Sigma_1 \triangleright e : U \triangleleft \Sigma_2 \qquad \Gamma \vdash \emptyset \triangleright v : T \triangleleft \emptyset}{\Gamma \vdash \Sigma_1 \triangleright e\{v/x\} : U \triangleleft \Sigma_2}\ \text{Lemma 6 (Substitution)}}{\Gamma \vdash \Sigma + \Sigma_3 = \Sigma_1 + \Sigma' + \Sigma_3 \triangleright e\{v/x\} : U \triangleleft \Sigma_2 + \Sigma' + \Sigma_3}\ \text{T-Weak}^*$$

Case R-Fix

Given

$$\dfrac{\dfrac{\Gamma \vdash \Sigma \triangleright \mathsf{fix} : (\emptyset; (\emptyset; T \to T; \emptyset) \to T; \emptyset) \triangleleft \Sigma \qquad \Gamma \vdash \Sigma \triangleright v : (\emptyset; T \to T; \emptyset) \triangleleft \Sigma}{\Gamma \vdash \Sigma \triangleright \mathsf{fix}\ v : T \triangleleft \Sigma}\ \textsc{T-App}}{\Gamma \vdash \Sigma + \Sigma_3 \triangleright \mathsf{fix}\ v : T \triangleleft \Sigma + \Sigma_3}\ \textsc{T-Weak}^*$$

where $T = \Sigma_1; T_1 \to T_2; \Sigma_2$.

Reduction

$$\Sigma + \Sigma_3, \mathsf{fix}\ v \longrightarrow_{\mathsf{v}} \Sigma + \Sigma_3, v(\lambda x.\mathsf{fix}\ v\ x)$$

Conclusion

$$\dfrac{\dfrac{\Gamma \vdash \Sigma \triangleright v : (\emptyset; T \to T; \emptyset) \triangleleft \Sigma \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \Sigma \triangleright v : (\emptyset; T \to T; \emptyset) \triangleleft \Sigma}{\Gamma \vdash \emptyset \triangleright v : (\emptyset; T \to T; \emptyset) \triangleleft \emptyset}\ \text{Lemma 5}}{\Gamma \vdash \Sigma_1 \triangleright v : (\emptyset; T \to T; \emptyset) \triangleleft \Sigma_1}\ \textsc{T-Weak}^*}{\Gamma + x\colon T_1 \vdash \Sigma_1 \triangleright \mathsf{fix}\ v : T \triangleleft \Sigma_1}\ \textsc{T-App} \qquad \dfrac{}{\Gamma + x\colon T_1 \vdash \Sigma_1 \triangleright x : T_1 \triangleleft \Sigma_1}\ \textsc{T-Var}}{\Gamma + x\colon T_1 \vdash \Sigma_1 \triangleright \mathsf{fix}\ v\ x : T_2 \triangleleft \Sigma_2}\ \textsc{T-Abs}}{\Gamma \vdash \emptyset \triangleright \lambda x.\mathsf{fix}\ v\ x : T \triangleleft \emptyset}\ \textsc{T-Abs}}{\Gamma \vdash \Sigma \triangleright \lambda x.\mathsf{fix}\ v\ x : T \triangleleft \Sigma}\ \textsc{T-Weak}^*}{\Gamma \vdash \Sigma \triangleright v(\lambda x.\mathsf{fix}\ v\ x) : T \triangleleft \Sigma}\ \textsc{T-App}}{\Gamma \vdash \Sigma + \Sigma_3 \triangleright v(\lambda x.\mathsf{fix}\ v\ x) : T \triangleleft \Sigma + \Sigma_3}\ \textsc{T-Weak}^*$$

Case R-New

Given

$$\dfrac{\Gamma \vdash \emptyset \triangleright \mathsf{new}\ S : \mathsf{Chan}\ c \triangleleft c\colon S}{\Gamma \vdash \Sigma \triangleright \mathsf{new}\ S : \mathsf{Chan}\ c \triangleleft \Sigma + c\colon S}\ \textsc{T-Weak}^*$$

Reduction

$$\Sigma, \mathsf{new}\ S \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, c$$

where we assume that the identifier c of the channel created by the reduction is the same as the channel identifier used in the typing derivation.

Conclusion

$$\dfrac{\dfrac{}{\Gamma \vdash \emptyset \triangleright c : \mathsf{Chan}\ c \triangleleft \emptyset}\ \textsc{T-Chan}}{\Gamma \vdash \Sigma + c\colon S \triangleright c : \mathsf{Chan}\ c \triangleleft \Sigma + c\colon S}\ \textsc{T-Weak}^*$$

Case R-SendD

Given

$$\dfrac{\begin{array}{c}\mathsf{send}\colon D \to (c\colon !D.S; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon S)\\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon !D.S \rhd \mathsf{send}\ v\ \mathsf{on}\ c : \mathsf{Unit} \lhd \Sigma + c\colon S}\ \text{T-}\textsc{App}$$

Reduction

$$\Sigma + c\colon !D.S, \mathsf{send}\ v\ \mathsf{on}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, \mathsf{unit}$$

Conclusion

$$\dfrac{}{\Gamma \vdash \Sigma + c\colon S \rhd \mathsf{unit} : \mathsf{Unit} \lhd \Sigma + c\colon S}\ \text{T-}\textsc{Const}$$

Case R-SendS

Given

$$\dfrac{\begin{array}{c}\mathsf{send}\colon \mathsf{Chan}\ d \to (c\colon !S'.S + d\colon S'; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon S)\\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon !S'.S + d\colon S' \rhd \mathsf{send}\ d\ \mathsf{on}\ c : \mathsf{Unit} \lhd \Sigma + c\colon S}\ \text{T-}\textsc{App}$$

Reduction

$$\Sigma + c\colon !S'.S + d\colon S', \mathsf{send}\ d\ \mathsf{on}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, \mathsf{unit}$$

Conclusion

$$\dfrac{}{\Gamma \vdash \Sigma + c\colon S \rhd \mathsf{unit} : \mathsf{Unit} \lhd \Sigma + c\colon S}\ \text{T-}\textsc{Const}$$

Case R-ReceiveD

Given

$$\dfrac{\begin{array}{c}\mathsf{receive}\colon c\colon ?D.S; \mathsf{Chan}\ c \to D; c\colon S\\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon ?D.S \rhd \mathsf{receive}\ c : D \lhd \Sigma + c\colon S}\ \text{T-}\textsc{App}$$

Reduction

$$\Sigma + c\colon ?D.S, \mathsf{receive}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S, v$$

where v is a closed value of type D, i.e. $\emptyset \vdash \emptyset \rhd v : D \lhd \emptyset$.

Conclusion

$$\dfrac{\dfrac{\emptyset \vdash \emptyset \rhd v : D \lhd \emptyset}{\Gamma \vdash \emptyset \rhd v : D \lhd \emptyset}\ \text{Lemma 4}}{\Gamma \vdash \Sigma + c\colon S \rhd v : D \lhd \Sigma + c\colon S}\ \text{T-}\textsc{Weak}^*$$

Case R-ReceiveS

Given

$$\dfrac{\begin{array}{c}\mathsf{receive}\colon c\colon ?S'.S; \mathsf{Chan}\ c \rightarrow \mathsf{Chan}\ d; c\colon S + d\colon S' \\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon ?S'.S \triangleright \mathsf{receive}\ c : \mathsf{Chan}\ d \triangleleft \Sigma + c\colon S + d\colon S'}\ \textsc{T-App}$$

Reduction

$$\Sigma + c\colon ?S'.S, \mathsf{receive}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S + d\colon S', d$$

where we assume that the identifier d of the received channel is the same as the channel identifier used in the typing derivation.

Conclusion

$$\dfrac{}{\Gamma \vdash \Sigma + c\colon S + d\colon S' \triangleright d : \mathsf{Chan}\ d \triangleleft \Sigma + c\colon S + d\colon S'}\ \textsc{T-Chan}$$

Case R-Close

Given

$$\dfrac{\begin{array}{c}\mathsf{close}\colon c\colon \mathsf{End}; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; \emptyset \\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon \mathsf{End} \triangleright \mathsf{close}\ c : \mathsf{Unit} \triangleleft \Sigma}\ \textsc{T-App}$$

Reduction

$$\Sigma + c\colon \mathsf{End}, \mathsf{close}\ c \longrightarrow_{\mathsf{v}} \Sigma, \mathsf{unit}$$

Conclusion

$$\dfrac{}{\Gamma \vdash \Sigma \triangleright \mathsf{unit} : \mathsf{Unit} \triangleleft \Sigma}\ \textsc{T-Const}$$

Case R-Case

Given

$$\dfrac{\begin{array}{l}\Gamma \vdash \Sigma + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I} \triangleright c : \mathsf{Chan}\ c \triangleleft \Sigma + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I} \\ \forall i.(\Gamma \vdash \Sigma + c\colon S_i \triangleright e_i : T \triangleleft \Sigma')\end{array}}{\Gamma \vdash \Sigma + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I} \triangleright \mathsf{case}\ c\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I} : T \triangleleft \Sigma'}\ \textsc{T-Case}$$

Reduction

$$\Sigma + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I}, \mathsf{case}\ c\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I} \longrightarrow_{\mathsf{v}} \Sigma + c\colon S_j, e_j$$

for some $j \in I$.

Conclusion

$$\Gamma \vdash \Sigma + c\colon S_j \triangleright e_j : T \triangleleft \Sigma'$$

Case R-Select

Given

$$\frac{\Gamma \vdash \Sigma + c\colon \oplus\langle\ l_i : S_i\ \rangle_{i\in I} \triangleright c : \mathsf{Chan}\ c \triangleleft \Sigma + c\colon \&\langle\ l_i : S_i\ \rangle_{i\in I}}{\Gamma \vdash \Sigma + c\colon \oplus\langle\ l_i : S_i\ \rangle_{i\in I} \triangleright \mathsf{select}\ l_j\ \mathsf{on}\ c : \mathsf{Unit} \triangleleft \Sigma + c\colon S_j}\ \text{T-Select}$$

Reduction

$$\Sigma + c\colon \oplus\langle\ l_i : S_i\ \rangle_{i\in I}, \mathsf{select}\ l_j\ \mathsf{on}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S_j, \mathsf{unit}$$

where $j \in I$.

Conclusion

$$\frac{}{\Gamma \vdash \Sigma + c\colon S_j \triangleright \mathsf{unit} : \mathsf{Unit} \triangleleft \Sigma + c\colon S_j}\ \text{T-Const}$$

Case R-Unfold

Given

$$\frac{\begin{array}{c}\mathsf{unfold}\colon c\colon \mu X\cdot S; \mathsf{Chan}\ c \rightarrow \mathsf{Unit}; c\colon S\{(\mu X\cdot S)/X\}\\ \vdots\end{array}}{\Gamma \vdash \Sigma + c\colon \mu X\cdot S \triangleright \mathsf{unfold}\ c : \mathsf{Unit} \triangleleft \Sigma + c\colon S\{(\mu X\cdot S)/X\}}\ \text{T-App}$$

Reduction

$$\Sigma + c\colon \mu X\cdot S, \mathsf{unfold}\ c \longrightarrow_{\mathsf{v}} \Sigma + c\colon S\{(\mu X\cdot S)/X\}, \mathsf{unit}$$

Conclusion

$$\frac{}{\Gamma \vdash \Sigma + c\colon S\{(\mu X\cdot S)/X\} \triangleright \mathsf{unit} : \mathsf{Unit} \triangleleft \Sigma + c\colon S\{(\mu X\cdot S)/X\}}\ \text{T-Const}$$

□

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: The reduction $\Sigma, e \longrightarrow \Sigma'', e'$ has the form $\Sigma, E[e_1] \longrightarrow \Sigma'', E[e_1']$ where $E[\ ]$ is an evaluation context, $e = E[e_1]$, $e' = E[e_1']$ and $\Sigma, e_1 \longrightarrow_{\mathsf{v}} \Sigma'', e_1'$.

By Lemma 2, there exist Σ_1 and U such that $\Gamma \vdash \Sigma \triangleright e_1 : U \triangleleft \Sigma_1$. By Lemma 7 we have $\Gamma \vdash \Sigma'' \triangleright e_1' : U \triangleleft \Sigma_1$. By Lemma 3 we have $\Gamma \vdash \Sigma'' \triangleright E[e_1'] : T \triangleleft \Sigma'$ as required. □