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 π-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, and our function types not only specify argument and result types but also describe changes in channels. We formalize the syntax, semantics and type checking system of our language, and prove subject reduction and runtime type safety theorems.

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

⋆ A revised and extended version of the paper in CONCUR 2004, volume 3170 of LNCS, pages 497–511, Springer-Verlag, 2004.

Email addresses: vv@di.fc.ul.pt (Vasco T. Vasconcelos), simon@dcs.gla.ac.uk (Simon J. Gay), amar@math.ist.utl.pt (António Ravara).

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, 13], or on programs written in a type-safe variant of C [18, 19]. Another approach to proving properties of protocols or their correct implementation comes from the general setting of the π-calculus [26, 32], and includes type and effect systems to check correspondence assertions [2, 3, 17], the approximation of the behaviour of π-processes by CCS terms [6, 23, 30], and session types to describe structured communication programming [14, 15, 20, 21, 33, 40].

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. Further properties, like authentication or data integrity and correct propagation, may be statically checked by combining session types with correspondence assertions.

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 [34, 35] using Gay and Hole's theory of subtyping [14, 15] to formalise compatibility and substitutability of components. Session types have also been encoded in the Haskell programming language [27]. 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, 10] have proposed a minimal object-oriented language with session types.

Our contribution to the problem of structured communication-based programming in general, and the verification of protocols in particular, is to transfer the concept of session types from the π-calculus to a multi-threaded functional language with side-effecting input/output operations. More precisely, we pro-

vide a static, decidable, discipline to ensure that a protocol implementation — code in a programming language — is correct with respect to its specification as a (set of) session type(s).

This shows that static checking of session types could be added to a language such as Concurrent ML [31] (at least without imperative features) or Concurrent Haskell [28] (cf. [27]). 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 π-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 [16], we have presented a language supporting typed functional programming with inter-process communication channels, but we only considered individual processes in isolation. In [36], we addressed collections of functional threads communicating via (session) channels created from shared names. Since we considered a concurrent scenario, the type system, compared with the previous one, is more complex.

Here we introduce a type checking algorithm for the language, and present the proofs for subject reduction and type safety.

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. In Sections 7 and 8, 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

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

in which ? means *receive*, ! means *send*, dot (.) is *sequencing*, and End indicates the end of the session. Note that the type does not correspond precisely to the specification, because it does not state that the server calculates the sum. We only use typechecking to verify observance of the protocol, not functional correctness. 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:

$$\begin{aligned} \mathit{server}\ u = {} & \mathsf{let}\ x = \mathsf{receive}\ u\ \mathsf{in} \\ & \mathsf{let}\ y = \mathsf{receive}\ u\ \mathsf{in} \\ & \mathsf{send}\ x + y\ \mathsf{on}\ u \end{aligned}$$

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

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

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

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

$$\begin{aligned} \mathit{client}\ u = {} & \mathsf{send}\ 2\ \mathsf{on}\ u \\ & \mathsf{send}\ 3\ \mathsf{on}\ u \\ & \mathsf{let}\ x = \mathsf{receive}\ u\ \mathsf{in}\ \mathit{code} \end{aligned}$$

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

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

replies with an integer which is its negation. In either case, the server then closes the connection.

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

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

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

$$\begin{aligned} server\ u = \mathsf{case}\ u\ \mathsf{of}\ \{ & \\ & add \Rightarrow \mathsf{send}\ (\mathsf{receive}\ u) + (\mathsf{receive}\ u)\ \mathsf{on}\ u \\ & neg \Rightarrow \mathsf{send}\ -(\mathsf{receive}\ u)\ \mathsf{on}\ u\ \} \end{aligned}$$

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

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

A client implementation makes a particular choice, for example:

$$\begin{aligned} addClient\ u = \mathsf{select}\ add\ \mathsf{on}\ u & \\ \mathsf{send}\ 2\ \mathsf{on}\ u & \\ \mathsf{send}\ 3\ \mathsf{on}\ u & \\ \mathsf{let}\ x = \mathsf{receive}\ u\ \mathsf{in} & \\ code & \end{aligned} \qquad \begin{aligned} negClient\ u = \mathsf{select}\ neg\ \mathsf{on}\ u & \\ \mathsf{send}\ 7\ \mathsf{on}\ u & \\ \mathsf{let}\ x = \mathsf{receive}\ u\ \mathsf{in} & \\ code & \end{aligned}$$

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 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.

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

This example shows that session types allow the description of protocols which cannot easily be accommodated with objects: a server invoking a method (*ok* or *error*) on the client, as opposed to the traditional sequence 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:

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

The server is now implemented by a recursive function, in which the positions of the recursive calls correspond to the recursive occurrences of S in the type definition. To simplify the theory we decided not to include recursive types in this paper; the interested reader may refer to report [16].

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 $\mathsf{Chan}\ c \to \mathsf{Unit}$. 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*.

$$\begin{aligned} \mathit{server} &:: (c\colon \&\langle \mathit{add}\colon \ldots, \mathit{neg}\colon \ldots\rangle; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End}) \\ \mathit{server}\ u &= \mathsf{case}\ u\ \mathsf{of}\ \{\mathit{add} \Rightarrow \ldots, \mathit{neg} \Rightarrow \ldots\} \end{aligned}$$

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]

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

[1] We often omit the empty channel environment on each side of the arrow, so that $\mathsf{Int} \to \mathsf{Bool}$ is short for $\emptyset; \mathsf{Int} \to \mathsf{Bool}; \emptyset$.

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

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

A client which requires a primality test service (perhaps the server has fast hardware) can be written as follows.

$$\begin{aligned} primeClient &:: (c\colon \oplus\langle add\colon \ldots, neg\colon \ldots, eval\colon \ldots\rangle; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End}) \\ primeClient\ u &= \mathsf{select}\ eval\ \mathsf{on}\ u \\ &\quad\ \mathsf{send}\ isPrime\ \mathsf{on}\ u \\ &\quad\ \mathsf{send}\ bigNumber\ \mathsf{on}\ u \\ &\quad\ \mathsf{let}\ x = \mathsf{receive}\ u\ \mathsf{in}\ code \end{aligned}$$

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 [33], and propose a pair of constructs: request v for use by clients, and accept v for use by servers. In use, request and accept occur in separate threads, and interact with each other to create a new channel. The value v in both request and accept, denotes the common knowledge of the two threads: a *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 *server* and *negClient* now receive a name of type $[\&\langle add\colon \ldots, neg\colon \ldots, eval\colon \ldots\rangle]$, as shown in the following piece of code.

$$\begin{aligned} server' &:: [\&\langle add\colon \ldots, neg\colon \ldots, eval\colon \ldots\rangle] \to \mathsf{Unit} \\ server'\ x &= \mathsf{let}\ u = \mathsf{accept}\ x\ \mathsf{in}\ (server\ u; \mathsf{close}\ u) \\ negClient' &:: [\&\langle add\colon \ldots, neg\colon \ldots, eval\colon \ldots\rangle] \to \mathsf{Unit} \\ negClient'\ x &= \mathsf{let}\ u = \mathsf{request}\ x\ \mathsf{in}\ (negClient\ u; \mathsf{close}\ u) \end{aligned}$$

Note that the same type for the shared name x is used both for the server and for the client; it is the accept/request construct that distinguishes one from the other. This is also where we introduce the operation to close a channel: accept/request creates a channel; 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 new S. To distribute it to a second thread, we fork a new thread, in whose code the name occurs. [2] Our

[2] Alternatively, we may send the name on an existing channel.

complete system creates a name x and launches three threads (a server and two clients), all sharing the newly created name.

$$
\begin{aligned}
&\mathit{system} :: \mathsf{Unit} \\
&\mathit{system} = \mathsf{let}\ x = \mathsf{new}\ \&\langle \mathit{add} \colon \ldots, \mathit{neg} \colon \ldots, \mathit{eval} \colon \ldots \rangle\ \mathsf{in} \\
&\qquad\qquad \mathsf{fork}\ \mathit{negClient}\ x;\ \mathsf{fork}\ \mathit{addClient}\ x;\ \mathsf{fork}\ \mathit{server}\ x
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{server} :: [\&\langle \mathit{add} \colon \ldots, \mathit{neg} \colon \ldots, \mathit{eval} \colon \ldots \rangle] \to \mathsf{Unit} \\
&\mathit{server}\ x = \mathsf{let}\ u = \mathsf{accept}\ x\ \mathsf{in} \\
&\qquad\qquad \mathsf{fork}\ (\mathsf{case}\ u\ \mathsf{of}\ \ldots; \mathsf{close}\ u); \mathit{server}\ x
\end{aligned}
$$

Sending Channels on Channels. Imagine two clients which need to cooperate in their interaction with the server: one client establishes a connection, selects the *neg* operation, and sends the argument; the second client receives the result. After selecting *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 ?(?Int.End).End (called S below) and establish a connection for the sole purpose of transmitting the server channel.

$$
\begin{aligned}
&\mathit{askNeg} :: [\langle \mathit{add} \colon \ldots \rangle] \to\ [S] \to \mathsf{Unit} \\
&\mathit{askNeg}\ x\ y = \mathsf{let}\ u = \mathsf{request}\ x\ \mathsf{in} \\
&\qquad\qquad \mathsf{select}\ \mathit{neg}\ \mathsf{on}\ u; \mathsf{send}\ 7\ \mathsf{on}\ u \\
&\qquad\qquad \mathsf{let}\ w = \mathsf{request}\ y\ \mathsf{in} \\
&\qquad\qquad \mathsf{send}\ u\ \mathsf{on}\ w; \mathsf{close}\ w
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathit{getNeg} :: [S] \to \mathsf{Unit} \\
&\mathit{getNeg}\ y = \mathsf{let}\ w = \mathsf{accept}\ y\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ u = \mathsf{receive}\ w\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ i = \mathsf{receive}\ u\ \mathsf{in} \\
&\qquad\qquad \mathsf{close}\ u; \mathsf{close}\ w; \mathit{code}
\end{aligned}
$$

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 $S = ?(?\mathsf{Int}.\mathsf{End}).\mathsf{End}$; after the second line, d is reduced to End, but c shows up with type ?Int.End; after the third line both channels are of type 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 *askNeg* cannot close u, for otherwise the channel's client side would be closed twice (in *askNeg* and in *getNeg*). On the other hand, channel w must be closed at both its ends, by *askNeg* and by *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.

$$\mathit{sendSend}\ u\ v = \ \mathsf{send}\ 1\ \mathsf{on}\ u; \mathsf{send}\ 2\ \mathsf{on}\ v$$

Function *sendSend* can be used in a number of different ways, including the one where u and v become aliases for a single underlying channel.

$$\begin{aligned} \mathit{sendTwice} &:: c\colon \mathsf{!Int.!Int.End}; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End} \\ \mathit{sendTwice}\ w &= \mathit{sendSend}\ w\ w \end{aligned}$$

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 *sendSend*, the types of u and v are both $\mathsf{Chan}\ c$. The state of c, initially $\mathsf{!Int.!Int.End}$, is recorded separately.

Free Variables in Functions. If we write

$$\mathit{sendFree}\ v = \ \mathsf{send}\ 1\ \mathsf{on}\ u; \mathsf{send}\ 2\ \mathsf{on}\ v$$

then function *sendSend* becomes $\lambda u.\mathit{sendFree}$. In order to type *sendTwice*, thus effectively aliasing u and v in *sendSend*, we must have [3]

$$\begin{aligned} \mathit{sendFree} &:: c\colon \mathsf{!Int.!Int.End}; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End} \\ \mathit{sendSend} &:: c\colon \mathsf{!Int.!Int.End}; \mathsf{Chan}\ c \to \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End} \end{aligned}$$

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

$$\begin{aligned} \mathit{sendFree} &:: c\colon \mathsf{!Int.End}, d\colon \mathsf{!Int.End}; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End}, d\colon \mathsf{End} \\ \mathit{sendSend} &:: c\colon \mathsf{!Int.End}, d\colon \mathsf{!Int.End}; \mathsf{Chan}\ c \to \mathsf{Chan}\ d \to \mathsf{Unit}; c\colon \mathsf{End}, d\colon \mathsf{End} \end{aligned}$$

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

[3] We abbreviate $\Sigma; T \to (\Sigma; U \to V; \Sigma'); \Sigma'$ to $\Sigma; T \to U \to V; \Sigma'$.

$$
\begin{array}{rcl}
C & ::= & \langle t \rangle \mid (C \mid C) \mid (\nu x\colon [S])C \mid (\nu\gamma)C \\
t & ::= & v \mid \mathsf{let}\ x = e\ \mathsf{in}\ t \mid \mathsf{fork}\ t; t \\
e & ::= & t \mid vv \mid \mathsf{new}\ S \mid \mathsf{accept}\ v \mid \mathsf{request}\ v \mid \mathsf{send}\ v\ \mathsf{on}\ v \mid \mathsf{receive}\ v \mid \\
 & & \quad \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mid \mathsf{select}\ l\ \mathsf{on}\ v \mid \mathsf{close}\ v \\
v & ::= & \alpha \mid \lambda(\Sigma; x\colon T).e \mid \mathsf{rec}\ (x\colon T).v \mid \mathsf{unit} \\
\alpha & ::= & x \mid \gamma^p \\
p & ::= & + \mid - \\
T & ::= & D \mid \mathsf{Chan}\ \alpha \\
D & ::= & [S] \mid \Sigma; T \to T; \Sigma \mid \mathsf{Unit} \\
S & ::= & ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \&\langle l_i\colon S_i\rangle_{i \in I} \mid \oplus\langle l_i\colon S_i\rangle_{i \in I} \mid \mathsf{End} \\
\Sigma & ::= & \emptyset \mid \Sigma, \alpha\colon S \quad (\alpha\colon S' \text{ not in } \Sigma) \\
\Gamma & ::= & \emptyset \mid \Gamma, x\colon T \quad (x\colon T' \text{ not in } \Gamma)
\end{array}
$$

Fig. 1. Syntax

Polymorphism. We have seen that *sendFree* admits at least two different types, a *share/not-share* kind of polymorphism. Other forms include *channel polymorphism* and *session polymorphism*. For an example of channel polymorphism, consider

$$
\begin{array}{l}
\mathit{sendTwiceSendTwice} :: c\colon S, d\colon S; \mathsf{Chan}\ c \to \mathsf{Chan}\ d \to \mathsf{Unit}; c\colon \mathsf{End}, d\colon \mathsf{End} \\
\mathit{sendTwiceSendTwice}\ x\ y = \ \mathit{sendTwice}\ x; \mathit{sendTwice}\ y
\end{array}
$$

where S is $!\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$. Here *sendTwice* must be typed once with channel c, and another with channel d. For an example of session polymorphism (indeed a variant of sharing polymorphism identified in the previous paragraph), we have:

$$
\begin{array}{l}
\mathit{sendQuad} :: c\colon !\mathsf{Int}.!\mathsf{Int}.!\mathsf{Int}.!\mathsf{Int}.\mathsf{End}; \mathsf{Chan}\ c \to \mathsf{Unit}; c\colon \mathsf{End} \\
\mathit{sendQuad}\ x = \mathit{sendTwice}\ x; \mathit{sendTwice}\ x
\end{array}
$$

where *sendTwice* must be typed once with $c\colon !\mathsf{Int}.!\mathsf{Int}.!\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$, and a second time with $c\colon !\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$. For simplicity we decided not to incorporate any form of polymorphism in our type system.

3 Syntax

One aspect of the syntax of our language has not been illustrated by the examples in Section 2. Channels are *runtime entities*, created when a request meets an accept. In order to define the operational semantics (Section 4) it is necessary to distinguish between the two ends of a channel, represented by

polarities. If γ is a channel, its two ends are written γ^+ and γ^-. Channels are not supposed to occur in a top-level program. When writing programs such as those in Section 2, we use conventional program variables to denote channels. Such variables may then be replaced by actual channels.

The syntax of our language is defined formally by the grammar in Figure 1. We use *term variables* $x, \ldots$, *labels* $l, \ldots$, and *channels* $\gamma, \ldots$, and define *configurations* C, *threads* t, *expressions* e, and *values* v. We write γ^p for a polarized channel, where p represents the polarity. Duality on polarities, written $\overline{p}$, exchanges $^+$ and $^-$. The dual polarities γ^+ and γ^- represent the opposite ends of channel γ.

Most of the syntactic constructs for expressions have been illustrated in Section 2. Threads comprise stacks of expressions waiting for evaluation (cf. [24]), possibly with occurrences of the fork primitive. Threads establish new sessions (create new channels γ) by synchronising, via accept/request, on *shared names*. Shared names are no different from ordinary term variables, but we usually use the letter n when referring to them. 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, $(\nu x\colon [S])C$; and declaration of a channel, $(\nu\gamma)C$. To support typechecking, we annotate name declaration with its type, and λ-abstractions with a channel environment as well as a typed argument. A conditional expression, if v then e else e, together with its corresponding values true, false, and datatype, Bool, could be easily added.

Declaration of a typed name has not been illustrated yet. It shows up during 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:

$$\begin{aligned} \mathit{systemConf} = (\nu n\colon [\&\langle \mathit{add}\colon \ldots, \mathit{neg}\colon \ldots, \mathit{eval}\colon \ldots\rangle]) \\ (\langle \mathit{negClient}\ n\rangle \mid \langle \mathit{negClient}\ n\rangle \mid \langle \mathit{server}\ n\rangle) \end{aligned}$$

This configuration arises during reduction of the thread $\langle \mathit{system} \rangle$ 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 also described in Figure 1. We define *term types* T, *data types* D, *session types* S, and *channel environments* Σ. The type Chan α represents the type of the channel with identity α; the session type associated with α is recorded separately in a channel environment Σ. In the process of reduction, program variables may be replaced by (runtime) channels, implying that type Chan c may become, say, Chan γ^+. Among datatypes we have channel-state annotated functional types $\Sigma; T \to T; \Sigma$, and types for names $[S]$ capable of establishing sessions of type S.

$$(C, |, \langle \mathsf{unit} \rangle) \text{ is a commutative monoid} \qquad \text{(S-Monoid)}$$
$$(\nu x\colon T)C_1 \mid C_2 \equiv (\nu x\colon T)(C_1 \mid C_2) \quad \text{if } x \text{ not free in } C_2 \qquad \text{(S-ScopeN)}$$
$$(\nu \gamma)C_1 \mid C_2 \equiv (\nu \gamma)(C_1 \mid C_2) \quad \text{if } \gamma \text{ not free in } C_2 \qquad \text{(S-ScopeC)}$$
$$(\nu \gamma)(\nu x\colon T)C \equiv (\nu x\colon T)(\nu \gamma)C \quad \text{if } \gamma \text{ not free in } T \qquad \text{(S-ScopeCN)}$$

Fig. 2. Structural congruence

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

4 Operational Semantics

The binding occurrences are the variable x in $\lambda(\Sigma; x\colon T).e$, in $\mathsf{rec}\ (x\colon T).e$, in $\mathsf{let}\ x = e\ \mathsf{in}\ t$, and in $(\nu x\colon T)C$, and the channel γ in $(\nu\gamma)C$. Free and bound identifiers are defined as usual and we work up to α-equivalence. We define a reduction semantics on configurations (Figure 3), making use of a simple structural congruence relation [26] (Figure 2), allowing for the rearrangement of threads in a configuration, so that reduction may happen. Substitution of values for variables is defined as expected.

R-Init synchronises two threads on a shared name n, creating a new channel γ known to both threads. One thread gets γ^+ and the other gets γ^-, representing the opposite ends of the channel. Rules R-Com, R-Branch, and R-Close synchronise two threads on a *channel* γ: 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. In each case, one thread has γ^+ and the other has γ^-. R-New creates a new name n, and records the fact that the name is potentially shared, by means of a $(\nu n\colon [S])$ 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.

$$\langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2\rangle \to (\nu\gamma)(\langle \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2\rangle) \quad \text{(R-Init)}$$

$$\langle \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t_1\rangle\langle \mathsf{let}\ y = \mathsf{send}\ v\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \to \langle \mathsf{let}\ x = v\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle \quad \text{(R-Com)}$$

$$\langle \mathsf{let}\ x = \mathsf{case}\ \gamma^p\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{select}\ l_j\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \to \langle \mathsf{let}\ x = e_j\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle \quad \text{(R-Branch)}$$

$$\langle \mathsf{let}\ x = \mathsf{close}\ \gamma^p\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{close}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \to \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle \quad \text{(R-Close)}$$

$$\langle \mathsf{let}\ x = \mathsf{new}\ S\ \mathsf{in}\ t\rangle \to (\nu n\colon [S])\langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle \quad \text{(R-New)}$$

$$\langle \mathsf{fork}\ t_1; t_2\rangle \to \langle t_1\rangle \mid \langle t_2\rangle \quad \text{(R-Fork)}$$

$$\langle \mathsf{let}\ x = (\lambda(\Sigma; y\colon T).e)v\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t\rangle \quad \text{(R-App)}$$

$$\langle \mathsf{let}\ x = (\mathsf{rec}\ (y\colon T).v)u\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ x = (v\{\mathsf{rec}\ (y\colon T).v/y\})u\ \mathsf{in}\ t\rangle \quad \text{(R-Rec)}$$

$$\langle \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ t')\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ y = e\ \mathsf{in}\ (\mathsf{let}\ x = t'\ \mathsf{in}\ t)\rangle \quad \text{(R-Let)}$$

$$\langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle \to \langle t\{v/x\}\rangle \quad \text{(R-Beta)}$$

$$\frac{C \to C'}{(\nu\gamma)C \to (\nu\gamma)C'} \quad \text{(R-NewC)}$$

$$\frac{C \to C'}{(\nu n\colon T)C \to (\nu n\colon T)C'} \quad \text{(R-NewN)}$$

$$\frac{C \to C'}{C \mid C'' \to C' \mid C''} \quad \text{(R-Par)}$$

$$\frac{C \equiv \to \equiv C'}{C \to C'} \quad \text{(R-Cong)}$$

In R-Init, γ is not free in t_1, t_2; in R-New, n is not free in t.

Fig. 3. Reduction rules

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

$$\begin{aligned}
&\langle system\rangle && (1)\\
\to^*\ &(\nu n)\ \langle \mathsf{fork}\ negClient\ n; \mathsf{fork}\ addClient\ n; \mathsf{fork}\ server\ n\rangle && (2)\\
\to^*\ &(\nu n)(\ \langle \mathsf{let}\ u = \mathsf{request}\ n\ \mathsf{in}\ \ldots\rangle \mid \\
&\qquad \langle \mathsf{let}\ u = \mathsf{request}\ n\ \mathsf{in}\ \ldots\rangle \mid \langle \mathsf{let}\ u = \mathsf{accept}\ n\ \mathsf{in}\ \ldots\rangle) && (3)\\
\to^*\ &(\nu n)(\ \langle \mathsf{let}\ u = \mathsf{request}\ n\ \mathsf{in}\ \ldots\rangle \mid \\
&\qquad (\nu\gamma)(\langle \mathsf{select}\ neg\ \mathsf{on}\ \gamma^+; \ldots\rangle \mid \langle \mathsf{case}\ \gamma^-\ \mathsf{of}\ \ldots\rangle)) && (4)\\
\to^*\ &(\nu n)(\ \langle \mathsf{let}\ u = \mathsf{request}\ n\ \mathsf{in}\ \ldots\rangle \mid (\nu\gamma)(\langle \mathsf{close}\ \gamma^+\rangle \mid \langle \mathsf{close}\ \gamma^-\rangle)) && (5)\\
\to^*\ &(\nu n)\ \langle \mathsf{let}\ u = \mathsf{request}\ n\ \mathsf{in}\ \ldots\rangle && (6)
\end{aligned}$$

$$\overline{\mathsf{End}} = \mathsf{End} \quad \overline{?D.S} = !D.\overline{S} \quad \overline{?S'.S} = !S'.\overline{S} \quad \overline{!D.S} = ?D.\overline{S} \quad \overline{!S'.S} = ?S'.\overline{S}$$
$$\overline{\&\langle l_i\colon S_i\rangle_{i\in I}} = \oplus\langle l_i\colon \overline{S_i}\rangle_{i\in I} \qquad \overline{\oplus\langle l_i\colon S_i\rangle_{i\in I}} = \&\langle l_i\colon \overline{S_i}\rangle_{i\in I}$$

Fig. 4. Duality on session types

In line (3) we have two threads competing for request*ing a session on name n. We have chosen the negClient to go ahead; a new channel γ, known only to the negClient and to the server, is created. The client gets γ^+ and the server gets γ^-. In line (4) we have a typical interaction between two threads on a common channel. Similar interactions continue until both ends of the channel 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 5–7, where Σ, Σ' is the pointwise extension of $\Sigma, \alpha\colon S$, as defined in Figure 1. Typing judgements for constants are of the form $\Gamma; v \mapsto T$, where Γ is a map from variables to types. Value judgements do not mention channel environments, for values, having no behaviour, do not change channels. Judgements for expressions are of the form $\Gamma; \Sigma; e \mapsto \Sigma'; T; \Sigma''$, where channel environment Σ' describes the *unused* part of Σ, whereas Σ'' represents the final types of the channels which are used by expression e. Channels which are created by e appear in Σ''. If $c\colon S \in \Sigma$ then we will have either $c\colon S \in \Sigma'$ if e does not use c, or $c\colon S' \in \Sigma''$ if e uses c and leaves it with type S', or neither if $S = \mathsf{End}$ and e closes c. For example

$$x\colon \mathsf{Chan}\ c; c\colon ?\mathsf{Int}.\mathsf{End}, d\colon !\mathsf{Int}.\mathsf{End}; \mathsf{receive}\ x \mapsto d\colon !\mathsf{Int}.\mathsf{End}; \mathsf{Int}; c\colon \mathsf{End}.$$

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

The typing rules can be interpreted directly as a typechecking algorithm, by reading sequents $X \mapsto Y$ as the description of a function that performs a pattern matching on input configuration, expression, or value in X, and outputs Y. We restrict the type checking algorithm to *top level* programs, containing no channel values, hence no channel bindings $(\nu\gamma)C$. For these programs, a straightforward analysis of the rules reveals that no backtracking is needed.

The splitting of the input channel environment in X (in most of the rules)

$$\Gamma; \mathsf{unit} \mapsto \mathsf{Unit} \qquad \text{(C-Const)}$$

$$\Gamma; \gamma^p \mapsto \mathsf{Chan}\ \gamma^p \qquad \text{(C-Chan)}$$

$$\Gamma, x\colon T; x \mapsto T \qquad \text{(C-Var)}$$

$$\frac{\Gamma, x\colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2}{\Gamma; \lambda(\Sigma; x\colon T).e \mapsto (\Sigma; T \to U; \Sigma_1, \Sigma_2)} \qquad \text{(C-Abs)}$$

$$\frac{\Gamma, x\colon T; v \mapsto T \qquad T = (\Sigma; U \to U'; \Sigma')}{\Gamma; \mathsf{rec}\ (x\colon T).v \mapsto T} \qquad \text{(C-Rec)}$$

Fig. 5. Type checking values

should only be performed after the auxiliary calls have been made. For example, for a goal of the form $(\Gamma; \Sigma; \mathsf{receive}\ v)$, we call the function on $(\Gamma; v)$, check that the output is of form $\mathsf{Chan}\ \alpha$. Then, we split Σ if possible, and terminate as prescribed by C-ReceiveD or C-ReceiveS, according to the type of α in Σ.

The algorithm relies on the ability to create new identifiers (rules C-Accept, C-Request, and C-ReceiveS), as well as to build a type $\overline{S}$ from type S (rule C-Request). The latter operation is straightforward from the definition in Figure 4.

The presentation of the typechecking algorithm, in the form of inference rules, follows a general pattern for type systems involving linear types [38].

Type Checking Values (Figure 5). C-Chan says that a polarized channel γ^p has type $\mathsf{Chan}\ \gamma^p$. The actual type (or state) of channel γ^p is to be found in a channel environment Σ, in the rules for expressions. In C-Abs, the initial and final channel environments of the function body are recorded in the function type.

Lemma 4.2 (page 26) guarantees that, in judgement $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$, Σ_1 and Σ_2 have disjoint domains, so that Σ_1, Σ_2 is defined and the overall transformation of channel types can be described by the function type $\Sigma; T \to U; \Sigma_1, \Sigma_2$.

Type Checking Expressions (Figure 6). Channels allow for the transmission of different kinds of values: for the purpose of type checking, we distinguish between sending/receiving channels, and sending/receiving the remaining values (constants and functions). As such, we find in Figure 6 two rules for receive and two rules for send, that can be selected based on the type for α.

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon ?D.S; \mathsf{receive}\ v \mapsto \Sigma; D; \alpha\colon S} \quad \text{(C-ReceiveD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad c \text{ fresh}}{\Gamma; \Sigma, \alpha\colon ?S'.S; \mathsf{receive}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S', \alpha\colon S} \quad \text{(C-ReceiveS)}$$

$$\frac{\Gamma; v \mapsto D \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !D.S; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \beta \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !S'.S, \beta\colon S'; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendS)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad j \in I}{\Gamma; \Sigma, \alpha\colon \oplus\langle l_i\colon S_i\rangle_{i\in I}; \mathsf{select}\ l_j\ \mathsf{on}\ v \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S_j} \quad \text{(C-Select)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad \forall j \in I.(\Gamma; \Sigma, \alpha\colon S_j; e_j \mapsto \Sigma_1; T; \Sigma_2)}{\Gamma; \Sigma, \alpha\colon \&\langle l_i\colon S_i\rangle_{i\in I}; \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I} \mapsto \Sigma_1; T; \Sigma_2} \quad \text{(C-Case)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon \mathsf{End}; \mathsf{close}\ v \mapsto \Sigma; \mathsf{Unit}; \emptyset} \quad \text{(C-Close)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{accept}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S} \quad \text{(C-Accept)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{request}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon \overline{S}} \quad \text{(C-Request)}$$

$$\frac{\Gamma; v \mapsto T}{\Gamma; \Sigma; v \mapsto \Sigma; T; \emptyset} \quad \text{(C-Val)}$$

$$\frac{\Gamma; v \mapsto (\Sigma; T \to U; \Sigma') \qquad \Gamma; v' \mapsto T}{\Gamma; \Sigma, \Sigma''; vv' \mapsto \Sigma''; U; \Sigma'} \quad \text{(C-App)}$$

$$\Gamma; \Sigma; \mathsf{new}\ S \mapsto \Sigma; [S]; \emptyset \quad \text{(C-New)}$$

$$\frac{\Gamma; \Sigma; e \mapsto \Sigma_1; T_1; \Sigma_1' \quad \Gamma, x\colon T_1; \Sigma_1, \Sigma_1'; t \mapsto \Sigma_2; T_2; \Sigma_2'}{\Gamma; \Sigma; \mathsf{let}\ x = e\ \mathsf{in}\ t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma_1' \cap \Sigma_2), \Sigma_2'} \quad \text{(C-Let)}$$

$$\frac{\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \emptyset \qquad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \emptyset}{\Gamma; \Sigma; (\mathsf{fork}\ t_1; t_2) \mapsto \Sigma_2; T_2; \emptyset} \quad \text{(C-Fork)}$$

Fig. 6. Type checking expressions

In C-ReceiveD, the prefix $?D$, of the type for channel α, is consumed, provided that we are receiving on a value aliased to α (that is a value of type $\mathsf{Chan}\ \alpha$). In C-ReceiveS, we receive a channel, that we decided to call c; the type of the expression is $\mathsf{Chan}\ c$, and we add a new entry to the final channel environment, where we record the type for c. The rules C-SendS and C-SendD, for sending channels and the remaining values, are similar.

In C-Select, the type for α in the final channel environment is that of branch l_j in the type for α in the source channel environment. In C-Case, all branches must produce the same final channel environment. This enables us to know the environment for any code following the case, independently of which branch is chosen at runtime. The same would apply to the two branches of a conditional

expression. Rule C-Close requires that the channel must be ready to be closed (of type End). We remove the closed channel from the environment.

Rules C-Request and C-Accept both introduce a new channel c in the channel environment, of *dual* types [14, 20, 21, 33, 35, 40]. At runtime the two occurrences of c are instantiated by the opposite ends (γ^+, γ^-) of some new channel γ. The dual of a session type S, denoted $\overline{S}$, is defined for all session types, 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 4.

Rule C-Val says that constants do not affect the state of channels. In C-App, the initial environment in the type of the function must be present in the overall environment, and is replaced by the final environment from the function type. Rule C-New specifies that the expression 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.

The most complex type checking rule is C-Let. The channel environment Σ_1, Σ_1' which is used when typechecking t consists of the channels (Σ_1) which are not used by e and the updated types (Σ_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 (Σ_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. Of the initial channels in Σ, some are consumed by t_1 and the remainder, Σ_1, are used to typecheck t_2. Any channels remaining after typechecking t_2, i.e. Σ_2, are returned as the channels which are not used by the fork expression.

Typing Configurations (Figure 7). Rule C-Thread requires threads to consume their channels, similarly to C-Fork. The rule requires, via a type environment of the form $\vec{x}$: $[\vec{S}]$, that the only free variables in a thread are those standing for names, used to initiate new sessions via accept and request. Type safety (Theorem 3) relies on this guarantee.

In rule C-Par, we type check C_1 and feed the output Σ_1, the unused part of Σ, into another call, this time for C_2. The output Σ_2 of the second call is the unused channel environment of the parallel composition $C_1 \mid C_2$ (cf. rule C-Fork in Figure 6).

C-NewN discards information on the bound name. There are two rules for

$$\frac{\vec{x}\colon [\vec{S}]; \Sigma; t \mapsto \Sigma'; T; \emptyset}{\vec{x}\colon [\vec{S}]; \Sigma; \langle t \rangle \mapsto \Sigma'} \quad \text{(C-Thread)}$$

$$\frac{\Gamma; \Sigma; C_1 \mapsto \Sigma_1 \quad \Gamma; \Sigma_1; C_2 \mapsto \Sigma_2}{\Gamma; \Sigma; C_1 \mid C_2 \mapsto \Sigma_2} \quad \text{(C-Par)}$$

$$\frac{\Gamma, n\colon T; \Sigma; C \mapsto \Sigma'}{\Gamma; \Sigma; (\nu n\colon T)C \mapsto \Sigma'} \quad \text{(C-NewN)}$$

$$\frac{\Gamma; \Sigma, \gamma^+\colon S, \gamma^-\colon \overline{S}; C \mapsto \Sigma' \qquad \gamma \text{ not in } \Gamma, \Sigma, \Sigma'}{\Gamma; \Sigma; (\nu\gamma)C \mapsto \Sigma'} \quad \text{(C-NewB)}$$

$$\frac{\Gamma; \Sigma; C \mapsto \Sigma' \qquad \gamma \text{ not in } \Gamma, \Sigma, \Sigma'}{\Gamma; \Sigma; (\nu\gamma)C \mapsto \Sigma'} \quad \text{(C-NewC)}$$

Fig. 7. Type checking configurations

channel creation. Rule C-NewB says that both ends of a newly created channel must be used with dual types. Rule C-NewC means that a configuration remains typable if, during reduction, one of its channels is consumed (cf. [40] and the absence of the bottom rule).

Typability over arbitrary channel environments is not closed under reduction. For example, the configuration

$$\langle \mathsf{send\ unit\ on}\ \gamma^+ \rangle \mid \langle (\mathsf{receive}\ \gamma^-)\mathsf{unit} \rangle$$

is typable with $\Sigma = \gamma^+\colon !\mathsf{Unit.End}, \gamma^-\colon ?(\mathsf{Unit} \to \mathsf{Unit}).\mathsf{End}$, yet, it reduces to a configuration $(\langle \mathsf{unit} \rangle \mid \langle \mathsf{unit\ unit} \rangle)$ that is not typable under any channel environment. Notice however that if C is the first configuration above, then $(\nu\gamma)C$ is not typable because the types for γ^+ and γ^- are not dual.

This observation leads us to consider only channel environments where the two ends of a channel are of dual types. We call such environments *balanced* [15]. A channel environment Σ is *balanced* if whenever $\gamma^+ : S, \gamma^- : S' \in \Sigma$, then $S = \overline{S}'$. The main result of this section says that typability under balanced environments is preserved by reduction; the proof is in Appendix A, page 26.

Theorem 2 (Subject Reduction) *If $\Delta; \Sigma; C \mapsto \Sigma'$ with Σ balanced, and $C \to C'$, then there is a Σ'' such that $\Delta; \Sigma''; C' \mapsto \Sigma'$, and Σ'' is balanced.*

A previous work of ours [36] does not enjoy the Subject Reduction property, as pointed out in [9]. The problem occurs when a thread is sent one end of channel while already possessing the second end. The system in this paper fixes the problem by working with polarised channels, allowing two distinct entries (γ^+, γ^-) for the same channel γ in a channel environment.

6 Type Safety

In our language of functional communicating threads different sorts of problems may occur at runtime, ranging from the traditional error of trying to apply a value to something that is not a function; 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 γ-*thread* any thread ready to perform an operation on channel γ^p, that is a thread of the form $\langle \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t\rangle$, and similarly for send, case, select, and close. A γ-*redex* is the parallel composition of two threads ready to communicate on channel γ, i.e.

$$\langle \mathsf{let}\ x = \mathsf{send}\ v\ \mathsf{on}\ \gamma^p\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{receive}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle$$

and similarly for case/select, close/close. A *configuration* C *is faulty* when $C \equiv (\nu\vec{x}\colon \vec{T})(\nu\vec{\gamma})(C_1 \mid C_2)$ and C_1 is

(1) the thread $\langle \mathsf{let}\ x = vv'\ \mathsf{in}\ t\rangle$, where $v \neq \lambda y.e$ and $v \neq \mathsf{rec}\ (y\colon T)\_.\_$; or is
(2) the thread $\langle \mathsf{let}\ x = \mathsf{accept/request}\ v\ \mathsf{in}\ t\rangle$, where v is not a variable; or is
(3) the thread $\langle \mathsf{let}\ x = e\ \mathsf{in}\ t\rangle$, 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 γ-threads that do not form a γ-redex; or is
(5) the parallel composition of three or more γ-threads.

Typability, per se, does not guarantee type safety. A configuration containing one thread trying to send on a given channel, and another trying to select on the same channel, can be typable. For example, the following sequent is derivable with $\Sigma = \gamma^+\colon \mathsf{!Unit.End}, \gamma^-\colon \oplus\langle l\colon \mathsf{End}\rangle$.

$$\emptyset; \Sigma; \langle \mathsf{send}\ \mathsf{unit}\ \mathsf{on}\ \gamma^+; \mathsf{close}\ \gamma^+\rangle \mid \langle \mathsf{select}\ l\ \mathsf{on}\ \gamma^-; \mathsf{close}\ \gamma^-\rangle \mapsto \emptyset$$

Notice however that if C is the above configuration, then $(\nu\gamma)C$ is not typable because the types for γ^+ and γ^- are not dual.

Similarly to Subject Reduction, we consider balanced channel environments only. The main property of this section says that configurations typable in balanced environments are not faulty; the proof is in Appendix B, page 37.

Theorem 3 (Type Safety) *If* $\Gamma; \Sigma; C \mapsto \Sigma'$ *with* Σ *balanced, then* C *is not faulty.*

7 Related Work

Cyclone [19] is a C-like type-safe polymorphic imperative language. It features region-based memory management, and more recently threads and locks [18], 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" [11], 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 [17] use one such system to prove progress properties of communication protocols written in the π-calculus. Bonelli, Compagnoni, and Gunter [2, 3] combine the language of Honda, Vasconcelos and Kubo [21] 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, 30] uses CCS to describe properties of π-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 π-calculus type systems can be defined [23]. 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 [39] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [22] 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 [27], but the results are limited to type soundness.

More recently, Dezani-Ciancaglini et al. [10] proposed a minimal distributed object-oriented language with session types, where higher-order channels are not allowed. This problem is overcome in [9], whose type system also enjoys a progress property, whereby well-typed programs do not starve at communication points, once a session is established. The price to pay is the impossibility of interleaving communications on different channels, by the same thread.

The language in [9] extends a conventional class based language with session primitives (including an interesting receive-while primitive), rather than trying to integrate session primitives within the conventional OO constructs. To ensure type preservation, the language presents strong syntactic restrictions so as not to allow channel aliasing, as opposed to our system where aliasing is allowed, albeit in a (type controlled) restricted form.

8 Future Work

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

Recursive Session Types. Recursive session types have been used in a variety of works [16, 21, 40]. We feel its incorporation in the present setting would not present major difficulties.

Polymorphism. The last part of Section 2 presents examples of polymorphic functions. A previous work of ours [36] addresses polymorphism by using an untyped syntax and separately typing different copies of the polymorphic value as proposed in [29, Chapter 22]. For type-checking purposes, we use in this paper a typed syntax, hence a new approach is needed. Channel polymorphism is easy to handle with a standard Damas-Milner (ML-style, or let-polymorphism) approach [25]. Session polymorphism remains an interesting challenge.

Web services. Our work opens up the possibility of an application of session types to verification of web service implementations [7, 37]. 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, 12]). 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, Profundis, IST–2001–33100 and Sensoria, IST–2005–16004), Fundação para a Ciência e a Tecnologia (via CLC, CITI, and the projects MIMO, POSI/CHS/39789/2001, and Space–Time–Types, POSC/EIA/55582/2004), and a Treaty of Windsor grant from the British Council in Portugal and the Portuguese Council of University Rectors.

The authors are indebted to D. Mostrous [9] for pointing out a counterexample to subject-reduction in a previous version of the type system [36].

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

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

[3] E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *Electronic Notes on Theoretical Computer Science*, pages 3–22. Elsevier Science, 2005.

[4] M. J. Butler and C. Ferreira. A process compensation language. In *International Conference on Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2000.

[5] M. J. Butler and C. Ferreira. Using SPIN and STeP to verify business processes specifications. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 207–213. Springer, 2003.

[6] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Symposium on Principles of Programming Languages*, pages 45–57. ACM Press, 2002.

[7] F. Curbera, Y. Goland, J. Klein, F. Leymann, S. T. D. Roller, and S. Weerawarana. Business process execution language for web services, version 1.1. Technical report, IBM, 2003.

[8] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Conference on Programming Language Design and Implementation*, volume 36(5) of *SIGPLAN Notices*, pages 59–69. ACM Press, 2001.

[9] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *Proceedings of ECOOP'06*, Lecture Notes in Computer Science. Springer, 2005.

[10] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *Proceedings of the Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*. Springer, 2005.

[11] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Conference on Programming Language Design and Implementation*, volume 37(5) of *SIGPLAN Notices*, pages 13–24, 2002.

[12] C. Ferreira and M. J. Butler. Using B refinement to analyse compensating business processes. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 477–496. Springer, 2003.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Conference on Programming Language Design and Implementation*, volume 37(5) of *SIGPLAN Notices*, pages 1–12. ACM Press, 2002.

[14] S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.

[15] S. J. Gay and M. J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[16] S. J. Gay, A. Ravara, and V. T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow, March 2003.

[17] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1–3):379–409, 2003.

[18] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation*, volume 38(3) of *SIGPLAN Notices*, pages 13–25. ACM Press, 2003.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Conference on Programming Language Design and Implementation*, volume 37(5) of *SIGPLAN Notices*, pages 282–293. ACM Press, 2002.

[20] K. Honda. Types for dyadic interaction. In *International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[21] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

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

[23] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.

[24] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Annual Symposium on Logic in Computer Science*, pages 101–112. IEEE Computer Society Press, 2002.

[25] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[27] M. Neubauer and P. Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[28] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.

[29] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[30] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *International Symposium on Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 375–394. Springer, 2001.

[31] J. H. Reppy. CML: a higher order concurrent language. In *Conference on Programming Language Design and Implementation*, volume 26(6) of *SIGPLAN Notices*, pages 293–305. ACM Press, 1991.

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

[33] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*. Springer, 1994.

[34] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticae*, to appear. Full version of [35].

[35] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *International Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[36] V. T. Vasconcelos, A. Ravara, and S. J. Gay. Session types for functional multithreading. In *International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2004.

[37] W3C. Web services choreography requirements, W3C working draft, 2004.

[38] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1. MIT Press, 2005.

[39] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[40] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited. In *1st International Workshop on Security and Rewriting Techniques*, Electronic Notes on Theoretical Computer Science, 2006. Extended version in Technical Report DI/FCUL TR–06–9. Department of Computer Science, University of Lisbon.

A Proof of Theorem 2, Subject Reduction

We start with a few auxiliary results; the proof of Subject Reduction is on page 31. To simplify the proofs, we make use of the *variable convention* [1], allowing, for example, to assume that, in sequent $\Delta \vdash \Sigma \triangleright (\nu\gamma)C$, channel γ does not occur in either Δ or Σ, in any form, γ^+ or γ^-. Relatedly, when we say that α 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, again in any c, γ^+, γ^- form.

We start with a basic result used in the proofs of both Subject Reduction (on page 31) and Type Safety (on page 37).

Lemma 4 *(1) If* $\Gamma; \Sigma; C \mapsto \Sigma'$ *then* $\Sigma' \subseteq \Sigma$.
(2) If $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ *then* $\Sigma_1 \subseteq \Sigma$ *and* $\mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) = \emptyset$.

PROOF. The proof of the first clause is by induction on the derivation of the judgement, using the result for expressions (clause 2). The proof of the second clause is by a straightforward induction on the derivation of the judgement.

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 13. [4]

Lemma 5 (Variable Weakening) *Suppose that* x *does not occur in* C, e, v.

(1) If $\Gamma; \Sigma; C \mapsto \Sigma'$, *then* $\Gamma, x\colon T; \Sigma; C \mapsto \Sigma'$.
(2) If $\Gamma; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2$, *then* $\Gamma, x\colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2$.
(3) If $\Gamma; v \mapsto U$, *then* $\Gamma, x\colon T; v \mapsto U$.

PROOF. The proof of the first clause is by induction on the derivation of the judgement, using the result for expressions (clause 2). The proofs of the second and third clauses, by mutual induction on the derivation of the judgements, are straightforward.

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

(1) If $\Gamma, x\colon T; \Sigma; C \mapsto \Sigma'$, *then* $\Gamma; \Sigma; C \mapsto \Sigma'$.
(2) If $\Gamma, x\colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2$, *then* $\Gamma; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2$.

[4] In the formulation of the lemma, we have omitted the hypothesis that x is not in the domain of Γ (for otherwise $\Gamma, x\colon T$ would not be defined in the conclusion). We henceforth follow this convention for all sorts of environments.

(3) If $\Gamma, x\colon T; v \mapsto U$*, then* $\Gamma; v \mapsto U$*.*

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

We repeat the above exercise, this time for channel enviroments rather than variables.

Lemma 7 (Channel Environment Narrowing)

(1) If $\Gamma; \Sigma, \Sigma'; C \mapsto \Sigma'', \Sigma'$ *then* $\Gamma; \Sigma; C \mapsto \Sigma''$*.*
(2) If $\Gamma; \Sigma, \Sigma'; e \mapsto \Sigma_1, \Sigma'; T; \Sigma_2$ *then* $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$*.*

PROOF. The proof of the first clause is by induction on the derivation of the judgement, using the result for expressions (clause 2). The proof of the second clause is by a straightforward induction on the derivation of the judgement.

Lemma 8 (Channel Environment Weakening)

(1) If $\Gamma; \Sigma; C \mapsto \Sigma'$ *then* $\Gamma; \Sigma, \Sigma''; C \mapsto \Sigma', \Sigma''$*.*
(2) If $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ *then* $\Gamma; \Sigma, \Sigma'; e \mapsto \Sigma_1, \Sigma'; T; \Sigma_2$*.*

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

The following two results allow to conclude that channels in the domain of a channel environment occur free in a configuration. It is needed in Subject Congruence (channel extrusion using rule C-NewB).

Lemma 9 (Free channels in expressions)

(1) If $\Gamma; \Sigma; e \mapsto \Sigma_1; \mathsf{Chan}\ \gamma^p; \Sigma_2$ *and* Γ *does not contain* $\mathsf{Chan}\ \gamma^p$ *then* γ^p *occurs free in* e*.*
(2) If $\Gamma; \Sigma, \gamma^p\colon S; e \mapsto \Sigma_1; T; \Sigma_2$ *and* Γ *does not contain* $\mathsf{Chan}\ \gamma^p$ *and* $\gamma^p\colon S \notin \Sigma_1$ *then* γ^p *occurs free in* e*.*

PROOF. Straightforward induction on the derivation of the judgements.

Lemma 10 (Free channels in configurations)
If $\Gamma; \Sigma, \gamma^p : S; C \mapsto \Sigma'$ *and* Γ *does not contain* $\mathsf{Chan}\ \gamma^p$ *and* $\gamma^p : S \notin \Sigma'$ *then* γ^p *occurs free in* C*.*

PROOF. Straightforward induction on the derivation of the judgement, using Lemma 9 for the case of C-Thread.

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

Lemma 11 (Subject Congruence) *If* $\Gamma; \Sigma; C \mapsto \Sigma'$ *and* $C \equiv C'$*, then* $\Gamma; \Sigma; C' \mapsto \Sigma'$*.*

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.

Commutative monoid rules. There are three cases to consider.

- $C' = C \mid \langle \mathsf{unit} \rangle$. From $\Gamma; \Sigma; C \mapsto \Sigma'$ we can build the derivation

$$\dfrac{\Gamma; \Sigma; C \mapsto \Sigma' \qquad \dfrac{\dfrac{}{\Gamma; \Sigma'; \mathsf{unit} \mapsto \Sigma'; \mathsf{Unit}; \emptyset}\ \text{C-Const}}{\Gamma; \Sigma'; \langle \mathsf{unit} \rangle \mapsto \Sigma'}\ \text{C-Thread}}{\Gamma; \Sigma; C \mid \langle \mathsf{unit} \rangle \mapsto \Sigma'}\ \text{C-Par}$$

  The case $C = C' \mid \langle \mathsf{unit} \rangle$ is direct from the above derivation tree.
- $C = C_1 \mid C_2$, and $C' = C_2 \mid C_1$. Assume the derivation

$$\dfrac{\Gamma; \Sigma; C_1 \mapsto \Sigma_1 \qquad \Gamma; \Sigma_1; C_2 \mapsto \Sigma'}{\Gamma; \Sigma; C_1 \mid C_2 \mapsto \Sigma'}\ \text{C-Par}$$

  From $\Gamma; \Sigma; C_1 \mapsto \Sigma_1$, Lemma 4 gives $\Sigma_1 \subseteq \Sigma$, so that $\Sigma = \Sigma_1, \Sigma_1'$ for some Σ_1'; Lemma 7 gives $\Gamma; \Sigma_1'; C_1 \mapsto \emptyset$; Lemma 8 gives $\Gamma; \Sigma', \Sigma_1'; C_1 \mapsto \Sigma'$. From $\Gamma; \Sigma_1; C_2 \mapsto \Sigma'$, Lemma 8 gives $\Gamma; \Sigma_1, \Sigma_1'; C_2 \mapsto \Sigma', \Sigma_1'$, i.e. $\Gamma; \Sigma; C_2 \mapsto \Sigma', \Sigma_1'$. Finally rule C-Par gives $\Gamma; \Sigma; C_2 \mid C_1 \mapsto \Sigma'$.
- $C = (C_1 \mid C_2) \mid C_3$, and $C' = C_1 \mid (C_2 \mid C_3)$. The assumptions in a derivation of $\Gamma; \Sigma; C \mapsto \Sigma'$ are easily rearranged to give a derivation of $\Gamma; \Sigma; C' \mapsto \Sigma'$, and vice-versa.

For the scope extrusion rules (S-ScopeN, S-ScopeC, and S-ScopeCN) we must consider each rule in both directions; for S-ScopeC we must further consider two cases, depending on whether the typing derivation uses C-NewB or C-NewC.

S-ScopeN. When reading the rule left-to-right we use variable weakening (Lemma 5). In the other direction we use variable narrowing (Lemma 6). In both cases, we use the hypothesis (in the congruence rule) that x is not free in C_2.

S-ScopeC, left-to-right, C-NewB. By hypothesis, we have

$$\dfrac{\dfrac{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mapsto \Sigma_1 \qquad \gamma \text{ not in } \Gamma,\Sigma,\Sigma_1}{\Gamma;\Sigma;(\nu\gamma)C_1 \mapsto \Sigma_1}\,\text{C-NewB} \qquad \Gamma;\Sigma_1;C_2 \mapsto \Sigma'}{\Gamma;\Sigma;(\nu\gamma)C_1 \mid C_2 \mapsto \Sigma'}\,\text{C-Par}$$

From the assumptions in the above tree, we build the following derivation. Because γ is not in Σ_1, Lemma 4 guarantees that it is not in Σ'.

$$\dfrac{\dfrac{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mapsto \Sigma_1 \qquad \Gamma;\Sigma_1;C_2 \mapsto \Sigma'}{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mid C_2 \mapsto \Sigma'}\,\text{C-Par} \qquad \gamma \text{ not in } \Gamma,\Sigma,\Sigma'}{\Gamma;\Sigma;(\nu\gamma)(C_1 \mid C_2) \mapsto \Sigma'}\,\text{C-NewB}$$

S-ScopeC, left-to-right, C-NewC. Similar to the previous case.

S-ScopeC, right-to-left, C-NewB. By hypothesis, we have a proof tree of the form:

$$\dfrac{\dfrac{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mapsto \Sigma_1 \qquad \Gamma;\Sigma_1;C_2 \mapsto \Sigma'}{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mid C_2 \mapsto \Sigma'}\,\text{C-Par} \qquad \gamma \text{ not in } \Gamma,\Sigma,\Sigma'}{\Gamma;\Sigma;(\nu\gamma)(C_1 \mid C_2) \mapsto \Sigma'}\,\text{C-NewB}$$

We analyse the possibilities for whether or not γ^+ and γ^- are in Σ_1. There are three cases.

(1) $\gamma^+, \gamma^- \notin \Sigma_1$.
(2) $\gamma^+, \gamma^- \in \Sigma_1$.
(3) $\gamma^+ \in \Sigma_1$, $\gamma^- \notin \Sigma_1$ (or symmetrically with the polarities exchanged).

In case 1 we build the following derivation.

$$\dfrac{\dfrac{\Gamma;\Sigma,\gamma^+\!:S,\gamma^-\!:\overline{S};C_1 \mapsto \Sigma_1 \qquad \gamma \text{ not in } \Gamma,\Sigma,\Sigma_1}{\Gamma;\Sigma;(\nu\gamma)C_1 \mapsto \Sigma_1}\,\text{C-NewB} \qquad \Gamma;\Sigma_1;C_2 \mapsto \Sigma'}{\Gamma;\Sigma;(\nu\gamma)C_1 \mid C_2 \mapsto \Sigma'}\,\text{C-Par}$$

In case 2 we have $\Sigma_1 = \Sigma_1', \gamma^+ \colon S, \gamma^- \colon \overline{S}$ and $\Gamma; \Sigma_1', \gamma^+ \colon S, \gamma^- \colon \overline{S}; C_2 \mapsto \Sigma'$. Because γ is not in Σ', Lemma 10 implies that γ^+ and γ^- occur free in C_2, contradicting the conditions of S-ScopeC. Therefore this case cannot arise.

In case 3, a similar analysis to case 2, with γ^+ only, shows that the case cannot arise.

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

S-ScopeCN. Straightforward.

The following result allows to replace a given channel with another one, throughout a derivation tree. We use it in Subject Reduction, for rule R-Init, to unify the two fresh channels in the hypothesis, and for rule R-Com, to unify the sent channel with its name in the receiver.

Lemma 12 (Channel replacement) *Suppose that γ^p does not occur in any of $\Gamma, \Sigma, \Sigma_1, \Sigma_2, T, e, v$, and that c does not occur in Γ.*

(1) If $\Gamma, x \colon \mathsf{Chan}\ c; \Sigma, c \colon S; e \mapsto \Sigma_1; T; \Sigma_2$ then
$\Gamma, x \colon \mathsf{Chan}\ \gamma^p; \Sigma, \gamma^p \colon S; e\{\gamma^p/c\} \mapsto \Sigma_1\{\gamma^p/c\}; T\{\gamma^p/c\}; \Sigma_2\{\gamma^p/c\}$.
(2) If $\Gamma, x \colon \mathsf{Chan}\ c; v \mapsto T$, then $\Gamma, x \colon \mathsf{Chan}\ \gamma^p; v\{\gamma^p/c\} \mapsto T\{\gamma^p/c\}$.

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 13 (Substitution) *Suppose that $\Gamma; v \mapsto T$.*

(1) If $\Gamma, x \colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2$ then $\Gamma; \Sigma; e\{v/x\} \mapsto \Sigma_1; U; \Sigma_2$.
(2) If $\Gamma, x \colon T; u \mapsto U$ then $\Gamma; u\{v/x\} \mapsto U$.

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 C-Const, C-Chan, and follow easily, observing that x does not occur in u, and applying Lemma 6. The case of rule C-

VAR follows trivially, as $u = x$. The case of rule C-ABS uses the result for expressions, and that of rule C-REC follows by induction.

We are finally in a position to prove Subject Reduction.

PROOF. [Theorem 2, page 18] The proof proceeds by induction on the derivation of $C \to C'$. We analyse each reduction rule in Figure 3, page 13, in turn.

R-Init. By hypothesis, we have

$$\langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2 \rangle \to (\nu\gamma)(\langle \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2 \rangle)$$

where γ is not free in t_1, t_2, and

$$\Gamma \vdash \Sigma \triangleright \langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2 \rangle.$$

The only proof tree for this sequent is of the form

$$\frac{(1) \qquad (2)}{\Gamma; \Sigma; \langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2 \rangle \mapsto \Sigma_0}\ \text{C-PAR}$$

where (1) is the tree

$$\frac{\dfrac{\Gamma; n \mapsto [S] \qquad d_1 \text{ fresh}}{\Gamma; \Sigma; \mathsf{request}\ n \mapsto \Sigma; \mathsf{Chan}\ d_1; d_1 \colon \overline{S}} \quad \Gamma, x \colon \mathsf{Chan}\ d_1; \Sigma, d_1 \colon \overline{S}; t_1 \mapsto \Sigma_2; T_1; \Sigma_2'}{\dfrac{\Gamma; \Sigma; \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1 \mapsto \Sigma_1; T_1; \emptyset}{\Gamma; \Sigma; \langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1 \rangle \mapsto \Sigma_1}\ \text{C-THREAD}}$$

and (2) is the tree

$$\frac{\dfrac{\Gamma; n \mapsto [S] \qquad d_2 \text{ fresh}}{\Gamma; \Sigma_1; \mathsf{accept}\ n \mapsto \Sigma_1; \mathsf{Chan}\ d_2; d_2 \colon S} \quad \Gamma, y \colon \mathsf{Chan}\ d_2; \Sigma_1, d_2 \colon S; t_2 \mapsto \Sigma_3; T_2; \Sigma_3'}{\dfrac{\Gamma; \Sigma_1; \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2 \mapsto \Sigma_0; T_2; \emptyset}{\Gamma; \Sigma_1; \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2 \rangle \mapsto \Sigma_0}\ \text{C-THREAD}}$$

By examining the environments in the instances of C-LET in the above deriva-

tions, we obtain

$$\begin{aligned} \Sigma_1 &= \Sigma \cap \Sigma_2 \\ (d_1 \colon \overline{S} \cap \Sigma_2), \Sigma_2' &= \emptyset \\ \Sigma_0 &= \Sigma_1 \cap \Sigma_3 \\ (d_2 \colon S \cap \Sigma_3), \Sigma_3' &= \emptyset \end{aligned}$$

and hence $\Sigma_2' = \emptyset$, $\Sigma_3' = \emptyset$, $d_1 \notin \Sigma_2$ and $d_2 \notin \Sigma_3$.

From the hypothesis $\Gamma, x \colon \mathsf{Chan}\ d_1; \Sigma, d_1 \colon \overline{S}; t_1 \mapsto \Sigma_2; T_1; \Sigma_2'$ in (1) we use Lemma 12 to replace d_1 by γ^+, and then weaken (Lemma 8) with γ^- to obtain

$$\Gamma, x \colon \mathsf{Chan}\ \gamma^+; \Sigma, \gamma^+ \colon \overline{S}, \gamma^- \colon S; t_1 \mapsto \Sigma_2, \gamma^- \colon S; T_1; \emptyset$$

and so we can construct the derivation (1*) as follows, where we let $\Sigma' = \Sigma, \gamma^+ \colon \overline{S}, \gamma^- \colon S$, and $\Sigma_1' = \Sigma_1, \gamma^- \colon S$:

$$\dfrac{\dfrac{\text{C-Chan, C-Val}}{\Gamma; \Sigma'; \gamma^+ \mapsto \Sigma'; \mathsf{Chan}\ \gamma^+; \emptyset} \qquad \Gamma, x \colon \mathsf{Chan}\ \gamma^+; \Sigma'; t_1 \mapsto \Sigma_2, \gamma^- \colon S; T_1; \emptyset}{\dfrac{\Gamma; \Sigma'; \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1 \mapsto \Sigma_1'; T_1; \emptyset}{\Gamma; \Sigma'; \langle \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1 \rangle \mapsto \Sigma_1'}\ \text{C-Thread}}$$

From the hypothesis $\Gamma, y \colon \mathsf{Chan}\ d_2; \Sigma_1, d_2 \colon \overline{S}; t_2 \mapsto \Sigma_3; T_2; \Sigma_3'$ in (1) we use Lemma 12 to replace d_2 by γ^- to obtain

$$\Gamma, y \colon \mathsf{Chan}\ \gamma^-; \Sigma_1, \gamma^- \colon S; t_2 \mapsto \Sigma_3; T_2; \emptyset$$

and so we can construct the derivation (2*) as follows:

$$\dfrac{\dfrac{\text{C-Chan, C-Val}}{\Gamma; \Sigma_1'; \gamma^- \mapsto \Sigma_1'; \mathsf{Chan}\ \gamma^-; \emptyset} \qquad \Gamma, y \colon \mathsf{Chan}\ \gamma^-; \Sigma, \gamma^- \colon S; t_2 \mapsto \Sigma_3; T_2; \emptyset}{\dfrac{\Gamma; \Sigma_1'; \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2 \mapsto \Sigma_0; T_2; \emptyset}{\Gamma; \Sigma_1'; \langle \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2 \rangle \mapsto \Sigma_0}\ \text{C-Thread}}$$

The following derivation completes this case:

$$\dfrac{\dfrac{(1^*) \qquad (2^*)}{\Gamma; \Sigma, \gamma^+ \colon \overline{S}, \gamma^- \colon S; \langle \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2 \rangle \mapsto \Sigma_0}\ \text{C-Par}}{\Gamma; \Sigma; (\nu\gamma)(\langle \mathsf{let}\ x = \gamma^+\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \gamma^-\ \mathsf{in}\ t_2 \rangle) \mapsto \Sigma_0}\ \text{C-NewB}$$

R-Com, sending a channel. By hypothesis, we have

$$\langle \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{send}\ \beta\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \rightarrow \langle \mathsf{let}\ x = \beta\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle$$

The typing derivation is as follows:

$$\frac{(1) \qquad (2)}{\Gamma; \Sigma; \langle \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{send}\ \beta\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \mapsto \Sigma_0}\ \text{C-Par}$$

where (1) is the tree

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash \gamma^p \colon \mathsf{Chan}\ \gamma^p}\ \text{C-Chan}}{\Gamma; \Sigma; \mathsf{receive}\ \gamma^p \mapsto \Sigma_2; \mathsf{Chan}\ d; \Sigma_3} \qquad \Gamma, x \colon \mathsf{Chan}\ d; \Sigma_2, \Sigma_3; t_1 \mapsto \Sigma_4; T_1; \Sigma_5}{\dfrac{\Gamma; \Sigma; \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t_1 \mapsto \Sigma_1; T_1; \emptyset}{\Gamma; \Sigma; \langle \mathsf{let}\ x = \mathsf{receive}\ \gamma^p\ \mathsf{in}\ t_1\rangle \mapsto \Sigma_1}\ \text{C-Thread}}$$

and (2) is the tree

$$\frac{\dfrac{\Gamma; \beta \mapsto \mathsf{Chan}\ \beta \qquad \Gamma; \gamma^{\overline{p}} \mapsto \mathsf{Chan}\ \gamma^{\overline{p}}}{\Gamma; \Sigma_1; \mathsf{send}\ \beta\ \mathsf{on}\ \gamma^{\overline{p}} \mapsto \Sigma_6; \mathsf{Unit}; \Sigma_7} \quad \Gamma, y \colon \mathsf{Unit}; \Sigma_6, \Sigma_7; t_2 \mapsto \Sigma_8; T_2; \Sigma_9}{\dfrac{\Gamma; \Sigma_1; \mathsf{let}\ y = \mathsf{send}\ \beta\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2 \mapsto \Sigma_0; T_2; \emptyset}{\Gamma; \Sigma_1; \langle \mathsf{let}\ y = \mathsf{send}\ \beta\ \mathsf{on}\ \gamma^{\overline{p}}\ \mathsf{in}\ t_2\rangle \mapsto \Sigma_0}\ \text{C-Thread}}$$

By analysing the rules used in (1) and (2), we can deduce more about the structure of the channel environments.

C-ReceiveS in (1) implies

$$\begin{aligned} \Sigma &= \Sigma', \gamma^p \colon ?S.S_1 \\ \Sigma_2 &= \Sigma' \\ \Sigma_3 &= \gamma^p \colon S_1, d \colon S \end{aligned}$$

C-Let in (1) implies

$$\begin{aligned} \Sigma_1 &= \Sigma_2 \cap \Sigma_4 \\ (\Sigma_3 \cap \Sigma_4), \Sigma_5 &= \emptyset \end{aligned}$$

from which we conclude that $\Sigma_5 = \emptyset$ and $\Sigma_3 \cap \Sigma_4 = \emptyset$, and hence $\gamma^p \notin \Sigma_4$ and $d \notin \Sigma_4$. By Lemma 4, $\Sigma_4 \subseteq (\Sigma_2, \Sigma_3)$, so $\Sigma_4 \subseteq \Sigma_2$ and $\Sigma_1 = \Sigma_4$.

C-SendS in (2), together with the fact that Σ is balanced, implies

$$\begin{aligned}\Sigma_1 &= \Sigma_1', \gamma^{\overline{p}} \colon !S.\overline{S_1}, \beta \colon S \\ \Sigma_6 &= \Sigma_1' \\ \Sigma_7 &= \gamma^{\overline{p}} \colon \overline{S_1}\end{aligned}$$

C-Let in (2) implies

$$\begin{aligned}\Sigma_0 &= \Sigma_6 \cap \Sigma_8 \\ (\Sigma_7 \cap \Sigma_8), \Sigma_9 &= \emptyset\end{aligned}$$

from which we conclude that $\Sigma_9 = \emptyset$ and $\Sigma_7 \cap \Sigma_8 = \emptyset$, and hence $\gamma^{\overline{p}} \notin \Sigma_8$. By Lemma 4, $\Sigma_8 \subseteq (\Sigma_6, \Sigma_7)$, so $\Sigma_8 \subseteq \Sigma_6$ and $\Sigma_0 = \Sigma_8$.

Since d is fresh, it is not in t_1. Hence we can construct the derivation (1*), with the following instance of C-Let, where we let $\Sigma_1'' = \Sigma_1', \gamma^{\overline{p}} \colon \overline{S_1}$:

$$\dfrac{\dfrac{\text{C-Chan, C-Val}}{\Gamma; \Sigma_1; \beta \mapsto \Sigma_1; \mathsf{Chan}\ \beta; \emptyset} \qquad \Gamma, x \colon \mathsf{Chan}\ \beta; \Sigma_1; t_1 \mapsto \Sigma_1''; T_1\{\beta/d\}; \emptyset}{\Gamma; \Sigma_1; \mathsf{let}\ x = \beta\ \mathsf{in}\ t_1 \mapsto \Sigma_1''; T_1\{\beta/d\}; \emptyset}$$

Also we can construct the derivation (2*) with the following instance of C-Let:

$$\dfrac{\dfrac{\text{C-Const, C-Val}}{\Gamma; \Sigma_1''; \mathsf{unit} \mapsto \Sigma_1''; \mathsf{Unit}; \emptyset} \qquad \Gamma, y \colon \mathsf{Unit}; \Sigma_1''; t_2 \mapsto \Sigma_0; T_2; \emptyset}{\Gamma; \Sigma_1''; \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2 \mapsto \Sigma_0; T_2; \emptyset}$$

From (1*) and (2*) we can construct a derivation of

$$\Gamma; \Sigma_1; \langle \mathsf{let}\ x = \beta\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2 \rangle \mapsto \Sigma_0$$

as required.

R-Com, sending a non-channel. Follows a pattern similar to, but simpler than, the one above.

R-Close. Similar to R-Com (non-channel).

R-New. By hypothesis, we have

$$\langle \mathsf{let}\ x = \mathsf{new}\ S\ \mathsf{in}\ t \rangle \to (\nu n \colon [S]) \langle \mathsf{let}\ x = n\ \mathsf{in}\ t \rangle$$

and

$$\dfrac{\dfrac{}{\Gamma;\Sigma;\mathsf{new}\ S \mapsto \Sigma;[S];\emptyset}\ \text{C-New} \qquad \Gamma, x\colon [S];\Sigma;t \mapsto \Sigma';T;\emptyset}{\dfrac{\Gamma;\Sigma;\mathsf{let}\ x = \mathsf{new}\ S\ \mathsf{in}\ t \mapsto \Sigma';T;\emptyset}{\Gamma;\Sigma;\langle \mathsf{let}\ x = \mathsf{new}\ S\ \mathsf{in}\ t\rangle \mapsto \Sigma'}\ \text{C-Thread}}\ \text{C-Let}$$

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 5 is applicable to the premise of rule C-Let above, and hence,

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{C-Var, C-Val}}{\Gamma, n\colon [S];\Sigma;n \mapsto \Sigma;[S];\emptyset} \qquad \dfrac{\Gamma, x\colon [S];\Sigma;t \mapsto \Sigma';T;\emptyset}{\Gamma, n\colon [S], x\colon [S];\Sigma;t \mapsto \Sigma';T;\emptyset}\ \text{Lemma 5}}{\Gamma, n\colon [S];\Sigma;\mathsf{let}\ x = n\ \mathsf{in}\ t \mapsto \Sigma';T;\emptyset}\ \text{C-Let}}{\Gamma, n\colon [S];\Sigma;\langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle \mapsto \Sigma'}\ \text{C-Thread}}{\Gamma;\Sigma;(\nu n\colon [S])\langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle \mapsto \Sigma'}\ \text{C-NewN}$$

R-Fork. By hypothesis, we have

$$\langle \mathsf{fork}\ t_1; t_2\rangle \to \langle t_1\rangle \mid \langle t_2\rangle$$

and

$$\dfrac{\dfrac{\Gamma;\Sigma;t \mapsto \Sigma_1;T_1;\emptyset \qquad \Gamma;\Sigma_1;t' \mapsto \Sigma';T_2;\emptyset}{\Gamma;\Sigma;\mathsf{fork}\ t;t' \mapsto \Sigma';T_2;\emptyset}\ \text{C-Fork}}{\Gamma;\Sigma;\langle \mathsf{fork}\ t;t'\rangle \mapsto \Sigma'}\ \text{C-Thread}$$

¿From the hypotheses in the above tree, we build a tree to complete the proof.

$$\dfrac{\dfrac{\dfrac{\Gamma;\Sigma;t \mapsto \Sigma_1;T_1;\emptyset}{\Gamma;\Sigma;\langle t\rangle \mapsto \Sigma_1}\ \text{C-Thread} \qquad \dfrac{\Gamma;\Sigma_1;t' \mapsto \Sigma';T_2;\emptyset}{\Gamma;\Sigma_1;\langle t'\rangle \mapsto \Sigma'}\ \text{C-Thread}}{\Gamma;\Sigma;\langle t\rangle \mid \langle t'\rangle \mapsto \Sigma'}\ \text{C-Par}$$

R-App. By hypothesis, we have

$$\langle \mathsf{let}\ x = (\lambda(\Sigma; y\colon T).e)v\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t\rangle$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, y\colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2}{\Gamma; \lambda(\Sigma; y\colon T).e \mapsto (\Sigma; T \to U; \Sigma_1, \Sigma_2)}\,\text{C-Abs} \quad \Gamma; v \mapsto T}{\Gamma; \Sigma, \Sigma'; (\lambda(\Sigma; y\colon T).e)v \mapsto \Sigma'; U; \Sigma_1, \Sigma_2}\,\text{C-App} \quad (1)}{\Gamma; \Sigma, \Sigma'; \mathsf{let}\ x = (\lambda(\Sigma; y\colon T).e)v\ \mathsf{in}\ t \mapsto \Sigma' \cap \Sigma_3; T'; \emptyset}\,\text{C-Let}}{\Gamma; \Sigma, \Sigma'; \langle \mathsf{let}\ x = (\lambda(\Sigma; y\colon T).e)v\ \mathsf{in}\ t\rangle \mapsto \Sigma' \cap \Sigma_3}\,\text{C-Thread}$$

where (1) is the sequent $\Gamma, x\colon U; \Sigma', \Sigma_1, \Sigma_2; t \mapsto \Sigma_3; T'; \emptyset$. Note that $(\Sigma_1, \Sigma_2) \cap \Sigma_3$ must be $\emptyset$, and so $\Sigma_1 \cap \Sigma_3 = \emptyset$. Then, one may build the following derivation to complete the proof.

$$\dfrac{\dfrac{\dfrac{\Gamma; v \mapsto T \quad \dfrac{\Gamma, y\colon T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2}{\Gamma, y\colon T; \Sigma, \Sigma'; e \mapsto \Sigma', \Sigma_1; U; \Sigma_2}\,\text{Lemma 8}}{\Gamma; \Sigma, \Sigma'; e\{v/y\} \mapsto \Sigma', \Sigma_1; U; \Sigma_2}\,\text{Lemma 13} \quad (1)}{\Gamma; \Sigma, \Sigma'; \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t \mapsto (\Sigma', \Sigma_1) \cap \Sigma_3; T'; \emptyset}\,\text{C-Let}}{\Delta; \Sigma, \Sigma'; \langle \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t\rangle \mapsto \Sigma' \cap \Sigma_3}\,\text{C-Thread}$$

R-Rec. By hypothesis, we have

$$\langle \mathsf{let}\ x = (\mathsf{rec}\ (y\colon T).v)u\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ x = (v\{\mathsf{rec}\ (y\colon T).v/y\})u\ \mathsf{in}\ t\rangle$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, y\colon T; v \mapsto T}{\Gamma; \mathsf{rec}\ (y\colon T).v \mapsto T}\,\text{C-Rec} \quad \Gamma; u \mapsto T'}{\Gamma; \Sigma, \Sigma''; (\mathsf{rec}\ (y\colon T).v)u \mapsto \Sigma''; U; \Sigma'}\,\text{C-App} \quad (1)}{\Gamma; \Sigma, \Sigma''; \mathsf{let}\ x = (\mathsf{rec}\ (y\colon T).v)u\ \mathsf{in}\ t \mapsto \Sigma'' \cap \Sigma_1; T; \emptyset}\,\text{C-Let}}{\Gamma; \Sigma, \Sigma''; \langle \mathsf{let}\ x = (\mathsf{rec}\ (y\colon T).v)u\ \mathsf{in}\ t\rangle \mapsto \Sigma'' \cap \Sigma_1}\,\text{C-Thread}$$

where (1) is $\Gamma, x\colon U; \Sigma'', \Sigma'; t \mapsto \Sigma_1; T; \emptyset$. Then, one may build the following derivation to complete the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, y\colon T; v \mapsto T}{\Gamma; \mathsf{rec}\ (y\colon T).v \mapsto T}\,\text{C-Rec}}{\Gamma; v\{\mathsf{rec}\ (y\colon T).v/y\} \mapsto T}\,\text{Lemma 13} \quad \Gamma; u \mapsto T'}{\Gamma; \Sigma, \Sigma''; (v\{\mathsf{rec}\ (y\colon T).v/y\})u \mapsto \Sigma''; U; \Sigma'}\,\text{C-App} \quad (1)}{\Gamma; \Sigma, \Sigma''; \mathsf{let}\ x = (v\{\mathsf{rec}\ (y\colon T).v/y\})u\ \mathsf{in}\ t \mapsto , \Sigma'' \cap \Sigma_1; T; \emptyset}\,\text{C-Let}}{\Gamma; \Sigma, \Sigma''; \langle \mathsf{let}\ x = (v\{\mathsf{rec}\ (y\colon T).v/y\})u\ \mathsf{in}\ t\rangle \mapsto , \Sigma'' \cap \Sigma_1}\,\text{C-Thread}$$

R-Beta. By hypothesis, we have

$$\langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle \to \langle t\{v/x\}\rangle.$$

and

$$\dfrac{\dfrac{\dfrac{\Gamma; v \mapsto T_1}{\Gamma; \Sigma; v \mapsto \Sigma; T_1; \emptyset}\ \text{C-Val} \qquad \Gamma, x\colon T_1; \Sigma; t \mapsto \Sigma_1; T_2; \emptyset}{\Gamma; \Sigma; \mathsf{let}\ x = v\ \mathsf{in}\ t \mapsto \Sigma_1; T_2; \emptyset}\ \text{C-Let}}{\Gamma; \Sigma; \langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle \mapsto \Sigma_1}\ \text{C-Thread}$$

¿From the hypotheses of this derivation, Lemma 13 gives $\Gamma; \Sigma; t\{v/x\} \mapsto \Sigma_1; T_2; \emptyset$ and C-Thread gives $\Gamma; \Sigma; \langle t\{v/x\}\rangle \mapsto \Sigma_1$ as required.

R-Let. A straightforward rearrangement of the derivation tree.

R-Branch. Follows the pattern in all the above cases.

R-NewC, R-NewN, R-Par, R-Cong. These cases follow directly by induction. For R-Cong, we use Lemma 11.

B Proof of Theorem 3, Type Safety

We start with three easy results.

Lemma 14 *Suppose that* $\Delta; \Sigma; C \mapsto \Sigma'$. *Then,*

(1) Δ *is of the form* $\vec{x}\colon [\vec{S}]$;
(2) If C *is a* γ*-redex, then* $\gamma^+\colon S$ *and* $\gamma^-\colon \overline{S}$ *are in* Σ *but not in* Σ' .
(3) If C *is a* γ*-thread, then for some* p, γ^p *is in the domain of* Σ;

PROOF. 1. Simple induction on the derivation tree of the sequent.
2. A simple analysis of the possible derivation trees for the three possible γ-redex cases.
3. A simple analysis of the conclusions of the last rule applied in the derivation of the sequent for γ-threads, namely C-SendD, C-SendS, C-ReceiveD, C-ReceiveS, C-Case, C-Select, and C-Close.

Then, for the main result of this section.

PROOF. [Theorem 3, page 19] By contradiction, assuming faulty configurations typable and performing a case analysis on the possible forms of the faulty configurations.

Assume that $C \equiv (\nu\vec{x}\colon \vec{T})(\nu\vec{\gamma})(C_1 \mid C_2)$, and that $\Delta;\Sigma;C \mapsto \Sigma'$ with Σ balanced. Build the only possible proof tree for the above sequent, first using rule C-NewN as many times as there are variables in $\vec{x}$, then proceeding similarly with rules C-NewB or C-NewC as many times as there are channels in $\vec{\gamma}$, and finally with rule C-Par, to obtain a subtree ending with the following sequent.

$$\Delta';\Sigma';C_1 \mapsto \_ \tag{B.1}$$

where $\Delta' = \Delta, \vec{x}\colon \vec{T}$, and Σ' is $\Sigma, \gamma_1^+ : S_1, \gamma_1^- : \overline{S_1}, \ldots, \gamma_r^+ : S_r, \gamma_r^- : \overline{S_r}$, and channels $\gamma_1, \ldots \gamma_r$ are part of $\vec{\gamma}$, introduced by rule C-NewB. Notice that Δ is of the form $\vec{x}\colon [\vec{S}]$, by Lemma 14. Also notice that Σ' is balanced, since Σ is balanced by hypothesis.

We now analyse each of the five possible classes of faulty configurations defined in Section 6.

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

$$\dfrac{\dfrac{\dfrac{\Delta';v \mapsto (\Sigma_1;T_1 \to T_2;\Sigma_2) \qquad \cdots}{\Delta';\Sigma';v\_ \mapsto \_;\_;\_}\,\text{C-App} \qquad \cdots}{\Delta';\Sigma';\mathsf{let}\ x = v\_\ \mathsf{in}\ t \mapsto \_;\_;\_}\,\text{C-Let}}{\Delta';\Sigma';\langle \mathsf{let}\ x = v\_\ \mathsf{in}\ t\rangle \mapsto \_}\,\text{C-Thread}$$

Analysing the rules for values (Figure 5, page 15), one realises that v can only be an abstraction or a recursive expression, for the C-Var does not apply given the form of Δ, and the type in the conclusion of the remaining rules (C-Const, C-Chan) is not a functional type.

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

$$\Delta';\Sigma';\langle \mathsf{let}\ x = \mathsf{accept}\ v\ \mathsf{in}\ t\rangle \mapsto \_$$

to obtain a tree for

$$\Delta';v \mapsto [S].$$

Given the form of the type for v, among the rules for values, only C-Var applies. Hence, v is a variable.

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

$$\Delta';\Sigma';\langle \mathsf{let}\ x = \mathsf{receive}\ v\ \mathsf{in}\ t\rangle \mapsto \_$$

to obtain a tree for

$$\Delta';v \mapsto \mathsf{Chan}\ \gamma^p.$$

Once again, given the form of Δ', among the rules for values, only C-Chan applies. Then v can only be γ^p.

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

$$\Delta'; \Sigma'; \mathsf{select}\ l\ \mathsf{on}\ \gamma^p \mapsto \Sigma''; \_; \_ \qquad\qquad \Delta'; \Sigma''; \mathsf{close}\ \gamma^q \mapsto \_; \_; \_$$

Analysing the rule for select, one finds that $\gamma^p \colon \oplus\langle l \colon S\rangle$ must be in Σ'. Similarly, analysing the rule for close one realises that $\gamma^q \colon \mathsf{End}$ must be in Σ''. From Lemma 4, we know that $\Sigma'' \subseteq \Sigma'$, hence that $\gamma^p \colon \oplus\langle l \colon S\rangle, \gamma^q \colon \mathsf{End}$ is in Σ'. Then it must be the case that $q = \overline{p}$. But Σ' is balanced (and $\oplus\langle l \colon S\rangle$ is not the dual of End), hence $(\nu\vec{x} \colon \vec{T})(\nu\vec{\gamma})(C_1 \mid C_2)$ is not typable.

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

$$\frac{\Delta'; \Sigma'; \langle t_1\rangle \mid \langle t_2\rangle \mapsto \Sigma'' \qquad \Delta'; \Sigma''; \langle t_3\rangle \mapsto \_}{\Delta'; \Sigma'; (\langle t_1\rangle \mid \langle t_2\rangle) \mid \langle t_3\rangle \mapsto \_}\ \text{C-Par}$$

If $\langle t_1\rangle \mid \langle t_2\rangle$ is not a γ-redex, then we use case **4**. Otherwise, by Lemma 14(2), it must be the case that $\gamma^+ \colon S$ and $\gamma^- \colon \overline{S}$ are in Σ' but not in Σ''. With Lemma 14(3), this contradicts the assumption that $\langle t_3\rangle$ is a γ-thread.