Typechecking a Multithreaded Functional Language with Session Types

Vasco T. Vasconcelos
Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal

Simon J. Gay
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK

António Ravara
CLC and Departamento de Matemática, Instituto Superior Técnico, 1049-001 Lisboa, Portugal

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 the state transitions of a protocol and also the data types of messages associated with transitions; thus typechecking can verify both correctness of individual messages and correctness of sequences of transitions. Previously, session types have mainly been studied in the context of the $\pi$-calculus; instead, our formulation is based on a multi-threaded functional language with side-effecting input/output operations. 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 channels. We formalize the syntax, semantics and typing system of our language, and prove subject reduction and runtime type safety theorems. We also present a type checking algorithm and prove that it is correct with respect to the type system.

Key words: Session types, static typechecking, concurrent programming, specification of communication protocols.


Email addresses: vv@di.fc.ul.pt (Vasco T. Vasconcelos),
1 Introduction

Communication in distributed systems is typically structured around protocols which specify the sequence and form of messages passing over communication channels. Correctness of such systems implies that protocols are obeyed.

Systems programming is traditionally performed in the C programming language. It thus comes as no surprise that many attempts to statically check protocols are based on this language: safe control of stateful resources is achieved via type systems that either run on annotated C programs [8, 12], or on programs written in a type-safe variant of C [17, 18]. Another approach to proving properties of protocols comes from the general setting of the π-calculus [24, 31], and includes type and effect systems to check correspondence assertions [2, 3, 16], the approximation of the behaviour of π-processes by CCS terms [6, 29], and session types to describe structured communication programming [13, 14, 19, 20, 32].

Session types allow the specification of a protocol to be expressed as a type; when a communication channel is created, a session type is associated with it. Such a 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 for conventional languages, 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, but until now, has not been studied theoretically in the context of a standard language paradigm, despite a few contributions which bridge session types and conventional languages. Session types have been used to add behavioural information to the interfaces of CORBA objects [33] using Gay and Hole’s theory of subtyping [13, 14] to formalise compatibility and substitutability of components. Session types have also been encoded in the Haskell programming language [25]. The former does not link the improved CORBA interfaces to actual programming languages; the latter does not address the correspondence between a session-based programming language and Haskell.

Very recently, Dezani-Ciancaglini et al. [9] have proposed a minimal distributed object-oriented language with session types. A detailed comparison of our system and theirs will be the subject of future work.

Our contribution to the problem of structured communication-based programming in general, and the verification of protocols in particular, is to transfer

\begin{flushright}
\texttt{simon@dcs.gla.ac.uk} (Simon J. Gay), \texttt{amar@math.ist.utl.pt} (António Ravara).
\end{flushright}
the concept of session types from the \( \pi \)-calculus to a multi-threaded functional language with side-effecting input/output operations. This shows that static checking of session types could be added to a language such as Concurrent ML [30] (at least without imperative features) or Concurrent Haskell [27] (cf. [25]). More generally, it is our view that by starting with typing concepts which are well-understood in the context of a theoretical calculus, and transferring them to a language which is closer to mainstream programming practice, we can achieve a powerful type system which is suited to practical programming while retaining the benefits of a sound foundation.

The key technical steps which we have undertaken, in order to address the differences between a conventional programming style and the programming notation of the \( \pi \)-calculus, are as follows:

- The operations on channels are 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 (that is, the collection of channel names and their types).
- We have separated naming and creation 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.

In previous work [15], we have presented a language supporting typed functional programming with inter-process communication channels, but we only considered individual processes in isolation. In [34], we addressed collections of functional threads communicating via (session) channels created from shared names. Here we present the proofs for subject reduction and type safety, claimed in [34], and introduce a type checking algorithm for the language, together with a proof of its correctness.

The structure of the paper is as follows. In Section 2, we explain session types in connection with a progressively more sophisticated example. Sections 3 to 5 define the syntax, operational semantics, and type system of our language. In Section 6, we present the runtime safety result. Section 7 presents a type checking algorithm. In Sections 8 and 9, we discuss related and future work. The appendices contain the proofs of all the results in the paper.

2 Session Types and the Maths Server

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

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

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 $u$; we think of the client engaging in a session with the server, using the channel $u$ for communication. In our language, the server looks like this:

$$\text{server } u = \text{let } x = \text{receive } u \text{ in}
\text{let } y = \text{receive } u \text{ in}
\text{send } x + y \text{ on } u$$

or more concisely: $\text{send } ((\text{receive } u) + (\text{receive } u)) \text{ on } u$.

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

$$\bar{S} = !\text{Int}.!\text{Int}.?\text{Int}.\text{End}$$

and a client implementation uses the server to add two particular integers; the code may use $x$ but cannot use the channel $u$ except for closing it.

$$\text{client } u = \text{send } 2 \text{ on } u
\text{send } 3 \text{ on } u
\text{let } x = \text{receive } u \text{ in } \text{code}$$

**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.

$$S = \& (\text{add}: ?\text{Int}.?\text{Int}.!\text{Int}.\text{End}, \text{neg}: ?\text{Int}.!\text{Int}.\text{End})$$
Both services must be implemented. We introduce a case construct:

\[
\text{server } u = \text{ case } u \text{ of } \{
\text{add } \Rightarrow \text{ send } (\text{receive } u) + (\text{receive } u) \text{ on } u \\
\text{neg } \Rightarrow \text{ send } -(\text{receive } u) \text{ on } u \}
\]

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

\[
\overline{S} = \oplus\langle add : !\text{Int}!\text{Int}.?\text{Int.End}, neg : !\text{Int}.?\text{Int.End} \rangle
\]

A client implementation makes a particular choice, for example:

\[
\text{addClient } u = \text{ select add on } u \\
\text{ send 2 on } u \\
\text{ send 3 on } u \\
\text{ let } x = \text{ receive } u \text{ in code}
\]

\[
\text{negClient } u = \text{ select neg on } u \\
\text{ send 7 on } u \\
\text{ let } x = \text{ receive } u \text{ in code}
\]

Note that the type of the subsequent interaction depends on the label which is selected. 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 \(\text{Real}\), 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 \(\oplus\) 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.

\[
S = \&\langle add : ?\text{Int}.?\text{Int}!\text{Int}.End, \\
\text{neg : } ?\text{Int}!\text{Int}.End, \\
\text{sqrt : } ?\text{Real} . \oplus\langle ok : !\text{Real.End}, error : \text{End} \rangle \rangle
\]

This example shows that session types allow the description of protocols which cannot easily be accommodated with objects, that is, with sequences of the form: select a method; send the arguments; receive the result.
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:

\[
S = \&\langle add : ?\text{Int} . ?\text{Int} . !\text{Int} . S, \\
    neg : ?\text{Int} !\text{Int} . S, \\
    sqrt : ?\text{Real} . \oplus (ok : !\text{Real} . S, \text{error} : S), \\
    quit : \text{End} \rangle
\]

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. To simplify the theory we decided not to include recursive types in this paper; the interested reader may refer to report [15].

Function Types. We have not mentioned the type of the server itself. Clearly, it accepts a channel and returns nothing. If \(c\) is the name of the channel, the input/output behaviour of the function is described by \(\text{Chan} \ c \rightarrow \text{Unit}\). When control enters the function, channel \(c\) is in a state where it offers add and neg services. The function then “consumes” the channel, leaving it in a state ready to be closed. In order to correctly control channel usage, we annotate function types with the initial and the final type of all the channels used by the function. If \(c\) is the (runtime) channel denoted by the (program) variable \(u\), we may assign the following type to server.

\[
\begin{aligned}
\text{server} :: (c : &\langle \text{add} : \ldots, \text{neg} : \ldots \rangle; \text{Chan} \ c \rightarrow \text{Unit}; c : \text{End}) \\
\text{server} \ u = \text{case} \ u \ of \ \{ \text{add} \Rightarrow \ldots, \text{neg} \Rightarrow \ldots \}
\end{aligned}
\]

Note how the function type describes not only the type of the parameter and that of the result, but also its effect on channel \(c\). It can also be useful to send functions on channels. For example we could add the component\(^1\)

\[
\text{eval} : ?(\text{Int} \rightarrow \text{Bool}).?\text{Int}!\text{Bool}.\text{End}
\]

to the branch type of the server, with corresponding server code, to be placed within the server’s case above.

\[
\text{eval} \Rightarrow \text{send} (\text{receive} \ u)(\text{receive} \ u) \ on \ u
\]

A client which requires a primality test service (perhaps the server has fast

\(^1\) We often omit the empty channel environment on each side of the arrow, so that Int \rightarrow \text{Bool} is short for \(\emptyset; \text{Int} \rightarrow \text{Bool}; \emptyset\).
hardware) can be written as follows.

\[
\text{primeClient} :: (c: \oplus \langle \text{add}: \ldots, \text{neg}: \ldots, \text{eval}: \ldots \rangle; \text{Chan } c \rightarrow \text{Unit}; c: \text{End})
\]

\[
\text{primeClient } u = \text{select eval on } u
\]

\[
\begin{align*}
& \text{send isPrime on } u \\
& \text{send bigNumber on } u \\
& \text{let } x = \text{receive } u \text{ in code}
\end{align*}
\]

**Establishing a Connection.** How do the client and the server reach a state in which they both know about channel \( u \)? We follow Takeuchi, Kubo and Honda [32], and propose a pair of constructs: \text{request} \( v \) for use by clients, and \text{accept} \( v \) for use by servers. In use, \text{request} and \text{accept} occur in separate threads, and interact with each other to create a new channel. The value \( v \) in both \text{request} and \text{accept}, denotes the common knowledge of the two threads: a \textit{shared name} used solely for the creation of new channels. If \( S \) is the type of a channel, the type of a name used to create channels of type \( S \) is denoted by \([S]\). Functions \text{server} and \text{negClient} now receive a name of type \([\&\langle \text{add}: \ldots, \text{neg}: \ldots, \text{eval}: \ldots \rangle]\), as shown in the following piece of code.

\[
\text{server} :: [\&\langle \text{add}: \ldots, \text{neg}: \ldots, \text{eval}: \ldots \rangle] \rightarrow \text{Unit}
\]

\[
\text{server } x = \text{let } u = \text{accept } x \text{ in (case } u \text{ of } \ldots; \text{close } u)
\]

\[
\text{negClient} :: [\&\langle \text{add}: \ldots, \text{neg}: \ldots, \text{eval}: \ldots \rangle] \rightarrow \text{Unit}
\]

\[
\text{negClient } x = \text{let } u = \text{request } x \text{ in (select neg on } u \ldots; \text{close } u)
\]

Note that the same type for the shared name \( x \) is used both for the server and for the client; it is the \text{accept/request} construct that distinguishes one from the other. This is also where we introduce the operation to \text{close} a channel: \text{accept/request} creates a channel; \text{close} destroys it.

**Sharing Names.** In order for a name to become known by a client and a server, it must be created somewhere and distributed to both. To create a new, potentially shared, name of type \([S]\), we write \text{new } S. To distribute it to a second thread, we \text{fork} a new thread, in whose code the name occurs.\(^2\) Our complete system creates a name \( x \) and launches three threads (a server and two clients), all sharing the newly created name.

\[
\text{system} :: \text{Unit}
\]

\[
\text{system} = \text{let } x = \text{new } \&\langle \text{add}: \ldots, \text{neg}: \ldots, \text{eval}: \ldots \rangle \text{ in}
\]

\[
\text{fork negClient } x; \text{ fork addClient } x; \text{ fork server } x
\]

\(^2\) Alternatively, we may send the name on an existing channel.
Given the above implementation of server, one of the clients will be forever requesting \( x \). Fortunately, it is easy to extend the server to accept more than one connection in its life time.

\[
\text{server} :: \{ \text{add} : \ldots, \text{neg} : \ldots, \text{eval} : \ldots \} \rightarrow \text{Unit} \\
\text{server} \ x = \text{let} \ u = \text{accept} \ x \ \text{in} \ (\text{fork} \ (\text{case} \ u \ \text{of} \ldots; \text{close} \ u)) \\
\]

**Sending Channels on Channels.** Imagine two clients which need to cooperate in their interaction with the server: one client establishes a connection, selects the \( \text{neg} \) operation, and sends the argument; the second client receives the result. After selecting \( \text{neg} \) and sending the argument, the first client must provide the second with the channel to the server. In order to do so, both clients must share a name of type \( ?(\text{?Int}. \text{End}) \) (called \( S \) below) and establish a connection for the sole purpose of transmitting the server channel.

\[
\text{askNeg} :: \{\text{add} : \ldots\} \rightarrow \ [S] \rightarrow \text{Unit} \\
\text{getNeg} :: \ [S] \rightarrow \text{Unit} \\
\text{askNeg} \ x \ y = \text{let} \ u = \text{request} \ x \ \text{in} \\
\hspace{1cm} \text{select} \ \text{neg} \ \text{on} \ u; \text{send} \ 7 \ \text{on} \ u \\
\hspace{1cm} \text{let} \ w = \text{request} \ y \ \text{in} \\
\hspace{1cm} \text{send} \ u \ \text{on} \ w; \text{close} \ w \\
\hspace{1cm} \text{let} \ i = \text{receive} \ u \ \text{in} \\
\hspace{1cm} \text{send} \ u \ \text{on} \ w; \text{close} \ w; \text{code} \\
\text{getNeg} \ y = \text{let} \ w = \text{accept} \ y \ \text{in} \\
\hspace{1cm} \text{let} \ u = \text{receive} \ w \ \text{in} \\
\hspace{1cm} \text{let} \ i = \text{receive} \ u \ \text{in} \\
\hspace{1cm} \text{close} \ u; \text{close} \ w; \text{code}
\]

It is instructive to follow the evolution of the state (the type) of channels \( c \) and \( d \), connected to variables \( u \) and \( w \), respectively. After the execution of the first line of getNeg, \( d \) has type \( S = ?(\text{?Int}. \text{End}). \text{End} \); after the second line, \( d \) is reduced to \( \text{End} \), but \( c \) shows up with type \( ?\text{Int}. \text{End} \); after the third line both channels are of type \( \text{End} \), that is, ready to be closed. By the end of the fourth line, we gather no more information on channels \( c \) and \( d \), for they are now closed. That is the sort of analysis our type system performs.

After sending a channel, no further interaction on the channel is possible. Note that \( \text{askNeg} \) cannot \( \text{close} \ u \), for otherwise the channel’s client side would be closed twice (in \( \text{askNeg} \) and in \( \text{getNeg} \)). On the other hand, channel \( w \) must be closed at both its ends, by \( \text{askNeg} \) and by \( \text{getNeg} \).

The remainder of this section deals with further issues arising from the interaction between types and programming.

**Channel Aliasing.** As soon as we separate creation and naming of channels, aliasing becomes an issue. Consider the function below.

\[
\text{sendSend} \ u \ v = \text{send} \ 1 \ \text{on} \ u; \text{send} \ 2 \ \text{on} \ v
\]
Function \textit{sendSend} can be used in a number of different ways, including the one where \( u \) and \( v \) become aliases for a single underlying channel.

\[
\text{sendTwice} :: c : \text{!Int!Int.End; Chan } c \rightarrow \text{Unit; c: End}
\]

\[
\text{sendTwice } w = \text{sendSend } w \ w
\]

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 the body of function \textit{sendSend}, the types of \( u \) and \( v \) are both \text{Chan } c. \text{The state of } c, \text{initially } \text{!Int!Int.End, is recorded separately.}

\textbf{Free Variables in Functions.} If we write

\[
\text{sendFree } v = \text{send } 1 \text{ on } u; \text{send } 2 \text{ on } v
\]

then function \textit{sendSend} becomes \( \lambda u. \text{sendFree} \). In order to type \textit{sendTwice}, thus effectively aliasing \( u \) and \( v \) in \textit{sendSend}, we must have\(^3\)

\[
\text{sendFree} :: c : \text{!Int!Int.End; Chan } c \rightarrow \text{Unit; c: End}
\]

\[
\text{sendSend} :: c : \text{!Int!Int.End; Chan } c \rightarrow \text{Chan } c \rightarrow \text{Unit; c: End}
\]

in a typing environment associating the type \text{Chan } c to the free variable \( u \) of \text{sendFree}. However, if we do not want to alias \( u \) and \( v \), then we must have

\[
\text{sendFree} :: c : \text{!Int.End, d: !Int.End; Chan } c \rightarrow \text{Unit; c: End, d: End}
\]

\[
\text{sendSend} :: c : \text{!Int.End, d: !Int.End; Chan } c \rightarrow \text{Chan } d \rightarrow \text{Unit; c: End, d: End}
\]

in a typing environment containing \( u: \text{Chan } d \). Note how the above type for \text{sendFree} captures changes to channels that are parameters (\( c \)) and to channels that occur free (\( d \)).

\textbf{Polymorphism.} We have seen that \text{sendFree} admits at least two different types. In order to allow for code reuse we type our \text{let-bound} values as many times as needed, potentially with different types. The paragraph above showed a \text{share/not-share} kind of polymorphism. Other forms include \textit{channel polymorphism} and \textit{session polymorphism}. For an example of channel polymorphism, consider

\[
\text{sendTwiceSendTwice} :: c : S, d: S; \text{Chan } c \rightarrow \text{Chan } d \rightarrow \text{Unit; c: End, d: End}
\]

\[
\text{sendTwiceSendTwice } x \ y = \text{sendTwice } x; \text{sendTwice } y
\]

\(^3\) We abbreviate \( \Sigma; T \rightarrow (\Sigma; U \rightarrow V; \Sigma'); \Sigma' \) to \( \Sigma; T \rightarrow U \rightarrow V; \Sigma' \).
\[ v ::= c \mid n \mid x \mid \lambda(\Sigma; x: T).e \mid \text{rec} (x: T).v \mid \text{true} \mid \text{false} \mid \text{unit} \]
\[
e ::= t \mid vv \mid \text{if } v \text{ then } e \text{ else } e \mid \text{new } S \mid \text{accept } v \mid \text{request } v \mid \text{send } v \text{ on } v \mid \text{receive } v \mid \text{case } v \text{ of } \{ l_i \Rightarrow e_i \}_{i \in I} \mid \text{select } l \text{ on } v \mid \text{close } v \]
\[
t ::= v \mid \text{let } x = e \text{ in } t \mid \text{fork } t; t \]
\[
C ::= \langle t \rangle \mid (C \mid C) \mid (\nu n: [S])C \mid (\nu c)C \]

Fig. 1. Syntax of values, expressions, threads, and configurations

\[ S ::= ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \&\langle l_i: S_i \rangle_{i \in I} \mid \oplus \langle l_i: S_i \rangle_{i \in I} \mid \text{End} \mid \perp \]
\[
D ::= \text{Bool} \mid \text{Unit} \mid \Sigma; T \rightarrow T; \Sigma \mid [S] \]
\[
T ::= D \mid \text{Chan } c \]
\[
\Sigma ::= \emptyset \mid \Sigma, c: S \quad (c: S' \text{ not in } \Sigma) \]

Fig. 2. Syntax of channel types, data types, types, and channel environments

where \( S \) is \(!\text{Int}!!\text{Int}.\text{End}\). Here \text{send}Twice must be typed once with channel \( c \), and another with channel \( d \). For an example of session polymorphism, we have:

\[
\text{sendQuad} :: c: !\text{Int}!!\text{Int}!!\text{Int}.\text{End}; \text{Chan } c \rightarrow \text{Unit}; c: \text{End} \\
\text{sendQuad} x = \text{send}Twice x; \text{send}Twice x
\]

where \text{send}Twice must be typed once with \( c: !\text{Int}!!\text{Int}!!\text{Int}.\text{End} \), and a second time with \( c: !\text{Int}!!\text{Int}.\text{End} \).

3 Syntax

The syntax of our language is defined formally by the grammar in Figures 1 and 2. We use channel identifiers \( c, \ldots \), name identifiers \( n, \ldots \), term variables \( x, \ldots \), and labels \( l, \ldots \), and define values \( v \), expressions \( e \), threads \( t \), and configurations \( C \).

Most of the syntactic constructs for expressions have been illustrated in Section 2. Threads comprise stacks of expressions waiting for evaluation (cf. [23]), possibly with occurrences of the fork primitive. Configurations have four forms: a single thread, \( \langle t \rangle \); a parallel collection of threads, \( (C_1 \mid C_2) \) (we consider the parallel composition left-associative, and usually omit parentheses); declaration of a typed name, \( (\nu n: [S])C \); and declaration of a channel, \( (\nu c)C \). The last form is not part of the top-level syntax of configurations; it only arises during reduction.

Declaration of a typed name has not been illustrated yet. It shows up during
(C, |, ⟨unit⟩) is a commutative monoid (S-MONOID)
(\(\nu n: [S]|C_1 \mid C_2 \equiv (\nu n: [S])(C_1 \mid C_2)\) if \(n\) not free in \(C_2\) (S-SCOPEN)
(\(\nu c|C_1 \mid C_2 \equiv (\nu c)(C_1 \mid C_2)\) if \(c\) not free in \(C_2\) (S-SCOP C)

Fig. 3. Structural congruence

reduction of new expressions, and can also be used at the top level to give a
more realistic model of the example on page 7:

\[
\text{systemConf} = (\nu n: [&⟨\text{add} : \ldots, \text{neg} : \ldots, \text{eval} : \ldots⟩])
\]
(\(\text{negClient } n \mid \text{negClient } n \mid \text{server } n\))

This configuration arises during reduction of thread \(⟨\text{system}⟩\) defined on page 7,
but defining it in this form, at the top level, represents a situation in which
the clients and server are defined as separate components which already share
the name \(n\).

The syntax of types is described in Figure 2. We define session types \(S\), data
types \(D\), term types \(T\), and channel environments \(\Sigma\). The type \(\text{Chan } c\) represents
the type of the channel with identity \(c\); the session type associated with
\(c\) is recorded separately in a channel environment \(\Sigma\). Channel type \(\perp\),
denotes a channel that is already in use by two threads, hence that cannot be
used further. Similarly to channel and name identifiers, \(\perp\) is not available at
the top level syntax, arising only via the channel environment composition op-
erator, \(\Sigma_1 \bullet \Sigma_2\), defined in Section 5. Among datatypes we have channel-state
annotated functional types \(\Sigma; T \rightarrow T; \Sigma\), and types for names \([S]\) capable of
establishing sessions of type \(S\).

In Section 2 we used several derived constructors. An expression \(e; t\) (some-
times implied in our examples by the indentation) is an abbreviation for
\(\text{let } y = e \text{ in } t\), for \(y\) a fresh variable. Idioms like \(\text{send } (\text{receive } c)(\text{receive } c) \text{ on } c\)
need appropriate de-sugaring into consecutive \text{lets}, making the evaluation or-
der order explicit. We sometimes “terminate” threads with an expression rather than
a value: a thread \(e\) is short for \(\text{let } x = e \text{ in Unit}\). Recursive function definitions
must be made explicit with \(\text{rec}\).

To support typechecking, we annotate name declaration with its type, and \(\lambda\-
abstractions with a channel environment as well as a typed argument. Channel
declaration takes no type, for we type check top-level configurations only.

11
\[
\langle \text{let } x = \text{request } n \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{accept } n \text{ in } t_2 \rangle \rightarrow \\
(\nu c)(\langle \text{let } x = c \text{ in } t_1 \rangle \mid \langle \text{let } y = c \text{ in } t_2 \rangle) \quad \text{(R-INIT)}
\]
\[
\langle \text{let } x = \text{receive } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{send } v \text{ on } c \text{ in } t_2 \rangle \rightarrow \\
\langle \text{let } x = v \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit} \text{ in } t_2 \rangle \quad \text{(R-COM)}
\]
\[
\langle \text{let } x = \text{case } c \text{ of } \{ t_i \Rightarrow e_i \}_{i \in I} \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{select } l_j \text{ on } c \text{ in } t_2 \rangle \rightarrow \\
\langle \text{let } x = e_j \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit} \text{ in } t_2 \rangle \quad \text{(R-BRANCH)}
\]
\[
\langle \text{let } x = \text{close } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{close } c \text{ in } t_2 \rangle \rightarrow \\
\langle \text{let } x = \text{unit} \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit} \text{ in } t_2 \rangle \quad \text{(R-CLOSE)}
\]
\[
\langle \text{let } x = \text{new } S \text{ in } t \rangle \rightarrow (\nu n: [S])\langle \text{let } x = n \text{ in } t \rangle \quad \text{(R-NEW)}
\]
\[
\langle \text{fork } t_1; t_2 \rangle \rightarrow \langle t_1 \rangle \mid \langle t_2 \rangle \quad \text{(R-FORK)}
\]
\[
\langle \text{let } x = \text{if } \text{true} \text{ then } e \text{ else } e' \text{ in } t \rangle \rightarrow \langle \text{let } x = e \text{ in } t \rangle \quad \text{(R-IFT)}
\]
\[
\langle \text{let } x = \text{if } \text{false} \text{ then } e \text{ else } e' \text{ in } t \rangle \rightarrow \langle \text{let } x = e' \text{ in } t \rangle \quad \text{(R-IFF)}
\]
\[
\langle \text{let } x = (\lambda (\Sigma; y: T).e)v \text{ in } t \rangle \rightarrow \langle \text{let } x = e\{v/y\} \text{ in } t \rangle \quad \text{(R-APP)}
\]
\[
\langle \text{let } x = (\text{rec } (y: T).v)u \text{ in } t \rangle \rightarrow \langle \text{let } x = (v\{\text{rec } (y: T).v/y\})u \text{ in } t \rangle \quad \text{(R-REC)}
\]
\[
\langle \text{let } x = (\text{let } y = e \text{ in } t') \text{ in } t \rangle \rightarrow \langle \text{let } y = e \text{ in } (\text{let } x = t') \text{ in } t \rangle \quad \text{(R-LET)}
\]
\[
\langle \text{let } x = v \text{ in } t \rangle \rightarrow \langle \text{let } v/x \rangle \quad \text{(R-BETA)}
\]
\[
\frac{C \rightarrow C'}{(\nu e)C \rightarrow (\nu e)C'} \quad \text{(R-NewC)}
\]
\[
\frac{C \rightarrow C'}{(\nu n: [S])C \rightarrow (\nu n: [S])C'} \quad \text{(R-NewN)}
\]
\[
\frac{C \rightarrow C'}{C \mid C'' \rightarrow C' \mid C''} \quad \text{(R-Par)}
\]
\[
\frac{C \equiv \equiv C'}{C \rightarrow C'} \quad \text{(R-Cong)}
\]

In R-INIT, \(c\) is not free in \(t_1, t_2\); in R-NEW, \(n\) is not free in \(t\).

Fig. 4. Reduction rules

4 Operational Semantics

The binding occurrences are the variable \(x\) in \(\lambda (\Sigma; x: T).e\), in \(\text{rec } (x: T).e\), and in \(\text{let } x = e \text{ in } t\), the name \(n\) in \((\nu n: [S])C\), and the channel \(c\) in \((\nu e)C\). Free

and bound identifiers are defined as usual and we work up to \(\alpha\)-equivalence.

We define a reduction semantics on configurations (Figure 4), making use

of a simple structural congruence relation [24] (Figure 3), allowing for the

rearrangement of threads in a configuration, so that reduction may happen.\(^4\)

Substitution of values for variables is defined as expected.

\(^4\) We could easily arrange for structural congruence to garbage collect all threads

of the form \(\langle v \rangle\), for \(v\) closed.
R-INIT synchronises two threads on a *shared name* \( n \), creating a new channel \( c \) known to both threads. Rules R-COM, R-BRANCH, and R-CLOSE synchronised two threads on a *channel* \( c \): R-COM transmits a value \( v \) from one thread to the other; R-BRANCH, rather than transmitting a value, chooses one of the branches in the *case* thread; and R-CLOSE closes a channel in *both* threads simultaneously. R-NEW creates a new name \( n \), and records the fact that the name is potentially shared, by means of a \((vn: [S])\) in the resulting configuration. The last four rules allow reduction to happen underneath restriction, parallel composition, and structural congruence.

Unlike other thread models, the value a thread reduces to is not communicated back to its parent thread (the one that forked the terminating thread). Such behaviour would have to be explicitly programmed by arranging for both threads to share a channel and explicitly sending the result back to the parent.

**Example 1** We follow the execution of thread \( \langle \text{system} \rangle \) on page 7. Types play no role in reduction; we omit them.

\[
\langle \text{system} \rangle \quad (1)
\]
\[
\rightarrow^* (vn) \quad \langle \text{fork negClient } n; \text{fork addClient } n; \text{fork server } n \rangle \quad (2)
\]
\[
\rightarrow^* (vn)( \langle \text{let } u = \text{request } n \text{ in } \ldots \rangle \mid \langle \text{let } u = \text{request } n \text{ in } \ldots \rangle \mid \langle \text{let } u = \text{accept } n \text{ in } \ldots \rangle) \quad (3)
\]
\[
\rightarrow^* (vn)( \langle \text{let } u = \text{request } n \text{ in } \ldots \rangle \mid \langle \nu c \rangle(\langle \text{select neg on } c; \ldots \rangle \mid \langle \text{case } c \text{ of } \ldots \rangle)) \quad (4)
\]
\[
\rightarrow^* (vn)( \langle \text{let } u = \text{request } n \text{ in } \ldots \rangle \mid \langle \nu c \rangle(\langle \text{close } c \rangle \mid \langle \text{close } c \rangle)) \quad (5)
\]
\[
\rightarrow^* (vn) \quad \langle \text{let } u = \text{request } n \text{ in } \ldots \rangle \quad (6)
\]

In line (3) we have two threads competing for requesting a session on name \( n \). We have chosen the negClient to go ahead; a new channel \( c \), known only to the negClient and to the server, is created. In line (4) we have a typical interaction between two threads on a common channel. Similar interactions continue until both channel’ ends are ready to be closed, in line (5). Once closed, structural congruence allows to get rid of the terminated threads. The run ends in line (6) with the addClient still waiting for a server which will never come.

5 **Typing**

The type system is presented in Figures 6–9. Typing judgements for constants are of the form \( \Gamma \vdash v : T \), where \( \Gamma \) is a map from names and variables to types.
\[ \text{End} = \text{End} \quad \forall D.S = !D.S \quad \forall S'.S = !S'.S \quad \forall D.S = ?D.S \quad \forall S'.S = ?S'.S \]
\[ \&\langle l_i; S_i \rangle_{i \in I} = \oplus\langle l_i; S_i \rangle_{i \in I} = \&\langle l_i; S_i \rangle_{i \in I} \]

Fig. 5. Duality on session types

\[ \Gamma \vdash \text{true}: \text{Bool} \quad \Gamma \vdash \text{false}: \text{Bool} \quad \Gamma \vdash \text{unit}: \text{Unit} \quad (\text{T-Const}) \]
\[ \Gamma \vdash c: \text{Chan} \ c \quad (\text{T-Chan}) \]
\[ \Gamma, n: [S] \vdash n: [S] \quad (\text{T-Name}) \]
\[ \Gamma, x: T \vdash x: T \quad (\text{T-Var}) \]
\[ \Gamma, x: T \vdash \Sigma \triangleright e: U \triangleleft \Sigma' \forall c.\Sigma(c) \neq \bot \]
\[ \Gamma \vdash \lambda (\Sigma; x: T).e: (\Sigma; T \rightarrow U; \Sigma') \quad (\text{T-Abs}) \]
\[ \Gamma, x: T \vdash v: T \quad T = (\Sigma; U \rightarrow U'; \Sigma') \]
\[ \Gamma \vdash \text{rec} (x: T).v: T \quad (\text{T-Rec}) \]

Fig. 6. Typing rules for values

Value judgements do not mention channel environments, for values, having no behaviour, do not change channels. Judgements for expressions are of the form \( \Gamma \vdash \Sigma \triangleright e: U \triangleleft \Sigma' \), where \( \Sigma \) is the channel environment of Figure 2, viewed as a map from channels into session types. The difference between \( \Sigma \) and \( \Sigma' \) reflects the effect of an expression on the types of channels. For example

\[ x: \text{Chan} \ c \vdash !\text{Int}. \text{End} \triangleright \text{receive} \ x: \text{Int} \triangleleft c: \text{End}. \]

Finally, typing judgements for configurations are of the form \( \Delta \vdash \Sigma \triangleright C \) where \( \Delta \) is a map from names to name types of the form \([S]\).

**Typing Values (Figure 6).** \( \text{T-Chan} \) says that a channel named \( c \) has type \( \text{Chan} \ c \). The actual type (or state) of channel \( c \) is to be found in a channel environment \( \Sigma \), in the rules for expressions. In \( \text{T-Abs} \), the initial and final channel environments of the function body are recorded in the function type. The condition \( \forall c.\Sigma(c) \neq \bot \) specifies that a function can only use one end of each channel. In the discussion of rule \( \text{T-Fork} \) (Figure 8), later, we explain that a channel \( c: \bot \) arises when an expression uses both ends of \( c \) with dual types; we do not allow abstraction on expressions of this kind. The reason for this restriction is to obtain completeness of our type checking algorithm (Section 7); the type checker requires that the initial environment does not contain \( \bot \).
Fig. 7. Typing rules for expressions I: Channel operations

\[\Gamma \vdash v : [S] \quad (\text{T-REQUEST})\]
\[\Gamma \vdash \Sigma \triangleright request v : Chan c \triangleleft \Sigma, c : \overline{S} \quad (\text{T-REQUEST})\]
\[\Gamma \vdash v : [S] \quad (\text{T-REQUEST})\]
\[\Gamma \vdash \Sigma \triangleright accept v : Chan c \triangleleft \Sigma, c : \overline{S} \quad (\text{T-REQUEST})\]
\[\Gamma \vdash v : T \quad (\text{T-TO})\]
\[\Gamma \vdash v : (\Sigma ; T \to U ; \Sigma') \quad \Gamma \vdash v' : T \quad (\text{T-APP})\]
\[\Gamma \vdash \Sigma \triangleright v \triangleright v' : U \triangleleft \Sigma' \quad (\text{T-NEW})\]
\[\Gamma \vdash v : \text{Bool} \quad \Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma' \quad \Gamma \vdash \Sigma \triangleright e' : T \triangleleft \Sigma' \quad (\text{T-LET})\]
\[\Gamma \vdash \Sigma \triangleright \text{let } x = e \text{ in } t : U \triangleleft \Sigma' \quad (\text{T-POLYLET})\]
\[\Gamma \vdash \Sigma \triangleright \text{let } x = v \text{ in } t : U \triangleleft \Sigma' \quad (\text{T-FORK})\]
\[\Gamma \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \text{fork } t_1 ; t_2 : T_2 \triangleleft \emptyset \quad (\text{T-FORK})\]

Fig. 8. Typing rules for expressions II: Other rules

**Typing Expressions (Figures 7, 8).** There are two rules for `receive` and two rules for `send`, for these constructors are overloaded: they allow transmission of data as well as channels. In T-RECEIVE-D, the prefix `?D`, of the type for channel `c`, is consumed, provided that we are receiving on a value aliased to channel `c` (of type `Chan c`). In T-RECEIVE-S, we receive a channel, that we
decided to call $d$; the type of the expression is $\text{Chan } d$, and we add a new entry to the final channel environment, where we record the type for $d$. The rules $\text{T-SendD}$ and $\text{T-SendS}$, for sending values and channels, are similar. In $\text{T-Select}$, the type for $c$ in the final channel environment is that of branch $l_j$ in the type for $c$ in the source channel environment. In $\text{T-Case}$, all branches must produce the same final channel environment. This enables us to know the environment for any code following the $\text{case}$, independently of which branch is chosen at runtime. The same applies to the two branches of the conditional in $\text{T-If}$. Rule $\text{T-Close}$ requires that the channel must be ready to be closed (of type $\text{End}$). We remove the closed channel from the environment.

In Figure 8, rules $\text{T-Request}$ and $\text{T-Accept}$ both introduce a new channel $c$ in the channel environment, of dual polarities [13, 19, 20, 32, 33]. The dual of a session type $S$, denoted $\overline{S}$, is defined for all session types except ⊥, and is obtained by interchanging output ! and input ?, and by interchanging branching & and selection $\oplus$, and leaving $S$ otherwise unchanged. The inductive definition of duality is in Figure 5.

Rule $\text{T-Val}$ says that constants do not affect the state of channels. In $\text{T-App}$, the initial and final channel environments in the type of the function are released into the typing for the application. Rule $\text{T-New}$ specifies that the expression $\text{new } S$ has type $[S]$, denoting a name which, when shared by two threads, is able to produce (via accept/request) new channels of type $S$.

In rule $\text{T-Let}$, notice how the control flow is recorded in the various channel environments: from $\Sigma$ to $\Sigma''$ in the evaluation of $e$, and from $\Sigma''$ to $\Sigma'$ in the evaluation of $t$, so that the whole expression takes $\Sigma$ into $\Sigma'$. Rule $\text{T-PolyLet}$ types the various forms of polymorphism identified in Section 2, by first substituting $v$ for $x$ in $t$, so that each copy of the polymorphic value $v$ is typed independently [28, Chapter 22].

Rule $\text{T-Fork}$ composes the initial channel environments of two configurations, by checking that the types of the channels occurring in both environments are dual. As for the final environment, the rule requires, via the empty channel environments in both antecedents, that each thread involved consumes its channels (that is, sends or closes them).

The composition of two channel environments, $\Sigma_1 \bullet \Sigma_2$, is defined only when $\Sigma_1(c) = \Sigma_2(c)$, for all $c \in \text{dom } \Sigma_1 \cap \text{dom } \Sigma_2$. In this case $\text{dom}(\Sigma_1 \bullet \Sigma_2) = \text{dom } \Sigma_1 \cup \text{dom } \Sigma_2$, and $(\Sigma_1 \bullet \Sigma_2)(c)$ is ⊥ when $c \in \text{dom } \Sigma_1 \cap \text{dom } \Sigma_2$, and is $\Sigma_i(c)$ when $c \in \text{dom } \Sigma_i \setminus \text{dom } \Sigma_{3-i}$, for $i = 1, 2$.

**Typing Configurations (Figure 9).** Rule $\text{T-Thread}$ requires threads to consume their channels, similarly to $\text{T-Fork}$. The $\Delta$ in the antecedent of rule
Fig. 9. Typing rules for configurations

T-Thread ensures that threads are closed for variables, for the domain of $\Delta$ does not include variables. Rule T-Par is similar to T-Fork. T-NewN discards information on the bound name. There are two rules for channel creation. Rule T-NewB says that a newly created channel must be used with dual modes by exactly two threads, since the type $\perp$ arises from the $\bullet$ operator in rules T-Par or T-Fork. Rule T-NewC means that a configuration remains typable if, during reduction, one of its channels is consumed.

In the original version of this paper [34], the typing rules T-Thread and T-Fork were more general, allowing the typing judgements in the hypotheses to have non-empty final channel environments in which all of the types are $\perp$. The present form of the rules means that typing guarantees a stronger property. A thread consumes (either by completely using and then closing, or by sending on a channel) every channel which it possesses or creates. A top-level configuration completely uses, and then closes, every channel which it creates. To be more precise, typing guarantees that code is present which can use and close every channel; it is of course possible that at runtime some channels may be left unclosed because of unreachable code or non-termination.

The formulation of Subject Reduction is standard; the proof is in Appendix A, page 28.

**Theorem 2 (Subject Reduction)** If $\Delta \vdash \Sigma \triangleright C$ and $C \rightarrow C'$, then $\Delta \vdash \Sigma' \triangleright C'$, where $\Sigma' \subseteq \Sigma$.

6 Type Safety

In our language of functional communicating threads different sorts of problems may occur at runtime, ranging from the traditional error of testing, in a
conditional expression, a value that is not `true` or `false`; through applying `close` to a value that is not a channel; to the most relevant to our work: having one thread trying to `send` on a given channel, and another trying to `select` on the same channel, or having three or more threads trying to synchronize on the same channel.

In order to define what we mean by a faulty configuration, we start by calling a `c-thread` any thread ready to perform an operation on channel `c`, that is a thread of the form `<let x = receive c in t>`, and similarly for `send`, `case`, `select`, and `close`. A `c-redex` is the parallel composition of two threads ready to communicate on channel `c`, that is `<let x = send v on c in t_1> | <let y = receive c in t_2>`, and similarly for `case/select`, `close/close`. A configuration `C` is faulty when `C ≡ (νc)(νn: [S])((ν⃗ c)(ν⃗ n: [S]))(C_1 | C_2)` and `C_1` is

1. the thread `<let x = e in t>`, where `e` is i) if `v then _ else _` with `v ≠ true, false`, or is ii) `v_` with `v ≠ λy.e’` and `v ≠ rec y.e’`; or is
2. the thread `<let x = accept/request v in t>`, where `v` is not a name; or is
3. the thread `<let x = e in t>`, where `e` is i) `receive/close v`, or ii) `send _ on v`, or iii) `case v of _`, or iv) `select _ on v`, with `v` not a channel; or is
4. the parallel composition of two `c-threads` that do not form a `c-redex`; or is
5. the parallel composition of three or more `c-threads`.

The main property of this section says that typable configurations are not faulty; the proof is in Appendix B, page 38.

**Theorem 3 (Type Safety)** Typable configurations are not faulty.
7 Type Checking

This section presents a type checking algorithm in the form of typing rules. We define type checking for values (Figure 10), expressions (Figures 11 and 12).
\[ \Delta; \Sigma; t \mapsto \Sigma'; T; \emptyset \]  
\[ \Delta; \Sigma; \langle t \rangle \mapsto \Sigma' \]  
\[ \Delta; \Sigma; C_1 \mapsto \Sigma_1, \Delta; \Sigma_1; C_2 \mapsto \Sigma_2 \]  
\[ \Delta; \Sigma; C_1 \mid C_2 \mapsto \Sigma_2 \]  
\[ \Delta, n: [S]; \Sigma; C \mapsto \Sigma' \]  
\[ \Delta; \Sigma; (\nu n: [S])C \mapsto \Sigma' \]  
\[ \Delta; \Sigma; \nu n: [S]; \Sigma; C \mapsto \Sigma' \]  
\[ \Delta; \Sigma; \nu n: \Sigma' \mapsto \Sigma' \]  
\[ \Delta; \Sigma; (\nu n: \Sigma')C \mapsto \Sigma' \]  

Fig. 13. Type checking configurations

and configurations (Figure 13).

We apply the type checking algorithm only to top-level configurations, that is configurations that do not contain the \((\nu c)C\) construct.\(^5\) This restriction implies that the type checking algorithm (for both configurations and expressions) is only needed for channel environments \(\Sigma\) such that \(\forall c. \Sigma(c) \neq \bot\).

Typically the algorithm is called at the top level with \(\Sigma = \emptyset\), for channels are not part of the top-level syntax. Type-checking the full calculus represents an increase in complexity without apparent benefits.

Type checking values is defined by inference rules for judgements of the form \(\Gamma; v \mapsto T\), where \(\Gamma\) and \(v\) describe the input, and \(T\) describes the output of the type checking function. Judgements for expressions are of the form \(\Gamma; \Sigma; e \mapsto \Sigma'; T; \Sigma''\), where the symbols at the left of the arrow represent the input, and those on the right the output. Channel environment \(\Sigma'\) describes the unused part of \(\Sigma\), whereas \(\Sigma''\) represents the final types of the channels which are used by \(e\). Channels which are created by \(e\) appear in \(\Sigma''\). If \(c: S \in \Sigma\) then we will have either \(c: S \in \Sigma'\) if \(e\) does not use \(c\), or \(c: S' \in \Sigma''\) if \(e\) uses \(c\) and leaves it with type \(S'\), or neither if \(S = \text{End}\) and \(e\) closes \(c\). For example

\[ x: \text{Chan}; c: ?!\text{Int}; d: !!\text{Int}; \text{receive} \ x \mapsto d: !!\text{Int}; \text{Int}; c: \text{End}. \]

Finally, judgements for configurations are of the form \(\Delta; \Sigma; C \mapsto \Sigma'\) where \(\Sigma'\) describes the unused part of \(\Sigma\).

The type checking algorithm is presented in Figures 10–13. Most of the rules are obtained from those in Figures 6–9, by a suitable reading of the sequents.

**Type Checking Values (Figure 10).** Now that we assume, globally, that the type checking algorithm is only called with a \(\Sigma\) which does not contain any

\(^5\) Such configurations only arise during reduction, and there is no need to use the type checking algorithm on them, because the Subject Reduction Theorem guarantees that they are typable.
⊥ types, rule C-Abs implicitly contains this assumption, matching the explicit assumption in rule T-Abs. This means that a function can only use one end of a channel. The following lemma (Lemma 19, Appendix C) guarantees that \( \Sigma_1 \) and \( \Sigma_2 \) have disjoint domains, so that \( \Sigma_1, \Sigma_2 \) is defined and the overall transformation of channel types can be described by the function type \( \Sigma; T \rightarrow U; \Sigma_1, \Sigma_2 \).

**Lemma 19.** If \( \Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2 \) and \( \Sigma \) contains no \( \bot \) types, then \( \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) = \emptyset \).

**Type Checking Expressions (Figures 11, 12).** Most of the rules are straightforward. In Figure 12, contrast rule C-New with rule C-App. Since values do not change channels, the former rule consumes nothing, placing the input \( \Sigma \) in the unused position in the output. The latter consumes channels as described by \( \Sigma \) in the type for the function, leaving empty the unused component in the output.

The most complex type checking rule is C-Let. The channel environment \( \Sigma_1, \Sigma'_1 \) which is used when typechecking \( t \) consists of the channels (\( \Sigma_1 \)) which are not used by \( e \) and the updated types (\( \Sigma'_1 \)) of the channels which are used by \( e \). In the conclusion, the unmodified channel environment \( \Sigma_1 \cap \Sigma_2 \) consists of the channels which are used by neither \( e \) nor \( t \). The modified channel environment, \( (\Sigma'_1 \cap \Sigma_2), \Sigma'_2 \), consists of channels which are modified by \( e \) but not by \( t \) (\( \Sigma'_1 \cap \Sigma_2 \)) and channels which are modified by \( t \) (\( \Sigma'_2 \)).

Rule C-Fork requires that \( t_1 \) and \( t_2 \) fully consume all of the channels that they use, either by using them completely and closing them, or by sending them to other threads. This corresponds to the hypotheses of rule T-Fork. Of the initial channels \( \Sigma \), some are consumed by \( t_1 \) and the remainder, \( \Sigma_1 \), are used to typecheck \( t_2 \). Any channels remaining after typechecking \( t_2 \), i.e. \( \Sigma_2 \), are returned as the channels which are not used by the fork expression.

**Type Checking Configurations (Figure 13).** In rule C-Par, we run the algorithm on \( C_1 \) and feed the output \( \Sigma_1 \), the unused part of \( \Sigma \), into another call, this time for \( C_2 \). The output \( \Sigma_2 \) of the second call is the unused channel environment of the parallel composition \( C_1 \parallel C_2 \) (cf. rule C-Fork in Figure 12).

**Correctness.** The type checking algorithm is sound and complete with respect to the type system presented in Section 5. To state soundness we define
the partial operation $\Sigma - \Sigma'$ as follows. $\Sigma - \Sigma'$ is defined if and only if $\Sigma' \subseteq \Sigma$, and in this case, $\Sigma - \Sigma' = \{ c : S | c : S \in \Sigma \text{ and } c : S \not\in \Sigma' \}$.

The formulation of correctness is standard; the proofs are in Appendices C (page 40), and D (page 44).

**Theorem 4 (Soundness)** If $\Delta; \Sigma; C \rightarrow \Sigma'$, then $\Sigma - \Sigma'$ is defined and $\Delta \vdash \Sigma - \Sigma' \triangleright C$.

**Theorem 5 (Completeness)** If $\Delta \vdash \Sigma \triangleright C$ without using rule T-POLYLET, then for all $\Sigma'$ such that $\Sigma \triangleright \Sigma'$ is defined and does not contain $\bot$, $\Delta; \Sigma \triangleright \Sigma'; C \rightarrow \Sigma'$.

8 Related Work

*Cyclone* [18] is a C-like type-safe polymorphic imperative language. It features region-based memory management, and more recently threads and locks [17], via a sophisticated type system. The multithreaded version requires “a lock name for every pointer and lock type, and an effect for every function”. Our locks are channels; but more than mutual exclusion, channels also allow a precise description of the protocol “between” acquiring and releasing the lock. In Cyclone a thread acquires a lock for a resource, uses the resource in whichever way it needs, and then releases the lock. Using our language a thread acquires the lock via a *request* operation, and then follows a specified protocol for the resource, before closing the channel obtained with *request*.

In the *Vault* system [8] 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. “Adoption and Focus” [10], by the same authors, is a type system able to track changes in the state of objects; the system handles aliasing, and includes a form of polymorphism in functions. In contrast, our system checks the types of individual messages, as well as tracking the state of the channel. Our system is more specialized, but the specialization allows more type checking in the situation that we handle.

*Type and effect* systems can be used to prove properties of protocols. Gordon and Jeffrey [16] use one such system to prove progress properties of communication protocols written in the $\pi$-calculus. Bonelli, Compagnoni, and Gunter [2, 3] combine the language of Honda, Vasconcelos and Kubo [20] with
the correspondence assertions of Gordon and Jeffrey, thus obtaining a setting where further properties can be proved about programs. Adding correspondence assertions to session types increases the expressiveness of the system in two ways. Although session types only specify the structure of interactions between pairs of participants of a possibly multiparty protocol, the new setting makes it possible to specify and check that the interactions between participants in different pairs respect the overall protocol. Furthermore, the integrity and correct propagation of data is also verifiable. However, this is a different kind of extension of session types than our work; their language does not include function types.

Rajamani et al.’s Behave [6, 29] uses CCS to describe properties of $\pi$-calculus programs, verified via a combination of type and model checking. Since our system is purely type checking (not model checking) verification is more efficient and easier to implement. Igarashi and Kobayashi have developed a generic framework in which a range of $\pi$-calculus type systems can be defined [22]. Although able to express sequencing of input and output types similarly to session types, it cannot express branching types.

A somewhat related line of research addresses resource access in general. Walker, Crary, and Morrisett [36] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [21] present a general framework comprising a language with primitives for creating and accessing resources, and a type inference algorithm that checks whether programs access resources in a disciplined manner. Although types for resources in this latter work are similar in spirit to session types, we work in a much simpler setting.

Neubauer and Thiemann encoded a version of session types in the Haskell programming language, and proved that the embedding preserves typings [25], but the results are limited to type soundness.

Very recently, Dezani-Ciancaglini et al. [9] have proposed a minimal distributed object-oriented language with session types. Apart from the use of objects, the main difference between their version of session types and ours seems to be that they do not allow channels to be sent along channels (although objects containing names of potential channels may be sent). A more detailed comparison is a subject for future work.

9 Future Work

A prototype implementation of the language, targeted at the Multithreaded Intermediate Language [26], is under way. We outline some of the issues in-
involved in extending our language to include a wider range of standard features.

**Recursive Session Types.** We have introduced recursive session types in a previous work [15]. We feel its incorporation in the present setting would not present major difficulties.

**Principal Typings.** For practical type inference, for separate compilation and modularity, one needs a notion of principal typings for the language. Particularly challenging is the share/not-share kind of polymorphism identified in Section 2.

**Type Inference.** We are working on a constraint-based type inference algorithm for (the monomorphic fragment of) the language.

**Web services.** Our work opens up the possibility of an application of session types to verification of web service implementations [7,35]. Web services require a model for business interactions, which typically assume the form of sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. Although some rigorous semantics have been developed (eg. [4]), there is still little assistance with the verification of the correctness of the protocol descriptions and their composition (eg. [5,11]). Session types may provide a useful static analysis tool.

**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.** This work was partially supported by FEDER, the EU IST proactive initiative FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100), Fundação para a Ciência e a Tecnologia (via CLC, CITI, and the project MIMO, POSI/CHS/39789/2001), and a Treaty of Windsor grant from the British Council in Portugal and the Portuguese Council of University Rectors.
References


25


A Proof of Theorem 2, Subject Reduction

We start with a few auxiliary results; the proof of Subject Reduction is on page 32. To simplify the proofs, we make use of the variable convention [1], allowing, for example, to assume that, in sequent $\Delta \vdash \Sigma \triangleright (\nu c)C$, channel $c$ does not occur in either $\Delta$ or $\Sigma$. Relatedly, when we say that $c$ does not occur in $C$, we mean that it does not occur free in $C$ and, by the variable convention, that it does not occur bound either.

The following easy results allow to grow and shrink the variable environment of an expression. Weakening is used in Subject Reduction (rule R-Let) and narrowing in the Substitution Lemma 15.\(^6\)

**Lemma 6 (Variable Weakening)** Suppose that $x$ does not occur in $e, v$.

1. If $\Gamma \vdash \Sigma \triangleright e: U \triangleleft \Sigma'$, then $\Gamma, x: T \vdash \Sigma \triangleright e: U \triangleleft \Sigma'$.
2. If $\Gamma \vdash v: U$, then $\Gamma, x: T \vdash v: U$.

**PROOF.** The proofs, by mutual induction on the derivation of the judgements, are straightforward.

**Lemma 7 (Variable Narrowing)** Suppose that $x$ does not occur in $e, v$.

1. If $\Gamma, x: T \vdash \Sigma \triangleright e: U \triangleleft \Sigma'$, then $\Gamma \vdash \Sigma \triangleright e: U \triangleleft \Sigma'$.
2. If $\Gamma, x: T \vdash v: U$, then $\Gamma \vdash v: U$.

**PROOF.** The proofs follow the pattern of the ones above.

The following two unchallenging results allow to grow and shrink, this time, the name environment of a configuration. They are used in the proofs of Subject Congruence (rule S-SCOPE\(N\)) and Subject Reduction (rule R-NEW).

**Lemma 8 (Name Weakening)** Suppose that $n$ does not occur in $C, e, v$.

1. If $\Delta \vdash \Sigma \triangleright C$, then $\Delta, n: [S] \vdash \Sigma \triangleright C$.
2. If $\Gamma \vdash \Sigma \triangleright e: T \triangleleft \Sigma'$, then $\Gamma, n: [S] \vdash \Sigma \triangleright e: T \triangleleft \Sigma'$.
3. If $\Gamma \vdash v: T$, then $\Gamma, n: [S] \vdash v: T$.

\(^6\) In the formulation of the lemma, we have omitted the hypothesis that $x$ is not in the domain of $\Gamma$ (for otherwise $\Gamma, x: T$ would not be defined in the conclusion). We henceforth follow this convention for all sorts of environments.
**PROOF.** The proof for configurations is a straightforward induction on the derivation of the judgement, using the result for expressions when the last rule in the derivation tree is T-Thread. The proofs for expressions and for values are by mutual induction.

**Lemma 9 (Name Narrowing)** Suppose that \( n \) is not in \( C, e, v \).

1. If \( \Delta, n: [S] \vdash \Sigma \triangleright C \), then \( \Delta \vdash \Sigma \triangleright C \).
2. If \( \Gamma, n: [S] \vdash \Sigma \triangleright e: T \triangleleft \Sigma' \), then \( \Gamma \vdash \Sigma \triangleright e: T \triangleleft \Sigma' \).
3. If \( \Gamma, n: [S] \vdash v: T \), then \( \Gamma \vdash v: T \).

**PROOF.** The proofs follow the pattern of the ones above.

The following two results allow to shrink the final channel environment of an expression, and to conclude that channels in the domain of a channel environment occur free in a configuration. Both results are needed in Subject Congruence (channel extrusion using rule T-NewB).

**Lemma 10 (Channel Narrowing on expressions and values)** Suppose that \( c \) does not occur in \( e, v, T \).

1. If \( \Gamma \vdash \Sigma, c: S \triangleright e: T \triangleleft \Sigma' \), then \( \Sigma' = \Sigma'', c: S \) and \( \Gamma \vdash v: (\Sigma; T \rightarrow U; \Sigma'') \).
2. If \( \Gamma \vdash v: (\Sigma, c: S; T \rightarrow U; \Sigma') \), then \( \Sigma = \Sigma'', c: S \) and \( \Gamma \vdash v: (\Sigma; T \rightarrow U; \Sigma'') \).

**PROOF.** The proofs, by mutual induction on the derivation of the judgement, are straightforward.

**Lemma 11 (Free channels in configurations)** If \( \Delta \vdash \Sigma, c: S \triangleright C \), then \( c \) occurs free in \( C \).

**PROOF.** Straightforward induction on the derivation of the judgement. For T-Thread, the key is that Lemma 10 implies that \( \Delta \vdash \Sigma, c: S \triangleright t: T \triangleleft \emptyset \) cannot hold unless \( c \) occurs free in \( t \). The other cases are easy.

The following result accounts for the monoidal structure of configurations; it is used in the proof of Subject Congruence.

**Lemma 12 (Channel environment monoid)** Consider the commutative monoid axioms expressed in terms of \( (\Sigma, \triangleright, \emptyset) \), each in the form \( \text{LHS} = \text{RHS} \). For each axiom, \( \text{LHS} \) is defined if and only if \( \text{RHS} \) is defined, and then they are equal.
PROOF. Directly from the definition of channel environment composition, on page 16.

Congruent configurations share the same typings. This result is used in the proof of Subject Reduction, rule R-CONF.

Lemma 13 (Subject Congruence) If \( \Delta \vdash \Sigma \triangleright C \) and \( C \equiv C' \), then \( \Delta \vdash \Sigma \triangleright C' \).

PROOF. The proof proceeds by induction on the derivation of \( C \equiv C' \). The inductive cases (the congruence rules) are straightforward. We now consider the base cases.

When the last rule applied is the commutative monoid rule, we use Lemma 12. For the scope extrusion rules S-SCOPE\textsubscript{N} and S-SCOPE\textsubscript{C} we must consider each rule in both directions; for S-SCOPE\textsubscript{C} we must consider two cases, depending on whether the typing derivation uses T-NEW\textsubscript{B} or T-NEW\textsubscript{C}.

S-SCOPE\textsubscript{N}. When reading the rule left-to-right we use name weakening (Lemma 8). In the other direction we use name narrowing (Lemma 9). In both cases, we use the hypothesis (in the congruence rule) that \( n \) is not in \( C_2 \).

S-SCOPE\textsubscript{C}, left-to-right, T-NEW\textsubscript{B}. By hypothesis, we have

\[
\begin{align*}
\Delta \vdash \Sigma_1, c: \bot \triangleright C_1 & \\
\text{c not in } \Delta, \Sigma_1 & \\
\Delta \vdash (\nu c)C_1 & \quad \text{T-NEW\textsubscript{B}} \\
\Delta \vdash \Sigma_1 \triangleright C_2 & \\
\Delta \vdash \Sigma_1 \cdot \Sigma_2 \triangleright (\nu c)C_1 \mid C_2 & \quad \text{T-PAR}
\end{align*}
\]

From the assumptions in the above tree, we build the following derivation, where we crucially use the variable convention to ensure that \( c \) is not in \( \Sigma_2 \).

\[
\begin{align*}
\Delta \vdash \Sigma_1, c: \bot \triangleright C_1 & \\
\Delta \vdash \Sigma_2 \triangleright C_2 & \\
\Delta \vdash \Sigma_1 \cdot \Sigma_2, c: \bot \triangleright C_1 \mid C_2 & \quad \text{T-PAR} \\
\text{c not in } \Delta, \Sigma_1, \Sigma_2 & \\
\Delta \vdash (\nu c)(C_1 \mid C_2) & \quad \text{T-NEW\textsubscript{B}}
\end{align*}
\]

S-SCOPE\textsubscript{C}, left-to-right, T-NEW\textsubscript{C}. Similar to the previous case, again using the variable convention.

S-SCOPE\textsubscript{C}, right-to-left, T-NEW\textsubscript{B}. By hypothesis, we have a proof tree of
the form:

\[
\frac{\Delta \vdash \Sigma_1 \triangleright C_1 \quad \Delta \vdash \Sigma_2 \triangleright C_2}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c: \bot \triangleright C_1 \mid C_2} \quad \text{T-PAR}
\]

\[
\frac{c \text{ not in } \Delta, \Sigma_1, \Sigma_2}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2) \triangleright (\nu c)(C_1 \mid C_2)} \quad \text{T-NEWB}
\]

We analyse the possibilities for splitting environment \((\Sigma_1 \bullet \Sigma_2), c: \bot\). There are three cases.

1. \(\Sigma_1^* = \Sigma_1, c: \bot\) and \(\Sigma_2^* = \Sigma_2\).
2. \(\Sigma_1^* = \Sigma_1\) and \(\Sigma_2^* = \Sigma_2, c: \bot\).
3. \(\Sigma_1^* = \Sigma_1, c: S\) and \(\Sigma_2^* = \Sigma_2, c: S\).

In case 1 we build the following derivation.

\[
\frac{\Delta \vdash \Sigma_1, c: \bot \triangleright C_1 \quad c \text{ not in } \Delta, \Sigma_1}{\Delta \vdash \Sigma_1 \triangleright (\nu c)C_1} \quad \text{T-NEWB}
\]

\[
\Delta \vdash \Sigma_2 \triangleright C_2 \quad \frac{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c)C_1 \mid C_2}{\text{T-PAR}}
\]

In case 2 we build the following derivation.

\[
\frac{\Delta \vdash \Sigma_1 \triangleright C_1 \quad c \not\in \Delta, \Sigma_1}{\Delta \vdash \Sigma_1 \triangleright (\nu c)C_1} \quad \text{T-NEWC}
\]

\[
\frac{\Delta \vdash \Sigma_2, c: \bot \triangleright C_2 \quad c \not\in \Delta, \Sigma_2, C_2}{\Delta \vdash \Sigma_2 \triangleright C_2} \quad \text{Lem 11}
\]

\[
\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c)C_1 \mid C_2 \quad \text{T-PAR}
\]

In case 3 the typing derivation gives us the additional hypothesis that \((\Sigma_1, c: S) \bullet (\Sigma_2, c: S)\) is defined. As we have \(\Delta \vdash \Sigma_2, c: S \triangleright C_2\) and we know that \(c\) is not in \(C_2\), the contrapositive of Lemma 11 gives \(\Delta \vdash \Sigma_2 \triangleright C_2\), contradicting the assumption that \((\Sigma_1, c: S) \bullet (\Sigma_2, c: S)\) is defined. Therefore this case cannot arise.

**S-ScopeC, right-to-left, T-NewC.** Similar to case 1 of the previous argument.

The following result allows to replace a given channel with another one, throughout a derivation tree. We use it in Subject Reduction, rule R-INIT, to unify the two fresh channels in the hypothesis.

**Lemma 14 (Channel replacement)** Suppose that \(d\) does not occur in any of \(\Gamma, \Sigma, \Sigma', T, e, v\), and \(c\) does not occur in \(e, v\).
(1) If $\Gamma \vdash \Sigma \vdash e : T \triangleleft \Sigma', \text{ then } \Gamma\{d/c\} \vdash \Sigma\{d/c\} \vdash e : T\{d/c\} \triangleleft \Sigma'\{d/c\}$. 

(2) If $\Gamma \vdash v : T$, then $\Gamma\{d/c\} \vdash v : T\{d/c\}$.

**PROOF.** The proof of the two results, by mutual induction on the derivation of the judgements, is straightforward.

The following lemma accounts for all cases in Subject Reduction where substitution is needed, namely, in rules R-APP, R-REC, and R-BETA.

**Lemma 15 (Substitution)** Suppose that $\Gamma \vdash v : T$.

(1) If $\Gamma, x : T \vdash \Sigma \vdash e : U \triangleleft \Sigma'$ then $\Gamma \vdash \Sigma \vdash e\{v/x\} : U \triangleleft \Sigma'$.

(2) If $\Gamma, x : T \vdash u : U$ then $\Gamma \vdash u\{v/x\} : U$.

**PROOF.** The proof of the two results is by mutual induction on the derivation of the judgement.

1. **Expressions.** The result follows easily using the result for values and induction.

2. **Values.** The cases of rules T-CONST, T-CHAN, and T-NAME follow easily, observing that $x$ does not occur in $u$, and applying Lemma 7. The case of rule T-VAR follows trivially, as $u = x$. The case of rule T-ABS uses the result for expressions, and that of rule T-REC follows by induction.

We are finally in a position to prove Subject Reduction.

**PROOF.** [Theorem 2, page 17] The proof proceeds by induction on the derivation of $C \rightarrow C'$. We analyse each reduction rule in Figure 4, page 12, in turn.

**R-Init.** By hypothesis, we have

$$\langle\text{let } x = \text{request } n \text{ in } t_1\rangle \mid \langle\text{let } y = \text{accept } n \text{ in } t_2\rangle \rightarrow (\nu c)(\langle\text{let } x = c \text{ in } t_1\rangle \mid \langle\text{let } y = c \text{ in } t_2\rangle).$$

where $c$ is not free in $t_1, t_2$, and

$$\Delta \vdash \Sigma \triangleright \langle\text{let } x = \text{request } n \text{ in } t_1\rangle \mid \langle\text{let } y = \text{accept } n \text{ in } t_2\rangle.$$
The only proof tree for this sequent is of the form

\[ \Delta \vdash \Sigma_1 \cdot \Sigma_2 \vdash \langle \text{let } x = \text{request } n \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{accept } n \text{ in } t_2 \rangle \]

where (1) is the tree

\[ \Delta \vdash n: [S] \quad d_1 \text{ fresh} \]

\[ \frac{\Delta \vdash \Sigma_1 \vdash \text{request } n: \text{Chan } d_1 \triangleleft \Sigma_1, d_1: S \quad \Delta, x: \text{Chan } d_1 \vdash \Sigma_1, d_1: S \triangleright t_1: T \triangleleft \emptyset}{\Delta \vdash \Sigma_1 \vdash \langle \text{let } x = \text{request } n \text{ in } t_1 \rangle} \]

T-THREAD

and (2) is the tree

\[ \Delta \vdash n: [S] \quad d_2 \text{ fresh} \]

\[ \frac{\Delta \vdash \Sigma_2 \vdash \text{accept } n: \text{Chan } d_2 \triangleleft \Sigma_2, d_2: S \quad \Delta, y: \text{Chan } d_2 \vdash \Sigma_2, d_2: S \triangleright t_2: U \triangleleft \emptyset}{\Delta \vdash \Sigma_2 \vdash \langle \text{let } y = \text{accept } n \text{ in } t_2 \rangle} \]

T-THREAD

From the assumptions in the above tree we may build the following derivation tree, since as \( c \) does not occur in \( t_1, t_2 \), by the variable convention it does not occur neither in \( \Delta, \Sigma_1, \Sigma_2 \). Thus, we are in the conditions of Lemma 14, since \( d_1 \) and \( d_2 \) are fresh in the assumptions of tree (1) and (2). By the same reason \( \Delta\{c/d_i\} = \Delta \), and similarly for \( \Sigma_1, \Sigma_2 \) and for \( T, U \).

\[ \frac{\Delta \vdash \Sigma_1 \cdot \Sigma_2, c: \bot \vdash \langle \text{let } x = c \text{ in } t_1 \rangle \mid \langle \text{let } y = c \text{ in } t_2 \rangle}{\text{T-PAR}} \]

\[ \frac{\Delta \vdash \Sigma_1 \cdot \Sigma_2 \vdash (\nu c)((\text{let } x = c \text{ in } t_1) \mid \langle \text{let } y = c \text{ in } t_2 \rangle)}{\text{T-NEWB}} \]

where (1*) is the tree

\[ \frac{\Delta \vdash \Sigma_1 \cdot \Sigma_2, c: \text{Chan } c}{\text{T-CHAN}} \]

\[ \frac{\Delta, x: \text{Chan } d_1 \vdash \Sigma_1, d_1: S \triangleright t_1: T \triangleleft \emptyset}{\text{Lemma 14}} \]

\[ \frac{\Delta, x: \text{Chan } c \vdash \Sigma_1, c: S \triangleright t_1: T \triangleleft \emptyset}{\text{T-LET}} \]

\[ \frac{\Delta \vdash \Sigma_1, c: S \triangleright \text{let } x = c \text{ in } t_1: T \triangleleft \emptyset}{\text{T-THREAD}} \]

and (2*) is the tree

\[ \frac{\Delta \vdash \Sigma_1 \cdot \Sigma_2, c: \text{Chan } c}{\text{T-CHAN}} \]

\[ \frac{\Delta, y: \text{Chan } d_2 \vdash \Sigma_2, d_2: S \triangleright t_2: U \triangleleft \emptyset}{\text{Lemma 14}} \]

\[ \frac{\Delta, y: \text{Chan } c \vdash \Sigma_2, c: S \triangleright t_2: U \triangleleft \emptyset}{\text{T-LET}} \]

\[ \frac{\Delta \vdash \Sigma_2, c: S \triangleright \text{let } y = c \text{ in } t_2: U \triangleleft \emptyset}{\text{T-THREAD}} \]
R-Com. By hypothesis, we have

\[
\langle \text{let } x = \text{receive } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{send } v \text{ on } c \text{ in } t_2 \rangle \rightarrow
\langle \text{let } x = v \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit in } t_2 \rangle
\]

There are two possible derivations to consider, depending on the kind of value \(v\) carried by channel \(c\). Let us consider the case where \(v\) is a channel (the other case is similar—and simpler). The typing derivation is as follows.

\[
\Delta \vdash \Sigma \triangleright \langle \text{let } x = \text{receive } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{send } v \text{ on } c \text{ in } t_2 \rangle \rightarrow
\langle \text{let } x = v \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit in } t_2 \rangle
\]

where \(\Sigma\) is \((\Sigma_1 \cdot \Sigma_2), c: \bot, d: S', (1)\) is the tree

\[
\Delta \vdash c: \text{Chan} \quad d: \text{fresh}
\]

\[
\Delta \vdash \Sigma_1, c: ?S'.S \triangleright \text{receive } c: \text{Chan} \quad d: \Sigma'_1 \quad \Delta, x: \text{Chan} \quad d: \Sigma'_1 \triangleright t_1: T \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_1, c: ?S'.S \triangleright \text{let } x = \text{receive } c \text{ in } t_1: T \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_1, c: ?S'.S \triangleright \langle \text{let } x = \text{receive } c \text{ in } t_1 \rangle
\]

where \(\Sigma'_1\) is \(\Sigma_1, c: S, d: S', (1)\) is the tree

\[
\Delta \vdash v: \text{Chan} \quad d \quad \Delta \vdash c: \text{Chan} \quad d
\]

\[
\Delta \vdash \Sigma_2, c: !S'.S, d: S' \triangleright \text{send } v \text{ on } c: \text{Unit} \quad d \quad \Delta, y: \text{Unit} \quad d \quad \Delta \vdash \Sigma'_2 \triangleright t_2: U \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_2, c: !S'.S, d: S' \triangleright \text{let } y = \text{send } v \text{ on } c \text{ in } t_2: U \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_2, c: !S'.S, d: S' \triangleright \langle \text{let } y = \text{send } v \text{ on } c \text{ in } t_2 \rangle
\]

where \(\Sigma'_2\) is \(\Sigma_2, c: S\).

From the assumptions in the above tree, one may build

\[
\Delta \vdash (\Sigma_1 \cdot \Sigma_2), c: \bot, d: S' \triangleright \langle \text{let } x = v \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{unit in } t_2 \rangle
\]

where \((1^*)\) is the tree

\[
\Delta \vdash v: \text{Chan} \quad d \quad \Delta, x: \text{Chan} \quad d \quad \Delta \vdash \Sigma_1, c: S, d: S' \triangleright t_1: T \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_1, c: S, d: S' \triangleright \text{let } x = v \text{ in } t_1: T \triangleleft \emptyset
\]

\[
\Delta \vdash \Sigma_1, c: S, d: S' \triangleright \langle \text{let } x = v \text{ in } t_1 \rangle
\]
and (2*) is the tree

\[
\begin{array}{c}
\Delta \vdash \text{unit: Unit} \quad \text{T-UNIT} \\
\Delta, y: \text{Unit} \vdash \Sigma_2, c: S \triangleright t_2: U \triangleleft \emptyset \quad \text{T-LET} \\
\Delta \vdash \Sigma_2, c: S \triangleright \langle \text{let } y = \text{unit in } t_2 \rangle \quad \text{T-THREAD}
\end{array}
\]

Notice that the type environment \((\Sigma_1, c: S, d: S') \bullet (\Sigma_2, c: S)\) in the conclusion of rule T-PAR above is defined, since \((\Sigma_1, c: ?S'.S) \bullet (\Sigma_2, c: !S'.S, d: S')\) is defined (in the tree for the hypothesis) and \(d\) is fresh (in tree (1)).

**R-Close.** By hypothesis, we have

\[
\langle \text{let } x = \text{close } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{close } c \text{ in } t_2 \rangle \rightarrow \\
\langle \text{let } x = \text{unit in } t_1 \rangle \mid \langle \text{let } y = \text{unit in } t_2 \rangle
\]

and

\[
\begin{array}{c}
\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c: \bot \triangleright \langle \text{let } x = \text{close } c \text{ in } t_1 \rangle \mid \langle \text{let } y = \text{close } c \text{ in } t_2 \rangle \\
\text{T-PAR}
\end{array}
\]

where (1) is the tree

\[
\begin{array}{c}
\cdot \cdot \cdot \\
\Delta, x: \text{Unit} \vdash \Sigma_1 \triangleright t_1: T \triangleleft \emptyset \quad \text{T-LET} \\
\Delta \vdash \Sigma_1, c: \text{End} \triangleright \langle \text{let } x = \text{close } c \text{ in } t_1 \rangle \quad \text{T-THREAD} \\
\Delta \vdash \Sigma_1, c: \text{End} \triangleright \langle \text{let } x = \text{close } c \text{ in } t_1 \rangle
\end{array}
\]

and (2) is the tree below.

\[
\begin{array}{c}
\cdot \cdot \cdot \\
\Delta, y: \text{Unit} \vdash \Sigma_2 \triangleright t_2: U \triangleleft \emptyset \quad \text{T-LET} \\
\Delta \vdash \Sigma_2, c: \text{End} \triangleright \langle \text{let } y = \text{close } c \text{ in } t_2 \rangle \quad \text{T-THREAD} \\
\Delta \vdash \Sigma_2, c: \text{End} \triangleright \langle \text{let } y = \text{close } c \text{ in } t_2 \rangle
\end{array}
\]

From the assumptions in the above tree, one may then build

\[
\begin{array}{c}
\Delta \vdash (\Sigma_1 \bullet \Sigma_2) \triangleright \langle \text{let } x = \text{unit in } t_1 \rangle \mid \langle \text{let } y = \text{unit in } t_2 \rangle \\
\text{T-PAR}
\end{array}
\]

where (1*) is the tree

\[
\begin{array}{c}
\Delta \vdash \text{unit: Unit} \quad \text{T-VAL} \\
\Delta \vdash \Sigma_1 \triangleright \text{unit: Unit} \quad \text{T-LET} \\
\Delta \vdash \Sigma_1 \triangleright \langle \text{let } x = \text{unit in } t_1 \rangle \quad \text{T-THREAD}
\end{array}
\]

35
and (2*) is a similar tree.

**R-New.** By hypothesis, we have

\[
\langle \text{let } x = \text{newS in } t \rangle \rightarrow (\nu n: [S]) \langle \text{let } x = n \text{ in } t \rangle
\]

and

\[
\begin{array}{c}
\Delta \vdash \Sigma \triangleright \text{newS}: [S] \triangleleft \Sigma \quad \text{T-NEW} \\
\Delta, x: [S] \vdash \Sigma \triangleright t: T < \emptyset \quad \text{T-LET} \\
\Delta \vdash \Sigma \triangleright \langle \text{let } x = \text{newS in } t \rangle
\end{array}
\]

From the hypothesis in the above tree, we build a tree to complete the proof. Notice that, by the hypothesis of rule R-New, \( n \) is not free in \( t \). Thus, Lemma 8 is applicable to the premise of rule T-LET above, and hence,

\[
\begin{array}{c}
\Delta, n: [S] \vdash n: [S] \\
\Delta, n: [S], x: [S] \vdash \Sigma \triangleright t: T < \emptyset \quad \text{T-LET} \\
\Delta, n: [S] \vdash \Sigma \triangleright \langle \text{let } x = n \text{ in } t \rangle \\
\Delta \vdash \Sigma \triangleright (\nu n) \langle \text{let } x = n \text{ in } t \rangle
\end{array}
\]

**R-Fork.** Follows the pattern in all the above cases.

**R-App.** Follows the pattern in all the above cases, using Lemma 15.

**R-Rec.** By hypothesis, we have

\[
\langle \text{let } x = (\text{rec } (y: T).v)u \text{ in } t \rangle \rightarrow \langle \text{let } x = (v\{\text{rec } (y: T).v/y\})u \text{ in } t \rangle
\]

and making \( T = (\Sigma; T' \rightarrow U; \Sigma') \), we also have

\[
\begin{array}{c}
\Delta, y: T \vdash v: T \\
\Delta \vdash \text{rec } (y: T).v: T \quad \text{T-REC} \\
\Delta \vdash u: T \quad \text{T-APP} \\
\Delta \vdash \Sigma \triangleright \langle \text{let } x = (\text{rec } (y: T).v)u \text{ in } t \rangle: T < \emptyset \quad \text{T-LET} \\
\Delta \vdash \Sigma \triangleright \langle \text{let } x = (\text{rec } (y: T).v)u \text{ in } t \rangle
\end{array}
\]

where (1) is \( \Delta, x: U \vdash \Sigma \triangleright t: T < \emptyset \). Then, one may build the following derivation
to complete the proof.

\[
\begin{align*}
\Delta, y: T & \vdash v: T & \text{T-REC} \\
\Delta & \vdash \text{rec} (y: T). v: T & \text{Lemma 15}
\end{align*}
\]

\[
\begin{align*}
\Delta & \vdash v \{ \text{rec} (y: T). v/y \}: T & \Delta & \vdash u: T & \text{T-APP} \\
\Delta & \vdash \Sigma \triangleright (v \{ \text{rec} (y: T). v/y \}) u: U \triangleleft \Sigma' & \hfill (1) & \text{T-LET} \\
\Delta & \vdash \Sigma \triangleright \text{let } x = (v \{ \text{rec} (y: T). v/y \}) u \text{ in } t: T \triangleleft \emptyset & \Delta & \vdash \Sigma \triangleright \langle \text{let } x = (v \{ \text{rec} (y: T). v/y \}) u \text{ in } t \rangle & \text{T-THREAD}
\end{align*}
\]

**R-Beta.** By hypothesis, we have

\[
\langle \text{let } x = v \text{ in } t \rangle \rightarrow \langle t \{ v/x \} \rangle.
\]

There are two possible derivations for \(\langle \text{let } x = v \text{ in } t \rangle\); we analyse each in turn. When the derivation uses rule T-LET, the result follows by Lemma 15. When the derivation uses rule T-POLYLET, the result is immediate.

**R-Let.** By hypothesis, we have

\[
\langle \text{let } x = (\text{let } y = e \text{ in } t') \text{ in } t \rangle \rightarrow \langle \text{let } y = e \text{ in } (\text{let } x = t' \text{ in } t) \rangle
\]

and

\[
\begin{align*}
\Delta & \vdash \Sigma \triangleright e: T \triangleleft \Sigma_1 & \Delta, y: T & \vdash \Sigma_1 \triangleright t': T_1 \triangleleft \Sigma'_1 \\
\Delta & \vdash \Sigma \triangleright \text{let } y = e \text{ in } t': T_1 \triangleleft \Sigma'_1 & \Delta, x: T_1 & \vdash \Sigma'_1 \triangleright t: U \triangleleft \emptyset
\end{align*}
\]

\[
\begin{align*}
\Delta & \vdash \Sigma \triangleright \text{let } x = (\text{let } y = e \text{ in } t') \text{ in } t: U \triangleleft \emptyset & \Delta & \vdash \Sigma \triangleright \langle \text{let } x = (\text{let } y = e \text{ in } t') \text{ in } t \rangle & \text{T-THREAD}
\end{align*}
\]

Then, using Lemma 6 we build the following derivation to complete the proof. Notice that, by the variable convention, \(y\) is not free in \(t\), since it is bound in \(\text{let } y = e \text{ in } t'\).

\[
\begin{align*}
\Delta & \vdash \Sigma \triangleright e: T \triangleleft \Sigma_1 & \hfill (1) & \text{T-LET} \\
\Delta & \vdash \Sigma \triangleright \text{let } y = e \text{ in } (\text{let } x = t' \text{ in } t): U \triangleleft \emptyset & \Delta & \vdash \Sigma \triangleright \langle \text{let } y = e \text{ in } (\text{let } x = t' \text{ in } t) \rangle & \text{T-THREAD}
\end{align*}
\]

where (1) is the tree

\[
\begin{align*}
\Delta, y: T & \vdash \Sigma_1 \triangleright t': T_1 \triangleleft \Sigma'_1 & \Delta, y: T, x: T_1 & \vdash \Sigma'_1 \triangleright t: U \triangleleft \emptyset & \text{Lemma 6} \\
\Delta, y: T & \vdash \Sigma_1 \triangleright \text{let } x = t' \text{ in } t: U \triangleleft \emptyset & \text{T-LET}
\end{align*}
\]

**R-IfT/R-IfF.** Follows the pattern in all the above cases.
R-Branch. Follows the pattern in all the above cases.

R-Conf. The three cases follow directly by induction. For the rule that uses structural congruence, we use Lemma 13.

B Proof of Theorem 3, Type Safety

We start with a couple of easy results.

Lemma 16 Suppose that $\Delta \vdash \Sigma \triangleright C$.

(1) If $C$ is a $c$-thread, then $c$ is in the domain of $\Sigma$.

(2) If $C$ is a $c$-redex, then $\Sigma$ is of the form $\Sigma', c: \bot$.

PROOF. 1. A simple analysis of the conclusions of the last rule applied in the derivation of the sequent for $c$-threads, namely $T$-SEND, $T$-SEMD, $T$-RECEIVE, $T$-RECEIVES, $T$-CASE, $T$-SELECT, and $T$-CLOSE.

2. A simple analysis of the possible derivation trees for the three possible $c$-redex cases.

PROOF. [Theorem 3, page 18] By contradiction, assuming faulty configurations typable and performing a case analysis on the possible forms of the faulty configurations.

Without loss of generality, assume that $\Delta \vdash \Sigma \triangleright (\nu \vec{n} : [\vec{S}]) (\nu \vec{c} (\nu \vec{d}))(C_1 \mid C_2)$, where $\vec{d}$ are the channels that do not occur in $\Sigma$. Build the only possible proof tree for the above sequent, first using rule $T$-NEWN as many times as there are names in $\vec{n}$, then proceeding similarly with rules $T$-NEWB for $\vec{c}$ and $T$-NEWC for $\vec{d}$, a final with rule $T$-PAR, to obtain two subtrees ending with the sequents $(i = 1, 2)$:

$$\Delta, \vec{n} : [\vec{S}] \vdash \Sigma_i \triangleright C_i$$

(B.1)

where $\Sigma, \vec{c} : \vec{\bot} = \Sigma_1 \bullet \Sigma_2$. We now analyse each of the five possible classes of faulty configurations defined in Section 6, where we let $\Delta' = \Delta, \vec{n} : [\vec{S}]$.

1. The three cases are similar. We analyse the conditional expression. The only derivation tree for sequent (B.1) above is of the form below.

$$\Delta' \vdash v : \text{Bool} \quad \cdots \quad \cdots \quad T\text{-If} \quad \Delta' \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \bot \quad \cdots \quad T\text{-Let}$$

$$\Delta' \vdash \Sigma_1 \triangleright \langle \text{let } x = \text{if } v \text{ then } e_1 \text{ else } e_2 \text{ in } t \rangle : \bot \quad T\text{-THREAD}$$

$$\Delta' \vdash \Sigma_1 \triangleright \langle \text{let } x = \text{if } v \text{ then } e_1 \text{ else } e_2 \text{ in } t \rangle$$
Analysing the rules for values (Figure 6, page 14), one realises that $v$ can only be true or false, for the T-VAR does not apply since variables are not in the domain of $\Delta'$, and the type in the conclusion of the remaining rules (T-Abs, T-Rec, T-Chan, T-Name, Unit) is not Bool.

2. As above, analyse the lower part of the only proof tree for, say,

$$\Delta' \vdash \Sigma_1 \triangleright \lbrack \text{let } x = \text{accept } v \text{ in } t \rbrack$$

to obtain a tree for

$$\Delta' \vdash v : \lbrack S \rbrack.$$

Once again, among the rules for values, only T-Name applies. Hence, $v$ is a name.

3. As above, analyse the lower part of the only proof tree for, say,

$$\Delta' \vdash \Sigma_1 \triangleright \lbrack \text{let } x = \text{receive } v \text{ in } t \rbrack$$

to obtain a tree for

$$\Delta' \vdash v : \text{Chan } c.$$

Once again, among the rules for values, only T-Chan applies. Clearly $v$ can only be the channel $c$.

4. There are several cases to check in this point; they are all similar. Pick, for example, the pair select/close, and expand the lower part of the proof tree, until obtaining subtrees for the following two sequents,

$$\Delta' \vdash \Theta_1 \triangleright \text{select } l \text{ on } c : T_1 \triangleleft \Theta_1' \quad \Delta' \vdash \Theta_2 \triangleright \text{close } c : T_2' \triangleleft \Theta_2'$$

where $\Sigma_1 = \Theta_1 \bullet \Theta_2$. Analysing the rule for select, one finds that $c : \oplus \lbrack l : S \rbrack$ must be in $\Theta_1$. Similarly, analysing the rule for close one realises that $c : \text{End}$ must be in $\Theta_2$. Then, $\Theta_1 \bullet \Theta_2$ is not defined (for $\oplus \lbrack l : S \rbrack$ is not the dual of End), hence $(\nu \vec{n} : \lbrack \vec{S} \rbrack)(\nu \vec{c})(\nu \vec{d})(C_1 \mid C_2)$ is not typable.

5. We check the case for three $c$-threads $(\langle t_1 \rangle \mid \langle t_2 \rangle) \mid \langle t_3 \rangle$, the others reduce to this. We have:

$$\Delta' \vdash \Theta_1 \triangleright \langle t_1 \rangle \quad \Delta' \vdash \Theta_2 \triangleright \langle t_2 \rangle \quad \Delta' \vdash \Theta_3 \triangleright \langle t_3 \rangle \quad \text{T-PAR}$$

$$\Delta' \vdash \Sigma_1 \triangleright \langle \langle t_1 \rangle \mid \langle t_2 \rangle \rangle \mid \langle t_3 \rangle$$

with $\Sigma_1 = \Theta_1 \bullet \Theta_2$. If $\langle t_1 \rangle \mid \langle t_2 \rangle$ is not a $c$-redex, then we use the previous case. Otherwise, by Lemma 16, it must be the case that $c : \perp$ is part of $\Theta_1$. Since $\langle t_3 \rangle$ is a $c$-thread, by Lemma 16, $c$ is in the domain of $\Theta_2$. But then $\Theta_1 \bullet \Theta_2$ is not defined (for $\perp$ is dual to no type), and $(\nu \vec{n} : \lbrack \vec{S} \rbrack)(\nu \vec{c})(\nu \vec{d})(C_1 \mid C_2)$ is not typable.
C Proof of Theorem 4, Soundness of Typechecking

The first two lemmas deal with the first conclusion of the soundness result: that $\Sigma - \Sigma'$ is defined.

**Lemma 17** If $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ then $\Sigma - \Sigma_1$ is defined.

**PROOF.** By induction on the derivation of $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$. In most cases the conclusion follows immediately from the inference rule and the definition of $\Sigma - \Sigma'$. Two cases are worthy of comment.

**C-Fork.** We have a derivation ending with

$$\begin{align*}
\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \emptyset & \quad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \emptyset \\
& \quad \frac{\text{fork } t_1; t_2 \mapsto \Sigma_2; T_2; \emptyset}{\Gamma; \Sigma; \text{fork } t_1; t_2 \mapsto \Sigma_2; T_2; \emptyset}
\end{align*}$$

By induction, $\Sigma - \Sigma_1$ and $\Sigma_1 - \Sigma_2$ are defined. Therefore $\Sigma - \Sigma_2$ is defined, because $\Sigma_2 \subseteq \Sigma_1 \subseteq \Sigma$.

**C-Let.** We have a derivation ending with

$$\begin{align*}
\Gamma; \Sigma; e \mapsto \Sigma_1; T_1; \emptyset & \quad \Gamma; \Sigma_1; x : T_1, \Sigma_1; t \mapsto \Sigma_2; T_2; \emptyset \\
& \quad \frac{\text{let } x = e \text{ in } t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma_1 \cap \Sigma_2), \Sigma_2'}{\Gamma; \Sigma; \text{let } x = e \text{ in } t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma_1 \cap \Sigma_2), \Sigma_2'}
\end{align*}$$

By induction, $\Sigma - \Sigma_1$ is defined, i.e. $\Sigma_1 \subseteq \Sigma$. Hence $\Sigma_1 \cap \Sigma_2 \subseteq \Sigma$, i.e. $\Sigma - (\Sigma_1 \cap \Sigma_2)$ is defined.

**Lemma 18** If $\Delta; \Sigma; C \mapsto \Sigma'$ then $\Sigma - \Sigma'$ is defined.

**PROOF.** By induction on the derivation of $\Delta; \Sigma; C \mapsto \Sigma'$. The case of rule C-THREAD uses Lemma 17. The case of rule C-PAR uses the same argument as the case of rule C-FORK in the proof of Lemma 17. The case of C-NEWN uses the induction hypothesis directly.

**Lemma 19** If $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ and $\Sigma$ contains no $\bot$ types then $\text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) = \emptyset$.

**PROOF.** Straightforward induction on the derivation.

Lemmas 20 and 21 are proved by simultaneous induction on the size of the derivation of the typechecking judgement in question. For clarity we present the proofs separately, noting that the use of each lemma in the proof of the other is valid.
Lemma 20 (Soundness for Values) If $\Gamma; v \mapsto T$ then $\Gamma \vdash v : T$.

**PROOF.** The cases for literal values, channels, names and variables are trivial.

**C-Abs.** We have a derivation finishing with

$$ \Gamma, x : T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2 \quad \Gamma; \lambda(\Sigma; x : T).e \mapsto (\Sigma; T \rightarrow U; \Sigma_1, \Sigma_2) $$

Applying soundness for expressions (Lemma 21) to the hypothesis, we have

$$ \Gamma, x : T \vdash \Sigma - \Sigma_1 \triangleright e : U \triangleleft \Sigma_2 $$

Using Weakening, we have

$$ \Gamma, x : T \vdash (\Sigma - \Sigma_1), \Sigma_1 \triangleright e : U \triangleleft \Sigma_1, \Sigma_2 $$

and $(\Sigma - \Sigma_1), \Sigma_1 = \Sigma$. Now we use rule T-Abs to conclude

$$ \Gamma \vdash \lambda(\Sigma; x : T).e : (\Sigma; T \rightarrow U; \Sigma_1, \Sigma_2) $$

as required.

**C-Rec.** We have a derivation finishing with

$$ \Gamma, x : T; v \mapsto T \quad T = (\Sigma; U \rightarrow U'; \Sigma') \quad \Gamma; \text{rec} (x : T).v \mapsto T $$

By induction we have

$$ \Gamma, x : T \vdash v : T $$

from which we use rule T-Rec to conclude

$$ \Gamma \vdash \text{rec} (x : T).v : T $$

as required.

Lemma 21 (Soundness for Expressions) If $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ then $\Sigma - \Sigma_1$ is defined and $\Gamma \vdash \Sigma - \Sigma_1 \triangleright e : T \triangleleft \Sigma_2$.

**PROOF.** By induction on the derivation of $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$. Lemma 17 guarantees that $\Sigma - \Sigma_1$ is defined. We consider cases for each of the inference rules in the typechecking algorithm.

**C-SendD.** We have

$$ \Gamma; v \mapsto D \quad \Gamma; v' \mapsto \text{Chan} \ c \quad \Gamma; \Sigma, c : !D.S; \text{send} \ v \text{ on } v' \mapsto \Sigma; \text{Unit}; c : S $$
Lemma 20 implies that $\Gamma \vdash v : D$ and $\Gamma \vdash v' : \text{Chan } c$. Using rule T-SENDD we build the following derivation, noting that $\Sigma, c : !D. S \prec c : !D. S$.

$$
\begin{array}{c}
\Gamma \vdash v : D \\
\Gamma \vdash v' : \text{Chan } c \\
\hline
\Gamma \vdash c : !D. S \triangleright \text{send } v \text{ on } v' : \text{Unit } \triangleleft c : S
\end{array}
$$

The remaining cases in Figure 11 are similar.

**C-Request.** We have

$$
\begin{array}{c}
\Gamma; v \mapsto [S] \\
\hline
\Gamma; \Sigma; \text{request } v \mapsto \Sigma; \text{Chan } c; c : S
\end{array}
$$

Lemma 20 implies that $\Gamma \vdash v : [S]$. Using rule T-REQUEST and noting that $\Sigma - \Sigma = \emptyset$, we obtain

$$
\begin{array}{c}
\Gamma \vdash v : [S] \\
\hline
\Gamma \vdash \emptyset \triangleright \text{request } v : \text{Chan } c \triangleleft c : S
\end{array}
$$

The case of C-ACCEPT is similar.

**C-Fork.** We have

$$
\begin{array}{c}
\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \emptyset \\
\Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \emptyset \\
\hline
\Gamma; \Sigma; \text{fork } t_1; t_2 \mapsto \Sigma_2; T_2; \emptyset
\end{array}
$$

The induction hypothesis yields the hypotheses in the following instance of rule T-FORK:

$$
\begin{array}{c}
\Gamma \vdash \Sigma - \Sigma_1 \triangleright t_1 : T_1 \triangleleft \emptyset \\
\Gamma \vdash \Sigma_1 - \Sigma_2 \triangleright t_2 : T_2 \triangleleft \emptyset \\
\hline
\Gamma \vdash (\Sigma - \Sigma_1) \bullet (\Sigma_1 - \Sigma_2) \triangleright \text{fork } t_1; t_2 : T_2 \triangleleft \emptyset
\end{array}
$$

and because $\Sigma$ contains no $\bot$ types, $(\Sigma - \Sigma_1) \bullet (\Sigma_1 - \Sigma_2) = \Sigma - \Sigma_2$.

**C-New.** We have

$$
\begin{array}{c}
\Gamma; \Sigma; \text{new } S \mapsto \Sigma; [S]; \emptyset
\end{array}
$$

and, noting that $\Sigma - \Sigma = \emptyset$, we obtain

$$
\begin{array}{c}
\Gamma \vdash \emptyset \triangleright \text{new } S : [S] \triangleleft \emptyset
\end{array}
$$

directly from rule T-New.

**C-App.** We have

$$
\begin{array}{c}
\Gamma; v \mapsto (\Sigma; T \rightarrow U; \Sigma') \\
\Gamma; v' \mapsto T \\
\hline
\Gamma; \Sigma; vv' \mapsto \emptyset; U; \Sigma'
\end{array}
$$
We have $\Sigma - \emptyset = \Sigma$, and Lemma 20 gives the hypotheses of the following instance of rule $T$-App:

\[
\frac{\Gamma \vdash v: (\Sigma; T \to U; \Sigma')}{\Gamma \vdash v' : T}
\]

C-Val. We have

\[
\frac{\Gamma; v \mapsto T}{\Gamma; \Sigma; v \mapsto \Sigma; T; \emptyset}
\]

Lemma 20 gives $\Gamma \vdash v: T$, and $\Sigma - \Sigma = \emptyset$, so we use rule $T$-Val to obtain

\[
\frac{\Gamma \vdash v: T}{\Gamma \vdash \emptyset \vdash v: T \lessdot \emptyset}
\]

C-Let. We have

\[
\frac{\Gamma; \Sigma; e \mapsto \Sigma_1; T_1; \Sigma'_1 \quad \Gamma, x: T_1; \Sigma_1, \Sigma'_1; t \mapsto \Sigma_2; T_2; \Sigma'_2}{\Gamma; \Sigma; \text{let } x = e \text{ in } t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma'_1 \cap \Sigma_2), \Sigma'_2}
\]

The induction hypothesis gives

\[
\Gamma \vdash \Sigma - \Sigma_1 \vdash e: T_1 \lessdot \Sigma'_1
\]

and

\[
\Gamma, x: T_1 \vdash (\Sigma_1, \Sigma'_1) - \Sigma_2 \vdash t: T_2 \lessdot \Sigma'_2
\]

Using Weakening (Lemma 6) with $\Sigma_1 - (\Sigma_1 \cap \Sigma_2)$ we get

\[
\Gamma \vdash (\Sigma - \Sigma_1) \cup (\Sigma_1 - (\Sigma_1 \cap \Sigma_2)) \vdash e: T_1 \lessdot \Sigma'_1 \cup (\Sigma_1 - (\Sigma_1 \cap \Sigma_2)).
\]

Using Weakening with $\Sigma'_1 \cap \Sigma_2$ we get

\[
\Gamma, x: T_1 \vdash ((\Sigma_1 \cup \Sigma'_1) - \Sigma_2) \cup (\Sigma'_1 \cap \Sigma_2) \vdash t: T_2 \lessdot (\Sigma'_1 \cap \Sigma_2) \cup \Sigma'_2.
\]

By straightforward set-theoretic reasoning, making use of the fact that $\Sigma_2 \subseteq \Sigma_1 \cup \Sigma'_1$ (Lemma 17 applied to the second typing judgement) we can show

\[
(\Sigma - \Sigma_1) \cup (\Sigma_1 - (\Sigma_1 \cap \Sigma_2)) = \Sigma - (\Sigma_1 \cap \Sigma_2)
\]

and

\[
\Sigma'_1 \cup (\Sigma_1 - (\Sigma_1 \cap \Sigma_2)) = ((\Sigma_1 \cup \Sigma'_1) - \Sigma_2) \cup (\Sigma'_1 \cap \Sigma_2).
\]

Therefore, using rule $T$-Let we get

\[
\Gamma \vdash \Sigma - (\Sigma_1 \cap \Sigma_2) \vdash \text{let } x = e \text{ in } t: T_2 \lessdot (\Sigma'_1 \cap \Sigma_2), \Sigma'_2.
\]

We are finally in a position to prove soundness.
**PROOF.** [Theorem 4, page 22] By induction on the derivation of $\Delta; \Sigma; C \leftrightarrow \Sigma'$. Lemma 18 guarantees that $\Sigma - \Sigma'$ is defined. We consider cases for each of the inference rules in the typechecking algorithm.

**C-Thread.** We have

\[
\Delta; \Sigma; t \mapsto \Sigma'; T; \emptyset \\
\Delta; \Sigma; \langle t \rangle \mapsto \Sigma'
\]

Lemma 21 gives $\Delta \vdash \Sigma - \Sigma' \triangleright t: T \triangleleft \emptyset$ and we use rule T-THREAD to obtain

\[
\Delta \vdash \Sigma - \Sigma' \triangleright t: T \triangleleft \emptyset \\
\Delta \vdash \Sigma - \Sigma' \triangleright \langle t \rangle
\]

**C-Par.** We have

\[
\Delta; \Sigma; C_1 \mapsto \Sigma_1 \quad \Delta; \Sigma_1; C_2 \mapsto \Sigma_2 \\
\Delta; \Sigma; C_1 | C_2 \mapsto \Sigma_2
\]

The induction hypothesis gives the hypotheses in the following instance of rule T-PAR:

\[
\Delta \vdash \Sigma - \Sigma_1 \triangleright C_1 \quad \Delta \vdash \Sigma_1 - \Sigma_2 \triangleright C_2 \\
\Delta \vdash (\Sigma - \Sigma_1) \bullet (\Sigma_1 - \Sigma_2) \triangleright C_1 | C_2
\]

and by the same reasoning as for case C-FORK in the proof of Lemma 21 we have $(\Sigma - \Sigma_1) \bullet (\Sigma_1 - \Sigma_2) = \Sigma - \Sigma_2$.

**C-NewN** follows directly by using the induction hypothesis.

**D** Proof of Theorem 5, Completeness of Typechecking

**Lemma 22** If $\Sigma \bullet \Sigma'$ is defined and does not contain $\bot$ then $\Sigma \bullet \Sigma' = \Sigma, \Sigma'$.

**PROOF.** Directly from the definition.

Lemmas 23 and 24 are proved by simultaneous induction on the size of the derivation of the typing judgement in question. For clarity we present the proofs separately, noting that the use of each lemma in the proof of the other is valid.

**Lemma 23 (Completeness for Values)** If $\Gamma \vdash v: T$ then $\Gamma; v \mapsto T$.

**PROOF.** The cases for literal values, channels, names and variables are trivial.
In the case of rule T-Abs we have a derivation finishing with

$$\Gamma, x: T \vdash \Sigma \triangleright e: U \triangleleft \Sigma'$$

$$\Gamma \vdash \lambda(\Sigma; x: T).e: (\Sigma; T \to U; \Sigma')$$

Applying completeness for expressions (Lemma 24) to the hypothesis, we have

$$\Gamma, x: T; \Sigma; e \mapsto \Sigma_2; U; \Sigma_3$$

where $\Sigma' = \Sigma_2, \Sigma_3$. Using C-Abs we obtain

$$\Gamma; \lambda(\Sigma; x: T).e \mapsto (\Sigma; T \to U; \Sigma_2, \Sigma_3)$$

as required.

The case of rule T-Rec follows directly from the induction hypothesis.

**Lemma 24 (Completeness for Expressions)** If $\Gamma \vdash \Sigma \triangleright e: T \triangleleft \Sigma_1$ and $\Sigma \bullet \Sigma'$ is defined and does not contain $\bot$ then $\Gamma; \Sigma \bullet \Sigma'; e \mapsto \Sigma_2 \bullet \Sigma'; T; \Sigma_3$ where $\Sigma_1 = \Sigma_2, \Sigma_3$ and $\Sigma_2 \subseteq \Sigma$.

**PROOF.** First observe that $\Sigma \bullet \Sigma' = \Sigma, \Sigma'$ and $\Sigma_2 \bullet \Sigma' = \Sigma_2, \Sigma'$.

The proof is by induction on the derivation of $\Gamma \vdash \Sigma \triangleright e: T \triangleleft \Sigma_1$ with a case for each inference rule.

For T-SEND D we have

$$\Gamma \vdash v: D \quad \Gamma \vdash v': \text{Chan } c$$

$$\Gamma \vdash \Sigma, c: !D.S \triangleright send v on v': \text{Unit} \triangleleft \Sigma, c: S$$

$$(\Sigma, c: !D.S) \bullet \Sigma' = (\Sigma \bullet \Sigma'), c: !D.S$$ and by using C-SEND D we obtain

$$\Gamma; (\Sigma \bullet \Sigma'), c: !D.S; send v on v' \mapsto \Sigma \bullet \Sigma'; \text{Unit}; c: S.$$  

For T-FORK we have

$$\Gamma \vdash \Sigma_1 \triangleright t_1: T_1 \triangleleft \emptyset \quad \Gamma \vdash \Sigma_2 \triangleright t_2: T_2 \triangleleft \emptyset$$

$$\Gamma \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \text{fork } t_1; t_2: T_2 \triangleleft \emptyset$$

By the induction hypothesis,

$$\Gamma; \Sigma_1 \bullet \Sigma_2 \bullet \Sigma'; t_1 \mapsto \Sigma_2 \bullet \Sigma'; T_1; \emptyset$$

and

$$\Gamma; \Sigma_2 \bullet \Sigma'; t_2 \mapsto \Sigma'; T_2; \emptyset$$
so by rule C-FORK we obtain

$\Gamma; \Sigma_1 \bullet \Sigma_2 \bullet \Sigma'; \text{fork } t_1; t_2 \mapsto \Sigma_2; T_2; \emptyset$

The case of T-Let is the most complex. We have

$\Gamma \vdash \Sigma \triangleright e: T_1 \triangleleft \Sigma'' \quad \Gamma, x: T_1 \vdash \Sigma'' \triangleright t: T_2 \triangleleft \Sigma'$

Changing notation, for clarity, we must show that

$\Gamma; \Sigma \bullet \Phi; \text{let } x = e \text{ in } t \mapsto \Psi_3 \bullet \Phi; T_2; \Phi_3$

with $\Sigma' = \Psi_3, \Phi_3$ and $\Psi_3 \subseteq \Sigma$.

Applying the induction hypothesis to the first hypothesis of T-Let gives

$\Gamma; \Sigma \bullet \Phi; e \mapsto \Psi_2 \bullet \Phi; T_1; \Phi_2$ \hspace{1cm} (D.1)

with $\Sigma'' = \Psi_2, \Phi_2$ and $\Psi_2 \subseteq \Sigma$.

Applying the induction hypothesis to the second hypothesis of T-Let gives

$\Gamma, x: T_1; \Sigma'' \bullet \Phi; t \mapsto \Psi_1 \bullet \Phi; T_2; \Phi_1$ \hspace{1cm} (D.2)

with $\Sigma' = \Psi_1, \Phi_1$ and $\Psi_1 \subseteq \Sigma''$.

Because $\Sigma \bullet \Phi$ does not contain $\bot$, we have $\Sigma \bullet \Phi = \Sigma, \Phi$.

Because $\Psi_2 \subseteq \Sigma$, $\Psi_2 \bullet \Phi$ does not contain $\bot$ and therefore $\Psi_2 \bullet \Phi = \Psi_2, \Phi$.

Because we assume that any new channels in $\Sigma''$ have fresh names, $\Sigma'' \bullet \Phi = \Sigma'', \Phi = \Psi_2, \Phi_2, \Phi$.

We can now construct an instance of C-Let from D.1 and D.2, rewritten according to the above observations:

$\Gamma; \Sigma, \Phi; e \mapsto \Psi_2, \Phi; T_1; \Phi_2 \quad \Gamma, x: T_1; \Psi_2, \Phi, \Phi_2; t \mapsto \Phi_1, \Phi; T_2; \Phi_1$

$\Gamma; \Sigma, \Phi; \text{let } x = e \text{ in } t \mapsto (\Psi_2, \Phi) \cap (\Psi_1, \Phi); T_2; (\Phi_2 \cap (\Psi_1, \Phi), \Phi_1$

In the conclusion we have

$$(\Psi_2, \Phi) \cap (\Psi_1, \Phi) = (\Psi_2 \cap \Psi_1), \Phi$$

$$= (\Psi_2 \cap \Psi_1) \bullet \Phi$$

and it remains to show that

$\Sigma' = \Psi_2 \cap \Psi_1, \Phi_2 \cap (\Psi_1, \Phi), \Phi_1$
and
\[ \Psi_2 \cap \Psi_1 \subseteq \Sigma. \]
The latter equation is immediate because \( \Psi_2 \subseteq \Sigma \). For the former, first note that \( \Phi_2 \) and \( \Phi \) are disjoint, because \( \Phi_2 \subseteq \Sigma'' \) and any channel names which are in \( \Sigma'' \) but not \( \Sigma \) are assumed to be fresh and therefore different from any channel names in \( \Phi \); we already know that \( \Phi \) and \( \Sigma \) are disjoint. Now we calculate
\[
\begin{align*}
\Psi_2 \cap \Psi_1, \Phi_2 \cap \Psi_1, \Phi_1 &= (\Psi_2, \Phi_2) \cap \Psi_1, \Phi_1 \\
&= \Sigma'' \cap \Psi_1, \Phi_1 \\
&= \Psi_1, \Phi_1 \text{ because } \Psi_1 \subseteq \Sigma'' \\
&= \Sigma'
\end{align*}
\]
as required.

We are finally in a position to prove completeness.

**PROOF.** [Theorem 5, page 22] By induction on the derivation of \( \Delta \vdash \Sigma \triangleright C \), considering a case for each typing rule.

For T-Thread we have
\[
\Delta \vdash \Sigma \triangleright t; T < \emptyset \quad \frac{}{\Delta \vdash \Sigma \triangleright \{t\}}
\]
By Lemma 24 we have
\[
\Delta; \Sigma \cdot \Sigma'; t \mapsto \Sigma'; T; \emptyset
\]
and rule C-Thread gives the required
\[
\Delta; \Sigma \cdot \Sigma'; \{t\} \mapsto \Sigma'.
\]

For T-Par we have
\[
\Delta \vdash \Sigma \triangleright C_1 \quad \Delta \vdash \Sigma \triangleright C_2 \quad \frac{}{\Delta \vdash \Sigma \triangleright \Sigma_1 \cdot \Sigma_2 \triangleright C_1 | C_2}
\]
By the induction hypothesis we have
\[
\Delta; \Sigma_1 \cdot \Sigma_2 \cdot \Sigma'; C_1 \mapsto \Sigma_2 \cdot \Sigma'
\]
and
\[
\Delta; \Sigma_2 \cdot \Sigma'; C_2 \mapsto \Sigma'
\]
and by using rule C-Par we obtain
\[
\Delta; \Sigma_1 \cdot \Sigma_2 \cdot \Sigma'; C_1 | C_2 \mapsto \Sigma'
\]

47
as required.

The case of T-NEWN follows directly from the induction hypothesis.