Session Types for Functional Multithreading*

Vasco Vasconcelos† António Ravara‡ Simon Gay§

May 20, 2005

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, our channel types statically track aliasing, and our function types not only specify argument and result types but also describe changes in channels. We formalize the syntax, semantics and typing rules of our language, and prove subject reduction and runtime type safety theorems.
Keywords: Session types, static typechecking, concurrent programming, specification of communication protocols.

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.

*A revised version of the paper in Concur 2004, volume 3170 of LNCS, pages 497–511, Springer-Verlag, 2004; includes proofs for the results. A forthcoming paper in Theoretical Computer Science describes a type inference algorithm for an explicitly typed version of the language.

†Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal. Email: <vv@di.fc.ul.pt>

‡CLC and Departamento de Matemática, Instituto Superior Técnico, 1049-001 Lisboa, Portugal. Email: <amar@math.ist.utl.pt>

§Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. Email: <simon@dcs.gla.ac.uk>

The theory of *session types* [9, 13, 14, 24] allows the specification of a protocol to be expressed as a type; when a communication channel is created, a session type is associated with it. 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, it becomes possible to verify, at compile-time, that an agent using the channel does so in accordance with the protocol.

The theory of session types has been developed in the context of the π-calculus [18, 23], an idealized concurrent programming language which focuses on inter-process communication. Session types have not yet been incorporated into a mainstream programmming language, or even studied theoretically in the context of a standard language paradigm: functional, imperative or object-oriented. Vallecillo et al. [25] use session types to add behavioural information to the interfaces of CORBA objects, and use Gay and Hole's [9] theory of subtyping to formalize compatibility and substitutability of components, but they have not attempted to design a complete language.

The Vault [5] and Cyclone [12] languages extend C with facilities for safe control of stateful resources. In Cyclone, locks must be acquired and released; Vault goes further by allowing operations on a resource to be statically checked against an automaton which specifies valid transitions. In contrast, session types are specialized to communication channels as a particular kind of resource, but as a result they enable further typechecking in association with each state transition: typechecking verifies the types of individual messages, as well as verifying that a sequence of messages obeys a given protocol. (These languages are further discussed in Section 7.)

In previous work [8] we have presented a language supporting typed functional programming with inter-process communication channels, but we only considered individual processes in isolation. Here we address collections of functional threads communicating via channels. This formulation allows us to prove a runtime safety property: well-typed programs do not misuse channels.

By transferring the concept of session types from the π-calculus to a multi-threaded functional language with side-effecting input/output operations, we show that static checking of session types could be added to a language such as Concurrent ML [22], at least without imperative features. In particular we have addressed the key differences between a conventional programming style and the programming notation of the π-calculus:

- 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 channel environment.
- We have separated creation and naming of channels, and because this

introduces the possibility of aliasing, we represent the types of channels by indirection from the main type environment to the channel environment.

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, 4 and 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.

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. The type does not correspond precisely to the specification, because it does not state that the server calculates the sum. However, the type captures the parts of the specification which we can reasonably expect to verify statically. The server communicates with a client on a channel called u; we think of the client engaging in a *session* with the server, using the channel u for communication. In our language, the server looks like this:

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

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:

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

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

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, that is, with sequences of the form: select a method; send the arguments; receive the result.

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

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

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 [8].

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*.

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

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]

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

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.

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

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 [24], 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.

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

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.

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

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 $\mathsf{new}\ S$. To distribute it to a second thread, we fork a new thread, in whose code the name occurs.[2] Our complete system creates a name x and launches three threads (a server and two clients), all sharing the newly created name.

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

Given the above implementation of *server*, one of the clients will be forever $\mathsf{requesting}$ x. Fortunately, it is easy to extend the *server* to accept more than one connection in its life time.

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

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.

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

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 $?\mathsf{Int}.\mathsf{End}$; 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.

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

After sending a channel, no further interaction on the channel is possible. Note that *askNeg* cannot $\mathsf{close}\ u$, 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 $\Sigma; T \to U \to V; \Sigma'$.

$$
\begin{aligned}
v &::= c \mid n \mid x \mid \lambda x.e \mid \mathsf{rec}\ x.v \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \\
e &::= t \mid vv \mid \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{new} \mid \mathsf{accept}\ v \mid \mathsf{request}\ v \mid \\
&\quad\ \mathsf{send}\ v\ \mathsf{on}\ v \mid \mathsf{receive}\ v \mid \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I} \mid \mathsf{select}\ l\ \mathsf{on}\ v \mid \mathsf{close}\ v \\
t &::= v \mid \mathsf{let}\ x = e\ \mathsf{in}\ t \mid \mathsf{fork}\ t; t \\
C &::= \langle t \rangle \mid (C \mid C) \mid (\nu n)C \mid (\nu c)C
\end{aligned}
$$

Figure 1: Syntax of values, expressions, threads and configurations

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

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

where S is !Int.!Int.End. Here *sendTwice* must be typed once with channel c, and another with channel d. For an example of session polymorphism, we have:

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

where *sendTwice* must be typed once with c: !Int.!Int.!Int.!Int.End, and a second time with c: !Int.!Int.End.

3 Syntax

Most of the syntax of our language has been illustrated in the previous section; here we define it formally by the grammar in Figure 1 (cf. [17]).

We use *channel identifiers* $c, \ldots$, *name identifiers* $n, \ldots$, *term variables* $x, \ldots$, and *labels* $l, \ldots$, and define *values* v, *expressions* e, *threads* t, and *configurations* C. To simplify some definitions, we use (νa) to stand for either (νc) or (νn).

Channel identifiers and name identifiers are not available in the top-level syntax of threads; they arise only during reduction, in a request/accept synchronization, and in a new operation, respectively, as described in Section 4.

In Section 2 we used several derived constructors. An expression $e; t$ (sometimes implied in our examples by the indentation) is an abbreviation for let $y = e$ in t, provided y does not occur free in t. Idioms like

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

Figure 2: Structural congruence

send (receive c)(receive c) 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 let $x = e$ in x. Recursive function definitions must be made explicit with rec.

4 Operational Semantics

The binding occurrences are x in $\lambda x.e$, rec $x.e$, let $x = e$ in t, n in $(\nu n)C$ and c in $(\nu c)C$. Free and bound identifiers are defined as usual and we work up to α-equivalence. Substitution, of values for variables, is defined as expected. We define a reduction semantics on configurations (Figure 3), making use of a simple structural congruence relation [18] (Figure 2), allowing for the rearrangement of threads in a configuration, so that reduction may happen.[4]

We now explain the reduction rules. R-Init synchronizes two threads on a *shared name* n, creating a new channel c known to both threads. Rules R-Com, R-Branch, and R-Close synchronize two threads on a *channel* c: R-Com transmits a value v from one thread to the other; R-Branch, rather than transmitting a value, chooses one of the branches in the case thread; and R-Close closes a channel in *both* threads simultaneously. R-New creates a new name n, and records the fact that the name is potentially shared, by means of a (νn) 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 programmed by arranging for both threads to share a channel and explicitly sending the result back to the parent.

[4] We could easily arrange for structural congruence to garbage collect all threads of the form $\langle v \rangle$, for v closed.

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

$$\langle \mathsf{let}\ x = \mathsf{receive}\ c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{send}\ v\ \mathsf{on}\ c\ \mathsf{in}\ t_2\rangle \rightarrow \langle \mathsf{let}\ x = v\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle \quad \text{(R-Com)}$$

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

$$\langle \mathsf{let}\ x = \mathsf{close}\ c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{close}\ c\ \mathsf{in}\ t_2\rangle \rightarrow \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle \quad \text{(R-Close)}$$

$$\langle \mathsf{let}\ x = \mathsf{new}\ \mathsf{in}\ t\rangle \rightarrow (\nu n)\langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle \quad \text{(R-New)}$$

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

$$\langle \mathsf{let}\ x = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e\ \mathsf{else}\ e'\ \mathsf{in}\ t\rangle \rightarrow \langle \mathsf{let}\ x = e\ \mathsf{in}\ t\rangle \quad \text{(R-IfT)}$$

$$\langle \mathsf{let}\ x = \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e\ \mathsf{else}\ e'\ \mathsf{in}\ t\rangle \rightarrow \langle \mathsf{let}\ x = e'\ \mathsf{in}\ t\rangle \quad \text{(R-IfF)}$$

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

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

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

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

$$\frac{C \rightarrow C'}{(\nu a)C \rightarrow (\nu a)C'} \qquad \frac{C \rightarrow C'}{C \mid C'' \rightarrow C' \mid C''} \qquad \frac{C \equiv \rightarrow \equiv C'}{C \rightarrow C'} \quad \text{(R-Conf)}$$

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

Figure 3: Reduction rules

5 Typing

The syntax of types is described in Figure 4. We define *session types* S, *channel environments* Σ, *data types* D, and *term types* T. The type $\mathsf{Chan}\ c$ represents the type of the channel with identity c; the session type associated with c is recorded separately in a channel environment Σ. Channel type *bottom*, $\bot$, denotes a channel that is already in use by two threads, hence that cannot be used further. Similarly to channel and name identifiers, $\bot$ is not available at the top level syntax, arising only via the channel environment composition operator, $\Sigma_1 \bullet \Sigma_2$, defined below. Among datatypes we have channel-state annotated functional types $\Sigma; T \rightarrow T; \Sigma$, and types for names $[S]$ capable of establishing sessions of type S.

The type system is presented in Figures 5 to 9. Typing judgements for constants are of the form $\Gamma \vdash v\colon T$, where Γ is a map from names and variables to types. Value judgements do not mention channel environments, for values, having no behaviour, do not change channels. Judgements for

$$
\begin{aligned}
S &::= ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \&\langle l_i\colon S_i\rangle_{i\in I} \mid \oplus\langle l_i\colon S_i\rangle_{i\in I} \mid \mathsf{End} \mid \bot \\
D &::= \mathsf{Bool} \mid \mathsf{Unit} \mid \Sigma; T \to T; \Sigma \mid [S] \\
\Sigma &::= \emptyset \mid \Sigma, c\colon S \quad (c\colon S \text{ not in } \Sigma) \\
T &::= D \mid \mathsf{Chan}\ c
\end{aligned}
$$

Figure 4: Syntax of types

$$
\Gamma \vdash \mathsf{true}\colon \mathsf{Bool} \qquad \Gamma \vdash \mathsf{false}\colon \mathsf{Bool} \qquad \Gamma \vdash \mathsf{unit}\colon \mathsf{Unit} \qquad \text{(T-Const)}
$$

$$
\Gamma \vdash c\colon \mathsf{Chan}\ c \qquad \Gamma, n\colon [S] \vdash n\colon [S] \qquad \Gamma, x\colon T \vdash x\colon T \qquad \text{(T-Chan,T-Name,T-Var)}
$$

$$
\frac{\Gamma, x\colon T \vdash \Sigma \triangleright e\colon U \triangleleft \Sigma'}{\Gamma \vdash \lambda x.e\colon (\Sigma; T \to U; \Sigma')} \qquad \frac{\Gamma, x\colon T \vdash v\colon T \qquad T = (\Sigma; U \to U'; \Sigma')}{\Gamma \vdash \mathsf{rec}\ x.v\colon T} \qquad \text{(T-Abs,T-Rec)}
$$

Figure 5: Typing rules for values

expressions are of the form $\Gamma \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma'$, where Σ is a channel environment (a map from channels into sorts, as in Figure 4). The difference between Σ and Σ' reflects the effect of an expression on the types of channels, for example

$$x\colon \mathsf{Chan}\ c \vdash c\colon ?\mathsf{Int}.\mathsf{End} \triangleright \mathsf{receive}\ x\colon \mathsf{Int} \triangleleft c\colon \mathsf{End}.$$

Finally, typing judgements for configurations are of the form $\Delta \vdash \Sigma \triangleright C$ where Δ is a map from names to datatypes of the form $[S]$.

Typing Values (Figure 5). T-Chan says that a channel named c has type $\mathsf{Chan}\ c$. The actual type (or state) of channel c is to be found in a channel environment Σ, in the rules for expressions. In T-Abs, the initial and final channel environments of the function body are recorded in the function type.

Typing Expressions (Figures 6, 7). There are two rules for receive and two rules for send, for these constructors are overloaded: they allow transmission of data as well as channels. In T-ReceiveD, the prefix $?D.$, of the type for channel c, is consumed, provided that we are receiving on a value aliased to channel c (of type $\mathsf{Chan}\ c$). In T-ReceiveS, we receive a channel, that we decided to call d; the type of the expression is $\mathsf{Chan}\ d$, and we add a new entry to the final channel environment, where we record the type for d. The particular form of the final channel environment allows

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ c}{\Gamma \vdash \Sigma, c\colon ?D.S \triangleright \mathsf{receive}\ v\colon D \triangleleft \Sigma, c\colon S} \quad \text{(T-ReceiveD)}$$

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ c \qquad d \text{ fresh}}{\Gamma \vdash \Sigma, c\colon ?S'.S \triangleright \mathsf{receive}\ v\colon \mathsf{Chan}\ d \triangleleft \Sigma, d\colon S', c\colon S} \quad \text{(T-ReceiveS)}$$

$$\frac{\Gamma \vdash v\colon D \qquad \Gamma \vdash v'\colon \mathsf{Chan}\ c}{\Gamma \vdash \Sigma, c\colon !D.S \triangleright \mathsf{send}\ v\ \mathsf{on}\ v'\colon \mathsf{Unit} \triangleleft \Sigma, c\colon S} \quad \text{(T-SendD)}$$

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ d \qquad \Gamma \vdash v'\colon \mathsf{Chan}\ c}{\Gamma \vdash \Sigma, c\colon !S'.S, d\colon S' \triangleright \mathsf{send}\ v\ \mathsf{on}\ v'\colon \mathsf{Unit} \triangleleft \Sigma, c\colon S} \quad \text{(T-SendS)}$$

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ c \qquad j \in I}{\Gamma \vdash \Sigma, c\colon \oplus\langle l_i\colon S_i\rangle_{i\in I} \triangleright \mathsf{select}\ l_j\ \mathsf{on}\ v\colon \mathsf{Unit} \triangleleft \Sigma, c\colon S_j} \quad \text{(T-Select)}$$

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ c \qquad \Gamma \vdash \Sigma, c\colon S_j \triangleright e_j\colon T \triangleleft \Sigma' \quad \forall j \in I}{\Gamma \vdash \Sigma, c\colon \&\langle l_i\colon S_i\rangle_{i\in I} \triangleright \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I}\colon T \triangleleft \Sigma'} \quad \text{(T-Case)}$$

$$\frac{\Gamma \vdash v\colon \mathsf{Chan}\ c}{\Gamma \vdash \Sigma, c\colon \mathsf{End} \triangleright \mathsf{close}\ v\colon \mathsf{Unit} \triangleleft \Sigma} \quad \text{(T-Close)}$$

$$\frac{\Gamma \vdash v\colon [S] \qquad c \text{ fresh}}{\Gamma \vdash \Sigma \triangleright \mathsf{request}\ v\colon \mathsf{Chan}\ c \triangleleft \Sigma, c\colon \overline{S}} \qquad \frac{\Gamma \vdash v\colon [S] \qquad c \text{ fresh}}{\Gamma \vdash \Sigma \triangleright \mathsf{accept}\ v\colon \mathsf{Chan}\ c \triangleleft \Sigma, c\colon S} \quad \text{(T-Request,T-Accept)}$$

Figure 6: Typing rules for expressions I: Channel operations

the continuation to hold both ends of the channel. The rules T-SendD and T-SendS, for sending values and channels, are similar. In T-Select, the type for c in the final channel environment is that of branch l_i in the type for c in the source channel environment. In T-Case, all branches must produce the same final channel environment. This enables us to know the environment for any code following the case, independently of which branch is chosen at runtime. The same applies to the two branches of the conditional in T-If. Rule T-Close requires that the channel must be ready to be closed (of type End). We remove the closed channel from the environment.

Rules T-Request and T-Accept both introduce a new channel c in the channel environment, of *dual polarities* [9, 13, 14, 24, 25]. The dual of a session type S, denoted $\overline{S}$, is defined for all session types except $\bot$, 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 8.

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

$$\frac{\Gamma \vdash \Sigma_1 \triangleright t_1 : T_1 \triangleleft \vec{c} : \bot \qquad \Gamma \vdash \Sigma_2 \triangleright t_2 : T_2 \triangleleft \vec{d} : \bot}{\Gamma \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \mathsf{fork}\ t_1; t_2 : T_2 \triangleleft \vec{c} : \bot, \vec{d} : \bot} \quad \text{(T-Fork)}$$

$$\Gamma \vdash \Sigma \triangleright \mathsf{new} : [S] \triangleleft \Sigma \qquad \frac{\Gamma \vdash v : (\Sigma; T \to U; \Sigma') \qquad \Gamma \vdash v' : T}{\Gamma \vdash \Sigma \triangleright v v' : U \triangleleft \Sigma'} \quad \text{(T-New,T-App)}$$

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

$$\frac{\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'' \quad \Gamma, x : T \vdash \Sigma'' \triangleright t : U \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright \mathsf{let}\ x = e\ \mathsf{in}\ t : U \triangleleft \Sigma'} \qquad \frac{\Gamma \vdash \Sigma \triangleright t\{v/x\} : T \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright \mathsf{let}\ x = v\ \mathsf{in}\ t : T \triangleleft \Sigma'} \quad \text{(T-Let,T-PolyLet)}$$

Figure 7: Typing rules for expressions II: Other rules

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

Figure 8: Duality on session types

Rule T-Fork composes the initial channel environments of two configurations, by checking that the types of the channels occurring in both environments are dual. As for the final environment, the rule requires, via the $\vec{c}: \bot$ and $\vec{d}: \bot$ in the antecedent, that each thread involved either consumes their channels (that is sends or closes), or uses them in dual mode.

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

Rule T-PolyLet types the various forms of polymorphism identified in Section 2, by separately typing different copies of the polymorphic value [20, Chapter 22].

Typing Configurations (Figure 9). Rule T-Thread requires that threads either consume their channels or use them in dual mode, similarly to T-Fork. The Δ in the antecedent of rule T-Thread ensures that threads are closed for variables, for the domain of Δ does not include variables. Rule T-Par is similar to T-Fork. T-NewN discards information on the bound name. There are two rules for channel creation. Rule T-NewB says that a newly created channel must be used with dual modes by exactly two

$$\frac{\Delta \vdash \Sigma \triangleright t\colon \_ \triangleleft \vec{c}\colon\ \bot}{\Delta \vdash \Sigma \triangleright \langle t \rangle} \qquad \frac{\Delta \vdash \Sigma_1 \triangleright C_1 \quad \Delta \vdash \Sigma_2 \triangleright C_2}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright C_1 \mid C_2} \qquad \frac{\Delta, n\colon [\_] \vdash \Sigma \triangleright C}{\Delta \vdash \Sigma \triangleright (\nu n)C}$$

(T-Thread,T-Par,T-NewN)

$$\frac{\Delta \vdash \Sigma, c\colon \bot \triangleright C \qquad c \text{ not in } \Delta, \Sigma}{\Delta \vdash \Sigma \triangleright (\nu c)C} \qquad \frac{\Delta \vdash \Sigma \triangleright C \qquad c \text{ not in } \Delta, \Sigma}{\Delta \vdash \Sigma \triangleright (\nu c)C}$$

(T-NewB,T-NewC)

Figure 9: Typing rules for configurations

threads, since the type $\bot$ usually arises from the $\bullet$ operator in rules T-Par or T-Fork. Rule T-NewC allows to garbage collect unused channels.

The formulation of Subject Reduction is standard; the proof is in Appendix A, page 21.

Theorem 1 (Subject Reduction)**.** *If* $\Delta \vdash \Sigma \triangleright C$ *and* $C \to C'$*, then* $\Delta \vdash \Sigma \triangleright C'$*.*

6 Type Safety

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

In order to define what we mean by a faulty configuration, we start by calling a *c-thread* any thread ready to perform an operation on channel c, that is a thread of the form $\langle \mathsf{let}\ x = \mathsf{receive}\ c\ \mathsf{in}\ t \rangle$, and similarly for send, case, select, and close. A *c-redex* is the parallel composition of two threads ready to communicate on channel c, that is $\langle \mathsf{let}\ x = \mathsf{send}\ v\ \mathsf{on}\ c\ \mathsf{in}\ t_1 \rangle \mid \langle \mathsf{let}\ y = \mathsf{receive}\ c\ \mathsf{in}\ t_2 \rangle$, and similarly for case/select, close/close. A *configuration C is faulty* when $C \equiv (\nu \vec{a})(C_1 \mid C_2)$ and C_1 is

1. the thread $\langle \mathsf{let}\ x = e\ \mathsf{in}\ t \rangle$, where e is i) $\mathsf{if}\ v\ \mathsf{then}\ \_\ \mathsf{else}\ \_$ with $v \neq \mathsf{true}, \mathsf{false}$, or is ii) $v\_$ with $v \neq \lambda y.e'$ and $v \neq \mathsf{rec}\ y.e'$; or is
2. the thread $\langle \mathsf{let}\ x = \mathsf{accept/request}\ v\ \mathsf{in}\ t \rangle$, where v is not a name; or is
3. the thread $\langle \mathsf{let}\ x = e\ \mathsf{in}\ t \rangle$, where e is i) $\mathsf{receive/close}\ v$, or ii) $\mathsf{send}\ \_\ \mathsf{on}\ v$, or iii) $\mathsf{case}\ v\ \mathsf{of}\ \_$, or iv) $\mathsf{select}\ \_\ \mathsf{on}\ v$, with v not a channel; or is
4. the parallel composition of two c-threads that do not form a c-redex; or is

5. the parallel composition of three or more c-threads.

The main property of this section says that typable configurations are not faulty; the proof is in Appendix B, page 31.

Theorem 2 (Type Safety)**.** *Typable configurations are not faulty.*

7 Related Work

Cyclone [12] is a C-like type-safe polymorphic imperative language. It features region-based memory management, and more recently threads and locks [11], 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 the protocol for the resource, before closing the channel obtained with request.

In the *Vault* system [5] 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" [7], 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 [10] use one such system to prove progress properties of communication protocols written in π-calculus. Bonelli, Compagnoni, and Gunter [2, 3] combine the language of Honda, Vasconcelos and Kubo [14] 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* [4, 21] 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) we expect verification to be 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 [16]. 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. Walker, Crary, and Morrisett [26] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [15] 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 [19], but the results are limited to type soundness.

Very recently, Dezani-Ciancaglini *et al.* [6] have proposed a minimal distributed object-oriented language with session types. Apart from the use of objects, the main difference between their version of session types and ours seems to be that they do not allow channels to be sent along channels (although objects containing *names* of potential channels may be sent). A more detailed comparison is a subject for future work.

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. We have introduced recursive session types in a previous work [8]. We feel its incorporation in the present setting would not present major difficulties, although care must be taken in the definition of duality [25].

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

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

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

References

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 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*, Electronic Notes on Theoretical Computer Science. Elsevier Science, 2005. To appear.

[4] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pages 45–57. ACM Press, 2002.

[5] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69. ACM Press, 2001.

[6] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *Symposium on Trustworthy Global Computing*. Springer-Verlag, 2005. To appear.

[7] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, pages 1–12, Berlin, Germany, June 2002.

[8] S. Gay, V. T. Vasconcelos, and A. Ravara. Session types for interprocess communication. TR 2003–133, Department of Computing, University of Glasgow, March 2003.

[9] S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.

[10] A. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.

[11] D. Grossman. Type-safe multithreading in cyclone. In *ACM Workshop on Types in Language Design and Implementation*. ACM Press, 2003.

[12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, pages 282–293. ACM Press, 2002.

[13] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.

[14] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[15] A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL*, pages 331–342. ACM Press, 2002.

[16] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2003.

[17] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proc. Lics2002,* 17^{th} *Annual Symposium on Logic in Computer Science, Copenhagen*, pages 101–112. IEEE Computer Society Press, 2002.

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

[19] 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-Verlag, 2004.

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

[21] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. volume 2126 of *LNCS*, pages 375–394. Springer-Verlag, 2001.

[22] J. Reppy. CML: a higher order concurrent language. In *PLDI*, pages 293–305. ACM Press, 1991.

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

[24] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. volume 817 of *LNCS*. Springer-Verlag, 1994.

[25] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *FOCLASA 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier, August 2002.

[26] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *TOPLAS*, 22(4):701–771, 2000.

A Proof of Theorem 1, Subject Reduction

We start with a few auxiliary results; the proof of Subject Reduction is on page 25. To simplify the proofs, we make use of the *variable convention* [1], allowing, for example, to assume that, in sequent $\Delta \vdash \Sigma \triangleright (\nu c)C$, channel c does not occur in either Δ or Σ. Relatedly, when we say that c does not occur in C, we mean that it does not occur free in C and, by the variable convention, that it does not occur bound either.

The following easy results allow to grow and shrink the variable environment of an expression. Weakening is used in Subject Reduction (rule R-Let) and narrowing in the Substitution Lemma 12.[5]

Lemma 3 (Variable Weakening)**.** *Suppose that x does not occur in e, v.*

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

Proof. The proofs, by mutual induction on the derivation of the judgements, are straightforward. □

Lemma 4 (Variable Narrowing)**.** *Suppose that x does not occur in e, v.*

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

Proof. The proofs follow the pattern of the ones above. □

The following two unchallenging results allow to grow and shrink, this time, the name environment of a configuration. They are both used in the proofs of Subject Congruence (rule S-ScopeN), and Weakening is also used in the proofs of Subject Reduction (rule R-New).

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

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

Proof. The proof for configurations is a straightforward induction on the derivation of the judgement, using the result for expressions when the last rule in the derivation tree is T-Thread. The proofs for expressions and for values are by mutual induction. □

Lemma 6 (Name Narrowing)**.** *Suppose that n is not in C, e, v.*

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

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

Proof. The proofs follow the pattern of the ones for the previous lemma. □

The following two results allow to grow and shrink the channel environment of a configuration. Weakening is needed in Subject Reduction (rule R-Close); Narrowing in Subject Congruence (channel extrusion using rule T-NewB).

Lemma 7 (Channel Weakening)**.** *Suppose that c does not occur in any of Σ, Σ', C, e, v.*

1. *If $\Delta \vdash \Sigma \triangleright C$, then $\Delta \vdash \Sigma, c\colon \bot \triangleright C$.*
2. *If $\Gamma \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma'$, then $\Gamma \vdash \Sigma, c\colon \bot \triangleright e\colon T \triangleleft \Sigma', c\colon \bot$.*
3. *If $\Gamma \vdash v\colon (\Sigma; T \to U; \Sigma')$, then $\Gamma \vdash v\colon (\Sigma, c\colon \bot; T \to U; \Sigma', c\colon \bot)$.*

Proof. The proofs follow the pattern of the previous two lemmas. □

Lemma 8 (Channel Narrowing)**.** *Suppose that c does not occur in any of Γ, C, e, v.*

1. *If $\Delta \vdash \Sigma, c\colon S \triangleright C$, then $S = \bot$ and $\Delta \vdash \Sigma \triangleright C$.*
2. *If $\Gamma \vdash \Sigma, c\colon S \triangleright e\colon T \triangleleft \Sigma'$, then $\Sigma' = \Sigma'', c\colon S$ and $\Gamma \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma''$.*
3. *If $\Gamma \vdash v\colon (\Sigma, c\colon S; T \to U; \Sigma', c\colon S)$, then $\Gamma \vdash v\colon (\Sigma; T \to U; \Sigma')$.*

Proof. The proofs are by induction on the possible derivation trees for the judgement.

1. Configurations. The only cases worth mentioning are when the last rule is T-Par or T-Thread; the four remaining cases follow directly by induction.

T-Par. By hypothesis we have a tree of the form:

$$\frac{\Delta \vdash \Sigma_1^* \triangleright C_1 \qquad \Delta \vdash \Sigma_2^* \triangleright C_2}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c\colon S \triangleright C_1 \mid C_2}\ \text{T-Par}$$

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

1. $\Sigma_1^* = \Sigma_1, c\colon S$ and $\Sigma_2^* = \Sigma_2$
2. $\Sigma_1^* = \Sigma_1$ and $\Sigma_2^* = \Sigma_2, c\colon S$
3. $S = \bot$, $\Sigma_1^* = \Sigma_1, c\colon S_1$ and $\Sigma_2^* = \Sigma_2, c\colon \overline{S_1}$

The first two cases follow directly by induction. In the third case, the induction hypothesis implies that $S_1 = \bot$ and $\overline{S_1} = \bot$, meaning that $\Sigma_1^* \bullet \Sigma_2^*$ is not defined. So the third case cannot arise.

T-Thread. By hypothesis we have a tree of the form:

$$\frac{\Delta \vdash \Sigma, c\colon S \triangleright t\colon T \triangleleft \vec{d}\colon\bot}{\Delta \vdash \Sigma, c\colon S \triangleright \langle t \rangle}$$

and use the clause for expressions in this lemma, to conclude that $\vec{d}\colon\bot = \Sigma', c\colon S$ (and therefore $S = \bot$) and that $\Delta \vdash \Sigma \triangleright t\colon T \triangleleft \Sigma'$. The result follows by induction.

2. Expressions. The cases for all the rules in Figure 6 (except T-Case), as well as T-New and T-Val in Figure 7 are direct, for the antecedents (if any) do not mention channel environments. The case for rule T-App uses the clause for

functional values in this lemma. All other cases, except T-Fork follow directly by induction.

T-Fork. By hypothesis we have a proof tree of the form:

$$\frac{\Delta \vdash \Sigma_1^* \triangleright t_1 : T_1 \triangleleft \vec{c}{:}\bot \qquad \Delta \vdash \Sigma_2^* \triangleright t_2 : T_2 \triangleleft \vec{d}{:}\bot}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c : S \triangleright \mathsf{fork}\ t_1; t_2 : T_2 \triangleleft \vec{c}{:}\bot, \vec{d}{:}\bot}\ \text{T-Fork}$$

A similar analyses to that done above in the case of rule T-Par of configurations concludes the proof.

3. Values. Rules T-Const, T-Chan, and T-Name do not apply, for the types in the axioms are not of the required form. Rule T-Var does not apply, for c is not in Γ, by hypothesis. Rule T-Rec follows by induction, and rule T-Abs uses the clause for expressions in this lemma. □

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

Lemma 9 (Channel environment monoid)**.** *Consider the commutative monoid axioms expressed in terms of $(\Sigma, \bullet, \emptyset)$, each in the form LHS = RHS. For each axiom, LHS is defined if and only if RHS is defined, and then they are equal.*

Proof. Directly from the definition of channel environment composition, on page 14. □

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

Lemma 10 (Subject Congruence)**.** *If $\Delta \vdash \Sigma \triangleright C$ and $C \equiv C'$, then $\Delta \vdash \Sigma \triangleright C'$.*

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

When the last rule applied is the commutative monoid rule, we use Lemma 9. For the scope extrusion rules S-ScopeN and S-ScopeC we must consider each rule in both directions; for S-ScopeC we must consider two cases, depending on whether the typing derivation uses T-NewB or T-NewC.

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

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

$$\frac{\dfrac{\Delta \vdash \Sigma_1, c : \bot \triangleright C_1 \qquad c \text{ not in } \Delta, \Sigma_1}{\Delta \vdash \Sigma_1 \triangleright (\nu c) C_1}\ \text{T-NewB} \qquad \Delta \vdash \Sigma_2 \triangleright C_2}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c) C_1 \mid C_2}\ \text{T-Par}$$

From the assumptions in the above tree, we build the following derivation, where we crucially use the variable convention to ensure that c is not in Σ_2.

$$\frac{\dfrac{\Delta \vdash \Sigma_1, c : \bot \triangleright C_1 \qquad \Delta \vdash \Sigma_2 \triangleright C_2}{\Delta \vdash \Sigma_1 \bullet \Sigma_2, c : \bot \triangleright C_1 \mid C_2}\ \text{T-Par} \qquad c \text{ not in } \Delta, \Sigma_1, \Sigma_2}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c)(C_1 \mid C_2)}\ \text{T-NewB}$$

S-ScopeC, left-to-right, T-NewC. Similar to the previous case, again using the variable convention.

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

$$\dfrac{\dfrac{\Delta \vdash \Sigma_1^* \triangleright C_1 \qquad \Delta \vdash \Sigma_2^* \triangleright C_2}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c\colon \bot \triangleright C_1 \mid C_2}\,\text{T-Par} \qquad c \text{ not in } \Delta, \Sigma_1, \Sigma_2}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2) \triangleright (\nu c)(C_1 \mid C_2)}\,\text{T-NewB}$$

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

1. $\Sigma_1^* = \Sigma_1, c\colon \bot$ and $\Sigma_2^* = \Sigma_2$.
2. $\Sigma_1^* = \Sigma_1$ and $\Sigma_2^* = \Sigma_2, c\colon \bot$.
3. $\Sigma_1^* = \Sigma_1, c\colon S$ and $\Sigma_2^* = \Sigma_2, c\colon \overline{S}$.

In case 1 we build the following derivation.

$$\dfrac{\dfrac{\Delta \vdash \Sigma_1, c\colon \bot \triangleright C_1 \qquad c \text{ not in } \Delta, \Sigma_1}{\Delta \vdash \Sigma_1 \triangleright (\nu c) C_1}\,\text{T-NewB} \qquad \Delta \vdash \Sigma_2 \triangleright C_2}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c) C_1 \mid C_2}\,\text{T-Par}$$

In case 2, as by hypothesis of rule S-ScopeC c does not occur free in C_2 (and thus, by the variable convention, it does not occur bound neither), we build the following derivation.

$$\dfrac{\dfrac{\Delta \vdash \Sigma_1 \triangleright C_1 \quad c \notin \Delta, \Sigma_1}{\Delta \vdash \Sigma_1 \triangleright (\nu c) C_1}\,\text{T-NewC} \qquad \dfrac{\Delta \vdash \Sigma_2, c\colon \bot \triangleright C_2 \quad c \notin \Delta, \Sigma_2, C_2}{\Delta \vdash \Sigma_2 \triangleright C_2}\,\text{Lemma 8}}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c) C_1 \mid C_2}\,\text{T-Par}$$

In case 3 the typing derivation gives us the additional hypothesis that $(\Sigma_1, c\colon S) \bullet (\Sigma_2, c\colon \overline{S})$ is defined. As we have $\Delta \vdash \Sigma_2, c\colon \overline{S} \triangleright C_2$ and we know that c is not in C_2, Lemma 8 applies and this judgement is $\Delta \vdash \Sigma_2, c\colon \bot \triangleright C_2$, contradicting the assumption that $(\Sigma_1, c\colon S) \bullet (\Sigma_2, c\colon \overline{S})$ is defined. Therefore this case cannot arise.

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

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

Lemma 11 (Channel replacement)**.** *Suppose that d does not occur in any of $\Gamma, \Sigma, \Sigma', T, e, v$, and c does not occur in e, v.*

1. *If $\Gamma \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma'$, then $\Gamma\{d/c\} \vdash \Sigma\{d/c\} \triangleright e\colon T\{d/c\} \triangleleft \Sigma'\{d/c\}$.*
2. *If $\Gamma \vdash v\colon T$, then $\Gamma\{d/c\} \vdash v\colon T\{d/c\}$.*

Proof. The proof of the two results, by mutual induction, 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 12 (Substitution)**.** *Suppose that* $\Gamma \vdash v\colon T$.

1. *If* $\Gamma, x\colon T \vdash \Sigma \triangleright e\colon U \triangleleft \Sigma'$ *then* $\Gamma \vdash \Sigma \triangleright e\{v/x\}\colon U \triangleleft \Sigma'$.
2. *If* $\Gamma, x\colon T \vdash u\colon U$ *then* $\Gamma \vdash u\{v/x\}\colon U$.

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

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

2. Values. The cases of rules T-Const, T-Chan, and T-Name follow easily, observing that x does not occur in u, and applying Lemma 4. The case of rule T-Var follows trivially, as $u = x$. The case of rule T-Abs uses the result for expressions, and that of rule T-Rec follows by induction. □

We are finally in a position to prove Subject Reduction.

Proof of Theorem 1, page 15. The proof proceeds by induction on the derivation of $C \to C'$. We analyse each reduction rule in Figure 3, page 11, in turn.

R-Init. By hypothesis, we have

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

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

$$\frac{(1) \qquad (2)}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}$$

where (1) is the tree

$$\frac{\dfrac{\Delta \vdash n\colon [S] \qquad d_1 \text{ fresh}}{\Delta \vdash \Sigma_1 \triangleright \mathsf{request}\ n\colon \mathsf{Chan}\ d_1 \triangleleft \Sigma_1, d_1\colon \overline{S}} \quad \Delta, x\colon \mathsf{Chan}\ d_1 \vdash \Sigma_1, d_1\colon \overline{S} \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot}{\dfrac{\Delta \vdash \Sigma_1 \triangleright \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma_1 \triangleright \langle \mathsf{let}\ x = \mathsf{request}\ n\ \mathsf{in}\ t_1\rangle}\ \text{T-Thread}}$$

and (2) is the tree

$$\frac{\dfrac{\Delta \vdash n\colon [S] \qquad d_2 \text{ fresh}}{\Delta \vdash \Sigma_2 \triangleright \mathsf{accept}\ n\colon \mathsf{Chan}\ d_2 \triangleleft \Sigma_2, d_2\colon S} \quad \Delta, y\colon \mathsf{Chan}\ d_2 \vdash \Sigma_2, d_2\colon S \triangleright t_2\colon U \triangleleft \vec{d}\colon \bot}{\dfrac{\Delta \vdash \Sigma_2 \triangleright \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2\colon U \triangleleft \vec{d}\colon \bot}{\Delta \vdash \Sigma_2 \triangleright \langle \mathsf{let}\ y = \mathsf{accept}\ n\ \mathsf{in}\ t_2\rangle}\ \text{T-Thread}}$$

From the assumptions in the above tree we may build the following derivation tree, since as c does not occur in t_1, t_2, by the variable convention it does not occur neither in $\Delta, \Sigma_1, \Sigma_2$. Thus, we are in the conditions of Lemma 11, since d_1 and d_2 are fresh in the assumptions of tree (1) and (2). By the same reason $\Delta\{c/d_i\} = \Delta$, and similarly for Σ_1, Σ_2 and for T, U.

$$\dfrac{\dfrac{\qquad (1^*) \qquad\qquad (2^*) \qquad}{\Delta \vdash \Sigma_1 \bullet \Sigma_2, c\colon \bot \triangleright \langle \mathsf{let}\ x = c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = c\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright (\nu c)(\langle \mathsf{let}\ x = c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = c\ \mathsf{in}\ t_2\rangle)}\ \text{T-NewB}$$

where (1*) is the tree

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash c\colon \mathsf{Chan}\ c}\ \text{T-Chan} \quad \dfrac{\Delta, x\colon \mathsf{Chan}\ d_1 \vdash \Sigma_1, d_1\colon S \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot}{\Delta, x\colon \mathsf{Chan}\ c \vdash \Sigma_1, c\colon S \triangleright t_1\colon T \triangleleft \vec{c}\{c/d_1\}\colon \bot}\ \text{Lemma 11}}{\Delta \vdash \Sigma_1, c\colon \overline{S} \triangleright \mathsf{let}\ x = c\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\{c/d_1\}\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma_1, c\colon \overline{S} \triangleright \langle \mathsf{let}\ x = c\ \mathsf{in}\ t_1\rangle}\ \text{T-Thread}$$

and (2*) is the tree

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash c\colon \mathsf{Chan}\ c}\ \text{T-Chan} \quad \dfrac{\Delta, y\colon \mathsf{Chan}\ d_2 \vdash \Sigma_2, d_2\colon S \triangleright t_2\colon U \triangleleft \vec{d}\colon \bot}{\Delta, y\colon \mathsf{Chan}\ c \vdash \Sigma_2, c\colon S \triangleright t_2\colon U \triangleleft \vec{d}\{c/d_2\}\colon \bot}\ \text{Lemma 11}}{\Delta \vdash \Sigma_2, c\colon S \triangleright \mathsf{let}\ y = c\ \mathsf{in}\ t_2\colon U \triangleleft \vec{d}\{c/d_2\}\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma_2, c\colon S \triangleright \langle \mathsf{let}\ y = c\ \mathsf{in}\ t_2\rangle}\ \text{T-Thread}$$

R-Com. By hypothesis, we have

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

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

$$\dfrac{\qquad (1) \qquad\qquad (2) \qquad}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = \mathsf{receive}\ c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{send}\ v\ \mathsf{on}\ c\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}$$

where Σ is $(\Sigma_1 \bullet \Sigma_2), c\colon \bot, d\colon S'$, and (1) is the tree

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash c\colon \mathsf{Chan}\ c}\ \text{T-Chan} \quad d\ \text{fresh}}{\Delta \vdash \Sigma_1, c\colon ?S'.S \triangleright \mathsf{receive}\ c\colon \mathsf{Chan}\ d \triangleleft \Sigma_1'} \quad \Delta, x\colon \mathsf{Chan}\ d \vdash \Sigma_1' \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot, c\colon \bot}{\Delta \vdash \Sigma_1, c\colon ?S'.S \triangleright \mathsf{let}\ x = \mathsf{receive}\ c\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\colon \bot, c\colon \bot}\ \text{T-Thread}}{\Delta \vdash \Sigma_1, c\colon ?S'.S \triangleright \langle \mathsf{let}\ x = \mathsf{receive}\ c\ \mathsf{in}\ t_1\rangle}$$

where Σ_1' is $\Sigma_1, c\colon S, d\colon S'$. (2) is the tree

$$\dfrac{\dfrac{\Delta \vdash v\colon \mathsf{Chan}\ d \qquad \dfrac{}{\Delta \vdash c\colon \mathsf{Chan}\ c}}{\Delta \vdash \Sigma_2, c\colon !S'.\overline{S}, d\colon S' \triangleright \mathsf{send}\ v\ \mathsf{on}\ c\colon \mathsf{Unit} \triangleleft \Sigma_2'} \quad \Delta, y\colon \mathsf{Unit} \vdash \Sigma_2' \triangleright t_2\colon U \triangleleft \vec{d}\colon \bot}{\dfrac{\Delta \vdash \Sigma_2, c\colon !S'.\overline{S}, d\colon S' \triangleright \mathsf{let}\ y = \mathsf{send}\ v\ \mathsf{on}\ c\ \mathsf{in}\ t_2\colon U \triangleleft \vec{d}\colon \bot}{\Delta \vdash \Sigma_2, c\colon !S'.\overline{S}, d\colon S' \triangleright \langle \mathsf{let}\ y = \mathsf{send}\ v\ \mathsf{on}\ c\ \mathsf{in}\ t_2\rangle}\ \text{T-Thread}}$$

where Σ_2' is $\Sigma_2, c\colon \overline{S}$.

From the assumptions in the above tree, one may build

$$\dfrac{(1^*) \qquad (2^*)}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c\colon \bot, d\colon S' \triangleright \langle \mathsf{let}\ x = v\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}$$

where (1*) is the tree

$$\dfrac{\dfrac{\Delta \vdash v\colon \mathsf{Chan}\ d \qquad \Delta, x\colon \mathsf{Chan}\ d \vdash \Sigma_1, c\colon S, d\colon S' \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot, c\colon \bot}{\Delta \vdash \Sigma_1, c\colon S, d\colon S' \triangleright \mathsf{let}\ x = v\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\colon \bot, c\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma_1, c\colon S, d\colon S' \triangleright \langle \mathsf{let}\ x = v\ \mathsf{in}\ t_1\rangle}\ \text{T-Thread}$$

and (2*) is the tree

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \mathsf{unit}\colon \mathsf{Unit}}\ \text{T-Unit} \qquad \Delta, y\colon \mathsf{Unit} \vdash \Sigma_2, c\colon \overline{S} \triangleright t_2\colon U \triangleleft \vec{d}\colon \bot}{\Delta \vdash \Sigma_2, c\colon \overline{S} \triangleright \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\colon U \triangleleft \vec{d}\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma_2, c\colon \overline{S} \triangleright \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle}\ \text{T-Thread}$$

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

R-Close. By hypothesis, we have

$$\langle \mathsf{let}\ x = \mathsf{close}\ c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{close}\ c\ \mathsf{in}\ t_2\rangle \rightarrow \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle$$

and

$$\dfrac{(1) \qquad (2)}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c\colon \bot \triangleright \langle \mathsf{let}\ x = \mathsf{close}\ c\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{close}\ c\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}$$

where (1) is the tree

$$\dfrac{\dfrac{\cdots \qquad \Delta, x\colon \mathsf{Unit} \vdash \Sigma_1 \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot_1}{\Delta \vdash \Sigma_1, c\colon \mathsf{End} \triangleright \mathsf{let}\ x = \mathsf{close}\ c\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\colon \bot_1}\ \text{T-Let}}{\Delta \vdash \Sigma_1, c\colon \mathsf{End} \triangleright \langle \mathsf{let}\ x = \mathsf{close}\ c\ \mathsf{in}\ t_1\rangle}\ \text{T-Thread}$$

and (2) is the tree below.

$$\dfrac{\dfrac{\cdots \qquad \Delta, y\colon \mathsf{Unit} \vdash \Sigma_2 \triangleright t_2\colon U \triangleleft \vec{c}\colon \bot_2}{\Delta \vdash \Sigma_2, c\colon \mathsf{End} \triangleright \mathsf{let}\ y = \mathsf{close}\ c\ \mathsf{in}\ t_2\colon U \triangleleft \vec{c}\colon \bot_2}\ \text{T-Let}}{\Delta \vdash \Sigma_2, c\colon \mathsf{End} \triangleright \langle \mathsf{let}\ y = \mathsf{close}\ c\ \mathsf{in}\ t_2\rangle}\ \text{T-Thread}$$

From the assumptions in the above tree, one may build

$$\dfrac{\dfrac{(1^*) \qquad\qquad (2^*)}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle}\ \text{T-Par}}{\Delta \vdash (\Sigma_1 \bullet \Sigma_2), c\colon \bot \triangleright \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle \mid \langle \mathsf{let}\ y = \mathsf{unit}\ \mathsf{in}\ t_2\rangle}\ \text{Lemma 7}$$

where (1*) is the tree

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \mathsf{unit}\colon \mathsf{Unit}}}{\Delta \vdash \Sigma_1 \triangleright \mathsf{unit}\colon \mathsf{Unit} \triangleleft \Sigma_1}\ \text{T-Val} \qquad \Delta, x\colon \mathsf{Unit} \vdash \Sigma_1 \triangleright t_1\colon T \triangleleft \vec{c}\colon \bot_1}{\dfrac{\Delta \vdash \Sigma_1 \triangleright \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\colon T \triangleleft \vec{c}\colon \bot_1}{\Delta \vdash \Sigma_1 \triangleright \langle \mathsf{let}\ x = \mathsf{unit}\ \mathsf{in}\ t_1\rangle}\ \text{T-Thread}}\ \text{T-Let}$$

and (2*) is a similar tree.

R-New. By hypothesis, we have

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

and

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \mathsf{new}\colon [S]}}{\Delta \vdash \Sigma \triangleright \mathsf{new}\colon [S] \triangleleft \Sigma}\ \text{T-Val} \qquad \Delta, x\colon [S] \vdash \Sigma \triangleright t\colon T \triangleleft \vec{c}\colon \bot}{\dfrac{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = \mathsf{new}\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = \mathsf{new}\ \mathsf{in}\ t\rangle}\ \text{T-Thread}}\ \text{T-Let}$$

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 T-Let above, and hence,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta, n\colon [S] \vdash n\colon [S]} \qquad \dfrac{\Delta, x\colon [S] \vdash \Sigma \triangleright t\colon T \triangleleft \vec{c}\colon \bot}{\Delta, n\colon [S], x\colon [S] \vdash \Sigma \triangleright t\colon T \triangleleft \vec{c}\colon \bot}\ \text{Lemma 5}}{\Delta, n\colon [S] \vdash \Sigma \triangleright \mathsf{let}\ x = n\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}\ \text{T-Let}}{\Delta, n\colon [S] \vdash \Sigma \triangleright \langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle}\ \text{T-Thread}}{\Delta \vdash \Sigma \triangleright (\nu n)\langle \mathsf{let}\ x = n\ \mathsf{in}\ t\rangle}\ \text{T-NewN}$$

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

$$\dfrac{\dfrac{\Delta \vdash \Sigma_1 \triangleright t\colon \_ \triangleleft \vec{c}\colon \bot \qquad \Delta \vdash \Sigma_2 \triangleright t'\colon \_ \triangleleft \vec{d}\colon \bot}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \mathsf{fork}\ t; t'\colon \_ \triangleleft \vec{c}\colon \bot, \vec{d}\colon \bot}\ \text{T-Fork}}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \langle \mathsf{fork}\ t; t' \rangle}\ \text{T-Thread}$$

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

$$\dfrac{\dfrac{\Delta \vdash \Sigma_1 \triangleright t\colon \_ \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma_1 \triangleright \langle t \rangle}\ \text{T-Thread} \qquad \dfrac{\Delta \vdash \Sigma_2 \triangleright t'\colon \_ \triangleleft \vec{d}\colon \bot}{\Delta \vdash \Sigma_2 \triangleright \langle t' \rangle}\ \text{T-Thread}}{\Delta \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \langle t \rangle \mid \langle t' \rangle}\ \text{T-Par}$$

R-App. By hypothesis, we have

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

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta, y\colon T \vdash \Sigma \triangleright e\colon U \triangleleft \Sigma'}{\Delta \vdash \lambda y.e\colon (\Sigma; T \to U; \Sigma')}\ \text{T-Abs} \qquad \Delta \vdash v\colon T}{\Delta \vdash \Sigma \triangleright (\lambda y.e) v\colon U \triangleleft \Sigma'}\ \text{T-App} \qquad \Delta, x\colon U \vdash \Sigma' \triangleright t\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = (\lambda y.e) v\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}\ \text{T-Thread}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = (\lambda y.e) v\ \mathsf{in}\ t \rangle}\ \text{T-Thread}$$

Then, one may build the following derivation to complete the proof.

$$\dfrac{\dfrac{\dfrac{\Delta \vdash v\colon T \quad \Delta, y\colon T \vdash \Sigma \triangleright e\colon U \triangleleft \Sigma'}{\Delta \vdash \Sigma \triangleright e\{v/y\}\colon U \triangleleft \Sigma'}\ \text{Lemma 12} \qquad \Delta, x\colon U \vdash \Sigma' \triangleright t\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = e\{v/y\}\ \mathsf{in}\ t \rangle}\ \text{T-Thread}$$

R-Rec. By hypothesis, we have

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

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta, y\colon (\Sigma; T \to U; \Sigma') \vdash v\colon (\Sigma; T \to U; \Sigma')}{\Delta \vdash \mathsf{rec}\ y.v\colon (\Sigma; T \to U; \Sigma')}\ \text{T-Rec} \qquad \Delta \vdash u\colon T}{\Delta \vdash \Sigma \triangleright (\mathsf{rec}\ y.v) u\colon U \triangleleft \Sigma'}\ \text{T-App} \qquad (1)}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = (\mathsf{rec}\ y.v) u\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}\ \text{T-Let}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = (\mathsf{rec}\ y.v) u\ \mathsf{in}\ t \rangle}\ \text{T-Thread}$$

where (1) is $\Delta, x\colon U \vdash \Sigma' \triangleright t\colon T \triangleleft \vec{c}\colon \bot$.

Then, one may build the following derivation to complete the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta, y\colon (\Sigma; T \to U; \Sigma') \vdash v\colon (\Sigma; T \to U; \Sigma')}\,\text{T-Rec}}{\Delta \vdash \mathsf{rec}\ y.v\colon (\Sigma; T \to U; \Sigma')}\,\text{Lemma 12}}{\Delta \vdash v\{\mathsf{rec}\ y.v/y\}\colon (\Sigma; T \to U; \Sigma')} \qquad \Delta \vdash u\colon T}\,\text{T-App}}{\Delta \vdash \Sigma \triangleright (v\{\mathsf{rec}\ y.v/y\})u\colon U \triangleleft \Sigma'}\,(1)}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = (v\{\mathsf{rec}\ y.v/y\})u\ \mathsf{in}\ t\colon T \triangleleft \vec{c}\colon \bot}\,\text{T-Let}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = (v\{\mathsf{rec}\ y.v/y\})u\ \mathsf{in}\ t\rangle}\,\text{T-Thread}$$

R-Beta. By hypothesis, we have

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

There are two possible derivations for $\langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle$; we analyse each in turn. When the derivation uses rule T-Let, the result follows by Lemma 12.

$$\dfrac{\dfrac{\dfrac{\Delta \vdash v\colon U}{\Delta \vdash \Sigma \triangleright v\colon U \triangleleft \Sigma}\,\text{T-Val} \qquad \Delta, x\colon U \vdash \Sigma \triangleright t\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = v\ \mathsf{in}\ t\colon \_ \triangleleft \vec{c}\colon \bot}\,\text{T-Let}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle}\,\text{T-Thread}$$

When the derivation uses rule T-PolyLet, the result is immediate.

$$\dfrac{\dfrac{\Delta \vdash \Sigma \triangleright t\{v/x\}\colon T \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle\colon T \triangleleft \vec{c}\colon \bot}\,\text{T-PolyLet}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ x = v\ \mathsf{in}\ t\rangle}\,\text{T-Thread}$$

R-Let. By hypothesis, we have

$$\langle \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ t')\ \mathsf{in}\ t\rangle \to \langle \mathsf{let}\ y = e\ \mathsf{in}\ (\mathsf{let}\ x = t'\ \mathsf{in}\ t)\rangle$$

and

$$\dfrac{\dfrac{\Delta \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma_1 \quad \Delta, y\colon T \vdash \Sigma_1 \triangleright t'\colon T_1 \triangleleft \Sigma_1'}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ y = e\ \mathsf{in}\ t'\colon T_1 \triangleleft \Sigma_1'} \qquad \Delta, x\colon T_1 \vdash \Sigma_1' \triangleright t\colon U \triangleleft \vec{c}\colon \bot}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ t')\ \mathsf{in}\ t\colon U \triangleleft \vec{c}\colon \bot}\,\text{T-Thread}$$

Then,, using Lemma 3 one may build the following derivation to complete the proof. Notice that, by the variable convention, y is not free in t, since it is bound in $\mathsf{let}\ y = e\ \mathsf{in}\ t'$.

$$\dfrac{\dfrac{\Delta \vdash \Sigma \triangleright e\colon T \triangleleft \Sigma_1 \qquad (1)}{\Delta \vdash \Sigma \triangleright \mathsf{let}\ y = e\ \mathsf{in}\ (\mathsf{let}\ x = t'\ \mathsf{in}\ t)\colon U \triangleleft \vec{c}\colon \bot}\,\text{T-Let}}{\Delta \vdash \Sigma \triangleright \langle \mathsf{let}\ y = e\ \mathsf{in}\ (\mathsf{let}\ x = t'\ \mathsf{in}\ t)\rangle}\,\text{T-Thread}$$

where (1) is the tree

$$\dfrac{\Delta, y\colon T \vdash \Sigma_1 \triangleright t'\colon T_1 \triangleleft \Sigma_1' \qquad \dfrac{\Delta, x\colon T_1 \vdash \Sigma_1' \triangleright t\colon U \triangleleft \vec{c}\colon \bot}{\Delta, y\colon T, x\colon T_1 \vdash \Sigma_1' \triangleright t\colon U \triangleleft \vec{c}\colon \bot}\ \text{Lemma 3}}{\Delta, y\colon T \vdash \Sigma_1 \triangleright \mathsf{let}\ x = t'\ \mathsf{in}\ t\colon U \triangleleft \vec{c}\colon \bot}\ \text{T-Let}$$

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

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

R-Conf. The three cases follow directly by induction. For the rule that uses structural congruence, we use Lemma 10. □

B Proof of Theorem 2, Type Safety

We start with a couple of easy results.

Lemma 13. *Suppose that* $\Delta \vdash \Sigma \triangleright C$.

1. *If* C *is a* c*-thread, then* c *is in the domain of* Σ.
2. *If* C *is a* c*-redex, then* Σ *is of the from* $\Sigma', c\colon \bot$.

Proof. 1. A simple analysis of the conclusions of the last rule applied in the derivation of the sequent for c-threads, namely T-SendD, T-SendS, T-ReceiveD, T-ReceiveS, T-Case, T-Select, and T-Close.

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

Proof of Theorem 2, page 16. By contradiction, assuming faulty configurations typable and performing a case analysis on the possible forms of the faulty configurations.

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

$$\Delta, \vec{n}\colon [\vec{S}] \vdash \Sigma_i \triangleright C_i \tag{1}$$

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

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

$$\dfrac{\dfrac{\dfrac{\Delta' \vdash v\colon \mathsf{Bool} \qquad \cdots \qquad \cdots}{\Delta' \vdash \Sigma_1 \triangleright \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\colon \_ \triangleleft \vec{f}\colon \bot}\ \text{T-If} \qquad \cdots}{\Delta' \vdash \Sigma_1 \triangleright \mathsf{let}\ x = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{in}\ t\colon \_ \triangleleft \vec{f}\colon \bot}\ \text{T-Let}}{\Delta' \vdash \Sigma_1 \triangleright \langle \mathsf{let}\ x = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{in}\ t \rangle}\ \text{T-Thread}$$

Analysing the rules for values (Figure 5, page 12), one realises that v can only be true or false, for the T-Var does not apply since variables are not in the domain of

Δ', and the type in the conclusion of the remaining rules (T-Abs, T-Rec, T-Chan, T-Name, Unit) is not Bool.

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

$$\Delta' \vdash \Sigma_1 \triangleright \langle \mathsf{let}\ x = \mathsf{accept}\ v\ \mathsf{in}\ t \rangle$$

to obtain a tree for

$$\Delta' \vdash v\colon [S].$$

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

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

$$\Delta' \vdash \Sigma_1 \triangleright \langle \mathsf{let}\ x = \mathsf{receive}\ v\ \mathsf{in}\ t \rangle$$

to obtain a tree for

$$\Delta' \vdash v\colon \mathsf{Chan}\ c.$$

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

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

$$\Delta' \vdash \Theta_1 \triangleright \mathsf{select}\ l\ \mathsf{on}\ c\colon T_1 \triangleleft \Theta_1' \qquad\qquad \Delta' \vdash \Theta_2 \triangleright \mathsf{close}\ c\colon T_2' \triangleleft \Theta_2'$$

where $\Sigma_1 = \Theta_1 \bullet \Theta_2$. Analysing the rule for select, one finds that $c\colon \oplus\langle l\colon S\rangle$ must be in Θ_1. Similarly, analysing the rule for close one realises that $c\colon \mathsf{End}$ must be in Θ_2. Then, $\Theta_1 \bullet \Theta_2$ is not defined (for $\oplus\langle l\colon S\rangle$ is not the dual of End), hence $(\nu\vec{a})(C_1 \mid C_2)$ is not typable.

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

$$\frac{\Delta' \vdash \Sigma' \triangleright \langle t_1\rangle \mid \langle t_2\rangle \qquad \Delta' \vdash \Sigma'' \triangleright \langle t_3\rangle}{\Delta' \vdash \Sigma_1 \triangleright \langle t_1\rangle \mid \langle t_2\rangle \mid \langle t_3\rangle}\ \text{T-Par}$$

with $\Sigma_1 = \Sigma' \bullet \Sigma''$. If $\langle t_1\rangle \mid \langle t_2\rangle$ is not a c-redex, then we use the previous case. Otherwise, by Lemma 13, it must be the case that $c\colon \bot$ is part of Σ'. Since $\langle t_3\rangle$ is a c-thread, by Lemma 13, c is in the domain of Σ''. But then $\Sigma' \bullet \Sigma''$ is not defined (for $\bot$ is dual to no type), and $(\nu\vec{a})(C_1 \mid C_2)$ is not typable. □