HAL open science

Types for Deadlock-Free Higher-Order Concurrent Programs

Luca Padovani, Luca Novara

▶ To cite this version:

Luca Padovani, Luca Novara. Types for Deadlock-Free Higher-Order Concurrent Programs. 2014. hal-00954364

HAL Id: hal-00954364
https://inria.hal.science/hal-00954364v1

Preprint submitted on 1 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Types for Deadlock-Free Higher-Order Concurrent Programs

Luca Padovani Luca Novara

Dipartimento di Informatica, Università di Torino, Italy

Abstract

Deadlock freedom is for concurrent programs what progress is for sequential ones: it indicates the absence of stable (*i.e.*, irreducible) states in which some pending operations cannot be completed. In the particular case of communicating processes, operations are inputs and outputs on channels and deadlocks may be caused by mutual dependencies between communications. In this work we define an effect system ensuring deadlock freedom of higher-order programs that communicate over *linear channels* and study its integration with polymorphic and recursive types.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Structures—Type structure

Keywords Types and effects, linear channels, deadlock freedom

1. Introduction

The inherent concurrency and parallelism of many software systems calls for programming abstractions to synchronize and exchange information between system components. There is a consequent demand for methods providing formally verified guarantees about the properties of programs making use of these abstractions. In this work, we are concerned with *communication channels* as the abstraction that enables synchronizations and interactions, with *deadlock* (and absence thereof) as the property being considered, and with *types* as the method for enforcing this property.

The history of types for communication channels spans from Milner's sorts [21] to session types [11, 12], going through several variations such as Pierce and Sangiorgi's I/O types [26] (see [30] for a survey). In many cases, channel types are meant to guarantee communication *safety*, namely that only messages of expected type can travel on a given channel. In other cases, such as in [26], the *direction* of messages is also specified. Richer types can guarantee even stronger properties: for example, session types enable the description of whole communication protocols made of an arbitrary, sometimes unbounded, number of message exchanges and can guarantee communication *fidelity* (the assurance that communications occur in the order specified by the protocol) as well as the absence of *deadlocks* (the assurance that the interacting parties keep communicating until the protocol is terminated).

[Copyright notice will appear here once 'preprint' option is removed.]

Deadlocks cannot occur if processes use just *one* communication channel or, more generally, if they are connected in an acyclic network. When processes simultaneously interact through several distinct channels, however, cycles in the network topology may arise and at that point the relative order of communications becomes relevant. A paradigmatic instance of such configurations that yields a deadlock can be observed in the program below

$$\{\ \texttt{send}\ a\ (\texttt{recv}\ b)\ \}\ |\ \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \} \qquad (1)$$

consisting of two threads (within $\{\cdots\}$) running in parallel (the operator $|$). The leftmost thread is trying to send on channel a the message received from channel b; the rightmost thread is trying to do the opposite. This program is in deadlock, for the communications on a and b are mutually dependent. Yet, it is commonly considered well typed, for example if a and b are channels for sending/receiving integer numbers.

In this article we study a typing discipline that flags (1) as ill typed and, more generally, that guarantees well-typed programs to be free from deadlocks. Type systems that ensure this or similar properties have already been defined for process calculi, for example in [17, 18, 25]. We contribute here in three ways:

1. We work with a core functional language equipped with communication primitives *à la* Concurrent ML [28, 29]. As far as we know, we give the first type system ensuring deadlock freedom for a higher-order language. The challenge is to conjugate the *locality* of the typing rules with the *non-locality* of the property (absence of deadlocks) they enforce.
2. We study the integration between the features of types concerning deadlock freedom with polymorphism and recursion. We show that these well-established features, which are essential for coping with common program patterns, must be carefully tuned and revisited in our setting.
3. We focus on communications on *linear channels*, *i.e.* channels that can be used for exactly one communication. In this way we are able to address non-trivial combinations of recursive programs and cyclic network topologies that are out of reach of existing type systems for deadlock freedom.

Of these contributions, the third one may sound questionable given the important restriction – channel linearity – that it relies on. In practice, linear channels account for large fraction of communication channels in many actual systems [13, 20] and structured communications can be modeled using linear channels taking advantage of *channel mobility*, that is sending channels as messages: we can model both buffered and unbuffered communications and, in the former case, with both finite and unbounded size. Finally, binary sessions can be encoded using linear channels [9] and a similar encoding is possible for a large fraction of multiparty sessions as well [25]. The benefits of linearity include an efficient implementation of channels, confluence of (concurrent) computations, and, as we have said, unprecedented accuracy of the typing discipline.

The basic idea for detecting dangerous programs like (1) is very simple: we assign each channel a *priority*, which is just a number, and we verify that sequential input/output operations are performed in an order that is consistent with such numbering: operations on channels with higher priority can only block operations on channels with lower priority (we adopt the convention that "smaller number" means "higher priority"). This mechanism flags (1) as being ill typed because the leftmost thread requires b to have a higher priority than a, and the rightmost thread requires a to have a higher priority than b. No priority assignment can simultaneously satisfy both constraints. The question, then, is how to perform these checks *compositionally*, *i.e.* using types.

The first obvious step is to attach the priority of channels to their types. For instance, we can assign the types $\texttt{![int]}^m$ and $\texttt{?[int]}^n$ respectively to a and b in the leftmost thread of (1), and $\texttt{?[int]}^m$ and $\texttt{![int]}^n$ to the same channels in the rightmost thread of (1). Crucially, distinct occurrences of the same channel have opposite polarities (input ? and output !) and same priority. We can also reasonably think of the assignments $\texttt{send} : \forall \imath.\texttt{![int]}^\imath \to \texttt{int} \to \texttt{unit}$ and $\texttt{recv} : \forall \imath.\texttt{?[int]}^\imath \to \texttt{int}$ for the communication primitives, where we allow polymorphism on channel priorities. In this case, the application $(\texttt{send}\ a\ (\texttt{recv}\ b))$ consists of two subexpressions, the partial application $(\texttt{send}\ a)$ having type $\texttt{int} \to \texttt{unit}$ and its argument $(\texttt{recv}\ b)$ having type $\texttt{int}$. Neither of these types hints at the I/O operations performed while evaluating the corresponding expressions, let alone at the priorities of the channels involved in these operations. In other words, they are not sufficiently informative for checking whether communication channels are used in an order that is consistent with their priority. A standard solution in these cases is to use *effects* [2, 23]: each expression has, along with its type, an effect describing what happens when the expression is evaluated. In our case, the effect is the priority of the channels on which I/O operations are performed, or $\bot$ in the case of pure expressions that perform no I/O. In particular, the judgment

$$b : \texttt{?[int]}^n \vdash \texttt{recv}\ b : \texttt{int}\ \&\ n$$

states that $(\texttt{recv}\ b)$ is an expression of type $\texttt{int}$ whose evaluation performs an I/O operation on a channel with priority n. As usual, function types are decorated with a *latent effect* saying what happens when the function is applied to its argument. So,

$$a : \texttt{![int]}^m \vdash \texttt{send}\ a : \texttt{int} \to^m \texttt{unit}\ \&\ \bot$$

states that $(\texttt{send}\ a)$ is a function that, applied to an argument of type $\texttt{int}$, produces a result of type $\texttt{unit}$ and, in doing so, performs an I/O operation on a channel with priority m. By itself, $(\texttt{send}\ a)$ is a pure expression whose evaluation performs no I/O operations, hence the effect $\bot$. With this information we can detect dangerous expressions: in a *call-by-value* language an application $e_1 e_2$ first evaluates e_1, then e_2, and then the body of the function resulting from e_1. These evaluations are deadlock free if e_1 does not perform operations on channels with priority lower than those occurring in e_2 ($\bot < n$, which is trivially satisfied) and the evaluation of e_2 does not perform operations on channels with priority lower than those occurring in e_1 ($m < n$). Since the same analysis on $(\texttt{send}\ b\ (\texttt{recv}\ a))$ requires the symmetric condition ($n < m$), the parallel composition of the two threads in (1) is ill typed due to unsatisfiable constraints between the priorities of a and b.

It turns out that the information given by latent effects in function types is insufficient for detecting some deadlocks. To see why, consider the function defined by

$$\texttt{let}\ f = \texttt{fun}\ x\ (\texttt{send}\ a\ x;\ \texttt{send}\ b\ x)$$

which sends its argument x on both a and b and where ; denotes the standard sequential composition. The priority n of b should be lower than the priority m of a, for b is used only after the communication on a has completed. The point here is how to choose the priority that decorates the type of f, which must have the form $\texttt{int} \to^h \texttt{unit}$. Each of the two obvious possibilities is dangerous: if we take $h = m$, then

$$\{\ \texttt{recv}\ a\ \}\ |\ \{\ f\ 3;\ \texttt{recv}\ b\ \} \tag{2}$$

is well typed because the latent effect m of $(f\ 3)$ is numerically smaller than the priority of b in $(\texttt{recv}\ b)$, which agrees with the fact that $(f\ 3)$ is evaluated *before* $(\texttt{recv}\ b)$; if we take $h = n$, then

$$\{\ \texttt{recv}\ a;\ f\ 3\ \}\ |\ \{\ \texttt{recv}\ b\ \} \tag{3}$$

is well typed for similar reasons. This is unfortunate because (3) is, and (2) reduces to, a deadlock. To flag both (2) and (3) as ill typed, we must refine the type of f to $\texttt{int} \to^{m,n} \texttt{unit}$ where we keep track of the highest priority of the channels that *occur* free in the body of f (that is m) as well as of the lowest priority of the channels on which an input/output operation is *performed* by f when f is applied to an argument (that is n). Intuitively, the first decoration is the "priority" of the whole function, which gives information on the free channels that occur within the function, while the second decoration is the latent effect of the function, as before. So (2) is ill typed because the latent effect of $(f\ 3)$ is the same as the priority of b in $(\texttt{recv}\ b)$ and (3) is ill typed because the effect of $(\texttt{recv}\ a)$ is the same as the priority of f in $(f\ 3)$.

Outline. In what follows we turn the technique sketched in here into a full-fledged type and effect system. We proceed defining the language (Section 2), the type system (Section 3) and studying the properties of well-typed programs (Section 3.4). A more comprehensive comparison with related work is postponed to Section 4. Section 5 discusses some limitations of the approach and hints at possible solutions. *Appendix A, which is not formally part of the submission, contains proofs and additional technical material.*

2. Language Syntax & Semantics

We use a countable set of *variables* $x, y, \ldots$, a countable set of *channels* $a, b, \ldots$, and a set of constants $\texttt{c}$. *Names* $u, \ldots$ are either variables or channels. We consider a language of *expressions* $e, \ldots$ and *processes* $P, Q, \ldots$ as defined below:

$$\begin{array}{rrcl} \textbf{expression} & e & ::= & \texttt{c} \mid u \mid (e, e) \mid ee \mid \texttt{fun}\ x\ e \mid \texttt{rec}\ x\ e \\ & & & \mid \texttt{let}\ (x, y) = e\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ \textbf{process} & P & ::= & \{e\} \mid P \mid P \mid \texttt{new}\ a\ \texttt{in}\ P \end{array}$$

Expressions comprise constants $\texttt{c}$, names u, abstractions $\texttt{fun}\ x\ e$, applications $e_1 e_2$, pairs (e_1, e_2), and recursion $\texttt{rec}\ x\ e$. We also have two $\texttt{let}$-binding constructs: a classical one $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ and another $\texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2$ that is used for decomposing pairs (the presence of this construct in place of the more conventional projections is due to the linear nature of some types). Constants include the unitary value $()$, the integer numbers $m, n, \ldots$, as well as the primitives $\texttt{fork}$, $\texttt{open}$, $\texttt{send}$, $\texttt{recv}$ whose semantics will be explained shortly. Processes are either sequential threads $\{e\}$ made of a single expression, or the parallel composition $P \mid Q$ of processes, or restrictions $\texttt{new}\ a\ \texttt{in}\ P$ denoting a communication channel with scope P.

The notions of free and bound names are as expected, given that the binders are abstractions, $\texttt{let}$s, and $\texttt{new}$s. We identify terms modulo renaming of bound names and we write $\mathsf{fn}(e)$ (respectively, $\mathsf{fn}(P)$) for the set of names occurring free in e (respectively, in P). As usual we assume that application is left associative, hence $e_1 e_2 e_3$ stands for $(e_1 e_2) e_3$. We also sugar the syntax with some evocative notation: we write $e_1\texttt{!}e_2$ for $\texttt{send}\ e_1\ e_2$; we write _ for unused/fresh variables; we write $e_1;\ e_2$ for $\texttt{let}\ _ = e_1\ \texttt{in}\ e_2$; we write $\texttt{let}\ f\ x_1 \cdots x_n = e_1\ \texttt{in}\ e_2$ as an abbreviation for

Reduction of expressions

$$
\begin{array}{rcll}
(\texttt{fun}\ x\ e)\mathsf{v} & \longrightarrow & e\{\mathsf{v}/x\} & \\
\texttt{rec}\ x\ e & \longrightarrow & e\{\texttt{rec}\ x\ e/x\} & \\
\texttt{let}\ (x,\ y)\ \texttt{=}\ (\mathsf{v},\ \mathsf{w})\ \texttt{in}\ e & \longrightarrow & e\{\mathsf{v},\mathsf{w}/x,y\} & \\
\texttt{let}\ x\ \texttt{=}\ \mathsf{v}\ \texttt{in}\ e & \longrightarrow & e\{\mathsf{v}/x\} & \\
\mathscr{E}[e] & \longrightarrow & \mathscr{E}[e'] & \text{if } e \longrightarrow e'
\end{array}
$$

Structural congruence

$$
\begin{array}{rcl}
P \mid \{()\} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
\texttt{new}\ a\ \texttt{in}\ P \mid Q & \equiv & \texttt{new}\ a\ \texttt{in}\ (P \mid Q) \text{ if } a \notin \mathsf{fn}(Q)
\end{array}
$$

Reduction of processes

$$
\begin{array}{rcll}
\{\mathscr{E}[a\texttt{!}\mathsf{v}]\} \mid \{\mathscr{E}'[\texttt{recv}\ a]\} & \xrightarrow{a} & \{\mathscr{E}[()]\} \mid \{\mathscr{E}'[\mathsf{v}]\} & \\
\{\mathscr{E}[\texttt{fork}\ \mathsf{v}]\} & \xrightarrow{\tau} & \{\mathscr{E}[()]\} \mid \{\mathsf{v}()\} & \\
\{\mathscr{E}[\texttt{open}()]\} & \xrightarrow{\tau} & \texttt{new}\ a\ \texttt{in}\ \{\mathscr{E}[a]\} & \text{if } a \notin \mathsf{fn}(\mathscr{E}) \\
\{e\} & \xrightarrow{\tau} & \{e'\} & \text{if } e \longrightarrow e' \\
P \mid Q & \xrightarrow{\ell} & P' \mid Q & \text{if } P \xrightarrow{\ell} P' \\
\texttt{new}\ a\ \texttt{in}\ P & \xrightarrow{\ell} & \texttt{new}\ a\ \texttt{in}\ Q & \text{if } P \xrightarrow{\ell} Q \\
 & & & \text{where } \ell \neq a \\
\texttt{new}\ a\ \texttt{in}\ P & \xrightarrow{\tau} & Q & \text{if } P \xrightarrow{a} Q \\
P & \xrightarrow{\ell} & Q & \text{if } P \equiv \xrightarrow{\ell} \equiv Q
\end{array}
$$

Table 1. Reduction semantics of expressions and processes.

$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x_1\ \cdots\ \texttt{fun}\ x_n\ e_1\ \texttt{in}\ e_2$ and $\texttt{let rec}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ as an abbreviation for $\texttt{let}\ f\ \texttt{=}\ \texttt{rec}\ f\ e_1\ \texttt{in}\ e_2$. In the examples we often omit the `in` part of a `let`, in which cases it is meant to be the rest of the program.

The reduction semantics of the language is given by two relations, one for expressions, another for processes (Table 1). We adopt a *call-by-value* reduction strategy, for which we need to define the *reduction contexts* $\mathscr{E}, \ldots$ and the set of *values* $\mathsf{v}, \mathsf{w}, \ldots$:

$$
\begin{array}{rrcl}
\textbf{context} & \mathscr{E} & ::= & [\,] \mid (\mathscr{E},\ e) \mid (\mathsf{v},\ \mathscr{E}) \mid \mathscr{E}e \mid \mathsf{v}\mathscr{E} \\
 & & \mid & \texttt{let}\ (x,\ y)\ \texttt{=}\ \mathscr{E}\ \texttt{in}\ e \mid \texttt{let}\ x\ \texttt{=}\ \mathscr{E}\ \texttt{in}\ e \\
\textbf{value} & \mathsf{v} & ::= & \texttt{c} \mid a \mid (\mathsf{v},\ \mathsf{v}) \mid \texttt{fun}\ x\ e \mid \texttt{send}\ a
\end{array}
$$

Reductions are either the beta rule that applies a function to its argument, the unfolding of a recursion, the splitting of a pair into its components, or the binding of a value to a variable. As usual $e\{\mathsf{v}/x\}$ denotes the expression obtained by replacing the free occurrences of x in e with v and reduction is closed by reduction contexts. Structural congruence rearranges equivalent processes and is directly linked to that of the π-calculus [21, 30]. The reduction relation of processes has *labels* $\ell, \ldots$ that are either a channel name a, denoting that a message has been exchanged on a, or the special symbol τ denoting any other reduction. There are four base reductions for processes: a synchronization can happen between two threads where one is willing to send a message v on a channel a and the other is waiting for a message from the same channel; a thread that contains a subexpression `fork` v spawns a new thread that evaluates $\mathsf{v}()$; a thread that contains a subexpression `open()` causes the creation of a new channel; the reduction of an expression causes a corresponding τ-labeled reduction of the thread in which it occurs. Reduction for processes is then closed by structural congruence, parallel compositions, and restrictions. In the latter case, the restriction of a disappears as soon as a communication on a occurs, witnessing the fact that channels are *linear* in our model and can be used for one communication only.

We conclude this section with a variety of examples whose purpose is threefold: first of all, we show how to model a number of significant communication patterns using linear channels; second, we argue about desirable properties of programs, in particular deadlock and absence thereof; finally, we put together a reasonably complete test bench for the type system that we develop in Section 3. In the examples we use a slightly more concrete language equipped with conditionals and a few additional operators. These can be easily accommodated without affecting the results in Section 3.

Example 2.1. Linear channels can be used for notifying the completion of independent subcomputations. For example, the `fibo` function below computes the n-th number in the Fibonacci sequence by spawning each recursive call in a distinct thread:

```
let rec fibo n =
  if n ≤ 1 then n
  else let (a, b) = (open(), open()) in
       fork fun _ (send a (fibo (n - 1)));
       fork fun _ (send b (fibo (n - 2)));
       (recv a) + (recv b)
```

Before the recursive calls, two new channels a and b are created. As soon as each recursive call terminates, the corresponding result is sent on a and b. These values are collected and added in `(recv a) + (recv b)`. Each channel syntactically occurs twice, but is used for exactly one communication. Note that the function is non-deterministic, in the sense that the spawned threads that evaluate recursive calls execute independently at possibly different speeds; also, expressions are intermingled with (blocking) input/output operations. It is therefore relevant to ask whether this version of `fibo` is *equivalent* to the sequential one, in the sense that it is able to reduce until a result is computed without blocking indefinitely on an input/output operation, and whether it always produces the same result, when applied to the same argument. ■

Example 2.2. Structured communications involving the exchange of several messages can be modeled using *continuations*. To this aim, it is convenient to define two functions

```
let ssend x y =          let asend x y =
  let c = open() in        let c = open() in
  x!(y, c); c              fork fun _ x!(y, c); c
```

that send a message y on channel x along with a fresh continuation c. The continuation is returned so that further messages can be sent in the same conversation. For example, writing $e_1\texttt{!!}e_2$ in place of `(ssend` e_1 e_2`)` and taking `!` and `!!` left-associative, the thread

```
{ a!!1!!2!3 }
```

sends three subsequent messages `1`, `2`, and `3` that the target thread can receive with a sequence of `recv` and pair splittings, thus:

```
{ let (v1, a') = recv a in
  let (v2, a'') = recv a' in
  let v3 = recv a'' in ... }
```

Note the use of `fork` in the definition of `asend` meaning that `asend`, unlike `ssend`, realizes a *non-blocking* output operation: the complete evaluation of `(asend` a `v)` does not guarantee that `v` has been received, but only that `v` has been sent on a. In other words, `asend` allows us to model asynchronous communication on unbounded FIFO buffers. ■

Example 2.3. Many parallel algorithms (Jacobi and Gauss-Seidel, leader election, vertex coloring, just to mention a few) use batteries of threads to elaborate some compound data structure. Each "node"

of the structure is assigned a distinct thread which iteratively communicates with its neighbors. To maximize parallelism, communication is *full duplex*, that is threads simultaneously send messages to each other. We can model full-duplex communication using pairs of linear channels, where one channel is used for sending messages to, and the other channel is used for receiving messages from, the neighbor thread. The `node` function below models one such thread:

```
1  let rec node state f x y =
2    let x' = asend x state in
3    let (state', y') = recv y in
4    node f (f state state') x' y'
5  in let (a, b) = (open(), open()) in
6    fork fun _ (node state_0 f_0 a b);
7    fork fun _ (node state_1 f_1 b a)
```

The thread is parametric in a *state* (*e.g.*, the current value of the node assigned to the thread), a function f that computes a new state from the current one and that of the neighbor, and two channels x and y for respectively sending the thread's own state (line 2) and receiving that of the neighbor (line 3). At the end of each iteration a new state $(f\ \ state\ \ state')$ is computed (line 4). Lines 5–7 instantiate two nodes that are initialized with unspecified initial states $state_i$ and transformation functions f_i.

An interesting aspect of the system above is that the two threads communicate in a cyclic network topology (the channel that one thread uses for sending is the same channel that the other thread uses for receiving, and vice versa) and that they both iteratively interleave input (*i.e.*, potentially blocking) actions on different channels. It may not be obvious that a system like this runs without deadlocks. For instance, while this is the case for the particular system above, using `ssend` instead of `asend` in line 2 or swapping lines 2 and 3 produces another system that is deadlock. ■

Example 2.4. In this example we model a stream processing network that computes the series of Fibonacci numbers. The network is taken from [29] and relies on the small library of stream combinators below (we keep using `!!` to denote `ssend` of Example 2.2):

```
let rec link x y =
  let (v, x') = recv x in link x' y!!v

let delay x y =
  let (_, x') = recv x in link x' y

let rec add x y z =
  let (v, x') = recv x in
  let (w, y') = recv y in add x' y' z!!(v + w)

let rec copy x y z =
  let (v, x') = recv x in copy x' y!!v z!!v
```

Each combinator operates on two or three streams that we model using linear channels as we have already done in Examples 2.2 and 2.3: `link` x y forwards all messages received from x to y; `delay` x y does the same except that it discards the first message received from x; `add` x y z receives integers from x and y and sends their sum to z; finally, `copy` x y z forwards any message from x to both y and z.

We use these combinators for building the network in Figure 1: each integer sent on c_1 is duplicated to c_2 and out; pairs of subsequent integers sent on c_2 are added together thanks to the `copy`, `delay`, and `add` combinators and the result of the addition is injected back on c_1. The integers retrieved from out are the Fibonacci series if the first two messages sent on c_1 are both `1`. In code:

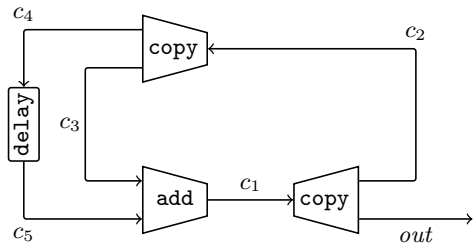

Figure 1. The Fibonacci stream network [29].

```
1  let mk_fibo_network () =
2    let (c_1, c_2) = (open(), open()) in
3    let (c_3, c_4) = (open(), open()) in
4    let (c_5, out) = (open(), open()) in
5    let c_1' = asend (asend c_1 1) 1 in
6    fork fun _ (delay c_4 c_5);
7    fork fun _ (copy c_2 c_3 c_4);
8    fork fun _ (add c_3 c_5 c_1');
9    fork fun _ (copy c_1 c_2 out); out
```

The example is interesting in many ways: in particular, it assembles a network compositionally in terms of simpler building blocks, the realized network contains cyclic dependencies (the messages sent on c_1 are necessary for generating further messages on the continuations of c_1) and interleaves I/O actions on different channels. Again, the fact that the network works as expected is not obvious. Already in [29] it is observed that the absence of deadlocks in this program, hence the actual production of an infinite sequence of messages on out, depends on the careful implementation and composition of the combinators. In particular: `add` must first read the non-delayed message, for otherwise the top `copy` combinator would block trying to deliver a message on c_3 before sending the one on c_4; the two outputs on line 5 must be asynchronous, for otherwise the program would block there before spawning the combinator that receives these messages (line 9); forgetting either of the two outputs on line 5 would also cause a deadlock, because `delay` discards one of them. ■

3. Type System

We introduce the features of the type system gradually, in three steps: we start with a monomorphic system (Section 3.1) to get some familiarity with priorities and types, then we add polymorphism (Section 3.2), and finally recursive types (Section 3.3). We conclude with the properties of the type system (Section 3.4). The reader interested in presentation of the full type system in one shot may refer to Appendix A.

3.1 Core Types

Let $\mathbb{P} \stackrel{\text{def}}{=} \mathbb{Z} \cup \{\bot, \top\}$ be the set of *priorities* ordered in the obvious way ($\bot < n < \top$ for every $n \in \mathbb{Z}$); we use $\rho, \sigma, \ldots$ to range over priorities and we write $\rho \wedge \sigma$ (respectively, $\rho \vee \sigma$) for the *minimum* (respectively, the *maximum*) of ρ and σ. We let $p, \ldots$ range over the set $\{\texttt{?}, \texttt{!}, \texttt{\#}\}$ of *polarities*. *Types* $t, s, \ldots$ are defined by

$$\textbf{type} \quad t \quad ::= \quad \texttt{B} \;\big|\; t \times t \;\big|\; p[t]^n \;\big|\; t \rightarrow^{\rho,\rho} t$$

where *basic types* `B`, ... include `unit` and `int` and $t \times s$ is the usual product type. The type $p[t]^n$ denotes a channel with priority n that can be used to exchange messages with type t according to p: `?` means that the channel can be used once for an input, `!` means that it can be used once for an output, and `#` means that it can be used once for an input *and* once for an output. The arrow $t \rightarrow^{\rho,\sigma} s$ is the type of a function that accepts an argument of type t and returns

a value of type s. The function has priority ρ (its closure contains channels with priority ρ or lower) and, when applied, performs I/O operations on channels with priority σ or higher. We write $\rightarrow$ as an abbreviation for $\rightarrow^{\top,\bot}$, that is a pure function not containing channels and not performing any I/O.

It is useful to extend the notion of priority to arbitrary types: basic types have the lowest priority $\top$ because their (lack of) use does not affect deadlock freedom in any way, while the priority of a pair is the highest (numerical minimum) of the priorities of its components. Formally, the priority of t, written $|t|$, is defined as:

$$|t| \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } t = \mathtt{B} \\ |t_1| \wedge |t_2| & \text{if } t = t_1 \times t_2 \\ \rho & \text{if } t = p[s]^{\rho} \text{ or } t = t_1 \rightarrow^{\rho,\sigma} t_2 \end{cases}$$

We use priorities to distinguish between linear and unlimited types. Linear types denote values (such as channels) that *must* be used to guarantee deadlock freedom; unlimited types denote values that have no effect on deadlock freedom and *may not* be used.

Definition 3.1. We say that t is *linear* if $|t| \in \mathbb{Z}$; we say that t is *unlimited*, written $\mathsf{un}(t)$, if $|t| = \top$.

Below are the types of each constant that we consider. We say "types" instead of "type" because, until the introduction of polymorphism in Section 3.2, the same constant has in general several types, and the right one is "guessed" depending on the context. We write $\mathsf{TypeOf}(\mathtt{c})$ for *one of* the types of $\mathtt{c}$.

$$\begin{array}{lll} & \mathtt{open} : \mathtt{unit} \rightarrow \#[t]^n & n < |t| \\ \mathtt{()} : \mathtt{unit} & \mathtt{recv} : ?[t]^n \rightarrow^{\top,n} t & n < |t| \\ n : \mathtt{int} & \mathtt{send} : ![t]^n \rightarrow t \rightarrow^{n,n} \mathtt{unit} & n < |t| \\ & \mathtt{fork} : (\mathtt{unit} \rightarrow^{\rho,\sigma} \mathtt{unit}) \rightarrow \mathtt{unit} & \end{array}$$

The primitive $\mathtt{open}$ creates a new channel with the full set # of polarities and arbitrary (but finite) priority n.

The primitive $\mathtt{recv}$ takes a channel of type $?[t]^n$, blocks until a message is received, and returns the message. The primitive itself contains no free channels in its closure (hence the priority $\top$) because the only channel it manipulates is its argument. The latent effect is the priority of the channel, as expected.

The primitive $\mathtt{send}$ takes a channel of type $![t]^n$, a message of type t, and sends the message on the channel. Note that the partial application $\mathtt{send}\ a$ is a function whose priority is the same as that of a, and that the latent effect is again the priority of a. Note also that in $\mathtt{open}$, $\mathtt{recv}$, and $\mathtt{send}$ the priority of the message must be lower than the priority of the channel. Since priorities are used to enforce an order on the use of values, this condition makes sense given that a message cannot be used until *after* it has been received, namely after the channel on which it travels has been used.

Finally, $\mathtt{fork}$ accepts a thunk with arbitrary priority ρ and latent effect σ and spawns the thunk into an independent thread (see Table 1). Note that $\mathtt{fork}$ is a pure function, with lowest priority and no latent effect, regardless of the priority and latent effect of the thunk. This phenomenon is an instance of *effect masking*, whereby the effect of evaluating an expression becomes unobservable: in our case, $\mathtt{fork}$ discharges effects because deadlocks are caused by sequential (rather than parallel) execution of input/output operations.

We now turn to the core set of typing rules. A *type environment* $\Gamma, \ldots$ is a finite map from names to types defined by

$$\Gamma ::= \emptyset \ \mid\ \Gamma, u : t$$

and considered modulo commutativity of bindings. We write $\mathsf{dom}(\Gamma)$ for the domain of Γ, namely the set of names for which there is a binding in Γ, and $\Gamma(u)$ for the type associated with u in Γ; we write Γ_1, Γ_2 for the union of Γ_1 and Γ_2 when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. As usual in substructural type systems, we need a more flexible way of combining type environments. In particular, we want to be sure that every linear name is used linearly and we must be able to distribute the polarities of a channel to different parts of the program. To this aim, and following [20, 30], we define a partial *combination* operator $+$ between types:

$$\begin{array}{rcll} t + t & \stackrel{\text{def}}{=} & t & \text{if } \mathsf{un}(t) \\ p[t]^n + q[t]^n & \stackrel{\text{def}}{=} & \#[t]^n & \text{if } \{p, q\} = \{?, !\} \end{array} \tag{4}$$

which we extend to type environments, thus:

$$\begin{array}{rcll} \Gamma + \Gamma' & \stackrel{\text{def}}{=} & \Gamma, \Gamma' & \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset \\ (\Gamma, u : t) + (\Gamma', u : s) & \stackrel{\text{def}}{=} & (\Gamma + \Gamma'), u : t + s & \end{array}$$

For example, we have $(x : \mathtt{int}, a : ![\mathtt{int}]^n) + (a : ?[\mathtt{int}]^n) = x : \mathtt{int}, a : \#[\mathtt{int}]^n$, so we might have some part of the program that (possibly) uses a variable x of type $\mathtt{int}$ along with channel a for sending an integer and another part of the program that uses the same channel a but this time for receiving an integer. The first part of the program would be typed in the environment $x : \mathtt{int}, a : ![\mathtt{int}]^n$ and the second one in the environment $a : ?[\mathtt{int}]^n$. Overall, the two parts would be typed in the environment $x : \mathtt{int}, a : \#[\mathtt{int}]^n$ indicating that a is used for both sending and receiving an integer. It is easy to show that $+$ is commutative and associative.

We extend the function $|\cdot|$ to type environments so that $|\Gamma| \stackrel{\text{def}}{=} \bigwedge_{u \in \mathsf{dom}(\Gamma)} |\Gamma(u)|$ with the convention that $|\emptyset| = \top$; we write $\mathsf{un}(\Gamma)$ if $|\Gamma| = \top$, namely if every type in the range of Γ is unlimited.

We are now ready to discuss the core typing rules, presented in Table 2 and deriving judgments of the form $\Gamma \vdash e : t \,\&\, \rho$ for expressions, denoting that e is well typed in Γ, it has type t and effect ρ and judgments of the form $\Gamma \vdash P$ for processes, simply denoting that P is well typed in Γ.

Rules for expressions are quite conventional for a substructural type system: axioms [T-NAME] and [T-CONST] verify that the unused part of the type environment is unlimited and rules such as [T-APP], [T-PAIR], and [T-SPLIT] that contain two subexpressions split the type environment using the $+$ operator (we omit the simple $\mathtt{let}$ binding operator which, in the monomorphic type system, is just a simpler variant of splitting).

Rule [T-REC] for recursions requires the environment to be unlimited, since a recursive term will unfold an arbitrary number of times and therefore cannot contain channels (but it general it will be a function that may accept channels). The condition prevents the typing of an expression such as $\mathtt{rec}\ x\ .a!x$, which has type $\mathtt{unit}$ and would otherwise be well typed in the environment $a : ![\mathtt{unit}]^n$.

Let us now focus on priorities and effects. Names and constants have no effect ($\bot$); they denote fully evaluated, pure expressions that do not reduce any further. In rule [T-FUN], the effect ρ caused by evaluating the body of the function becomes the latent effect in the arrow type of the function, and the function itself has no effect ($\bot$). In addition, the arrow is annotated with the priority of the environment Γ in which the function is typed. Intuitively, the names in Γ are stored in the *closure* of the function; if any of these names is a channel, then we must be sure that the function is eventually used (*i.e.*, applied) in order to guarantee deadlock freedom. In fact, the priority $|\Gamma|$ gives a slightly more precise information, since it keeps track of the highest priority of any channel that occurs in the closure of the function. We have seen in Section 1 why this information is useful. Below are a few examples:

- the identity function $\mathtt{fun}\ x\ x$ has type $\mathtt{int} \rightarrow^{\top,\bot} \mathtt{int}$ in any unlimited environment; in particular, it contains no channels in its closure and it performs no I/O, so it is pure;
- the function $\mathtt{fun}\ x\ (x,\ a)$ has type $\mathtt{int} \rightarrow^{n,\bot} (\mathtt{int} \times ![\mathtt{int}]^n)$ in the environment $a : ![\mathtt{int}]^n$; it contains channel a with priority n in its closure (whence the priority n in the arrow),

Typing of expressions

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, u : t \vdash u : t \,\&\, \bot}\ [\text{T-NAME}] \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \texttt{c} : \mathsf{TypeOf}(\texttt{c}) \,\&\, \bot}\ [\text{T-CONST}] \qquad \frac{\mathsf{un}(\Gamma) \qquad \Gamma, x : t \vdash e : t \,\&\, \bot}{\Gamma \vdash \texttt{rec}\ x\ e : t \,\&\, \bot}\ [\text{T-REC}]$$

$$\frac{\Gamma, x : t \vdash e : s \,\&\, \rho}{\Gamma \vdash \texttt{fun}\ x\ e : t \to^{|\Gamma|,\rho} s \,\&\, \bot}\ [\text{T-FUN}] \qquad \frac{\Gamma_1 \vdash e_1 : t \to^{\rho,\sigma} s \,\&\, \tau_1 \qquad \Gamma_2 \vdash e_2 : t \,\&\, \tau_2 \qquad \tau_1 < |\Gamma_2| \qquad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \,\&\, \sigma \vee \tau_1 \vee \tau_2}\ [\text{T-APP}]$$

$$\frac{\Gamma_i \vdash e_i : t_i \,\&\, \rho_i \ {}^{(i=1,2)} \qquad \rho_1 < |\Gamma_2| \qquad \rho_2 < |t_1|}{\Gamma_1 + \Gamma_2 \vdash (e_1\texttt{,}\ e_2) : t_1 \times t_2 \,\&\, \rho_1 \vee \rho_2}\ [\text{T-PAIR}] \qquad \frac{\Gamma_1 \vdash e_1 : t_1 \times t_2 \,\&\, \rho \qquad \Gamma_2, x : t_1, y : t_2 \vdash e_2 : s \,\&\, \sigma \qquad \rho < |\Gamma_2|}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ (x\texttt{,}\ y) \texttt{=} e_1\ \texttt{in}\ e_2 : s \,\&\, \rho \vee \sigma}\ [\text{T-SPLIT}]$$

Typing of processes

$$\frac{\Gamma \vdash e : \texttt{unit} \,\&\, \rho}{\Gamma \vdash \{e\}}\ [\text{T-THREAD}] \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}\ [\text{T-PAR}] \qquad \frac{\Gamma, a : \texttt{\#}[t]^n \vdash P}{\Gamma \vdash \texttt{new}\ a\ \texttt{in}\ P}\ [\text{T-NEW}]$$

Table 2. Core typing rules for expressions and processes.

but it does not use a for input/output (whence the latent effect $\bot$); it is nonetheless well typed because a, which is a linear value, is returned as result;

- the function $\texttt{fun}\ x\ x\texttt{!}3$ has type $\texttt{![int]}^n \to^{\top,n} \texttt{unit}$; it has no channels in its closure (priority $\top$) but it performs an output on the channel it receives as argument (priority n);
- the function $\texttt{fun}\ x\ (\texttt{recv}\ a \texttt{+} x)$ has type $\texttt{int} \to^{n,n} \texttt{int}$ in the environment $a : \texttt{?[int]}^n$; note that neither the domain nor the codomain of the function mention any channel, so the fact that the function has a channel in its closure (and that it performs some I/O) is only apparent from the priorities on the arrow;
- the function $\texttt{fun}\ x\ x\texttt{!}(\texttt{recv}\ a)$ has type $\texttt{![int]}^{n+1} \to^{n,n+1} \texttt{unit}$ in the environment $a : \texttt{![int]}^n$; it contains channel a with priority n in its closure and performs input/output operations on channels with priority $n+1$ (or higher) when applied.

Rule [T-APP] deals with applications $e_1 e_2$. In the premises, τ_i is the effect caused by the evaluation of e_i. As expected, e_1 must result in a function of type $t \to^{\rho,\sigma} s$ and e_2 in a value of type t. The evaluation of e_1 and e_2 may however involve potentially blocking I/O operations on channels, and the two remaining premises make sure that no deadlock can arise. To better understand them, keep in mind that we work with a *call-by-value* language and that applications, such as $e_1 e_2$, are evaluated *sequentially from left to right*. Now, the premise $\tau_1 < |\Gamma_2|$ makes sure that any I/O operation performed during the evaluation of e_1 involves only channels with higher priority than those occurring free in e_2 (the free channels of e_2 must necessarily occur in Γ_2). This is enough to guarantee that the functional part of the application can be fully evaluated without blocking on operations concerning channels that occur *later* in the program. In principle, this premise should be paired with the symmetric one $\tau_2 < |\Gamma_1|$ making sure that any I/O operation performed during the evaluation of the argument does not involve channels that occur in the functional part. However, while the argument is being evaluated, we know that the functional part has already become a value (see the definition of reduction contexts in Section 2). Therefore, the only really critical condition to check is that no channels involved in I/O operations during the evaluation of e_2 occur in the *value* of e_1. This is expressed by the condition $\tau_2 < \rho$, where ρ is the priority of the functional part. Note that when e_1 is an abstraction, by rule [T-FUN] ρ coincides with $|\Gamma_1|$, but in general ρ may be lower than Γ_1, so the premise $\tau_2 < \rho$ gives better accuracy. The effect of the whole application $e_1 e_2$ is, as expected, the combination of the effects of evaluating e_1, e_2, and the latent effect of the function being applied. In our case the "combination" is the lowest priority of any channel involved in the application. Below are some examples:

- the application $(\texttt{fun}\ x\ x)\ 3$ is well typed, because both $\texttt{fun}\ x\ x$ and 3 are pure expressions whose effect is $\bot$, hence the two rightmost premises of [T-APP] are trivially satisfied;
- the application $(\texttt{fun}\ x\ x)\ a$ is also well typed in the environment $a : p[t]^n$, because the functional part has no effect ($\bot$);
- the application $(\texttt{fun}\ x\ x)\ (\texttt{recv}\ a)$ is well typed in the environment $a : \texttt{?[int]}^n$: the effect of $(\texttt{recv}\ a)$ is n (the priority of a) which is higher than the priority $\top$ of the function;
- the application $(\texttt{fun}\ x\ (x\texttt{,}\ a))\ (\texttt{recv}\ a)$ is ill typed because the effect of evaluating the argument is the same as the priority of the function. Indeed, for the evaluation of $\texttt{recv}\ a$ to complete it is necessary that a with output polarity is owned by a thread running in parallel with respect to the application;
- the application $(\texttt{recv}\ a)\ (\texttt{recv}\ b)$ is well typed in the environment $a : \texttt{?[int} \to \texttt{int]}^0, b : \texttt{?[int]}^1$. Note in particular that the effect of the argument is 1, which is *not* higher than the priority of the environment $a : \texttt{?[int} \to \texttt{int]}^0$ used for typing the function. However, 1 is higher than $\top$, which is the priority of the *result* of the evaluation of the functional part of the application. Therefore, this application would be rejected had we used the premise $\tau_2 < |\Gamma_1|$ in [T-APP].

Rule [T-PAIR] has many similarities with [T-APP], so we omit a detailed explanation (we see it at work in Example 3.2 below). Just note the premises making sure that the sequential evaluation of the two components of the pair do not block on channels involved in both of them, and the combination of the effects in the conclusion of the rule. Rule [T-SPLIT] is also a standard pair-splitting rule. There is only one premise $\rho < |\Gamma_2|$ checking that no channels used for evaluating the pair also occur in the body e_2 of the $\texttt{let}$.

The typing rules for processes are unremarkable: [T-PAR] splits contexts for typing the processes in parallel, [T-NEW] introduces a new channel in the environment, and [T-THREAD] types threads. Note that the effect of threads is ignored: effects are meant to capture

circular dependencies between communications and the conditions concerning priorities in the typing rules detect dependencies that arise within sequential parts of the program (*i.e.*, within expressions); circular dependencies that arise between parallel threads are indirectly detected by the fact that each occurrence of a channel is typed with the same priority (see the discussion of (1) in Section 1).

Example 3.2. In this example we see the combination of effect masking and the asymmetric conditions on priorities in rule [T-PAIR]. In particular, we can derive

$$\frac{\begin{array}{c} a : \texttt{![int]}^0 \vdash \texttt{fork} \cdots : \texttt{unit} \,\&\, \bot \\ a : \texttt{?[int]}^0 \vdash \texttt{recv}\ a : \texttt{int} \,\&\, 0 \end{array}}{a : \texttt{\#[int]}^0 \vdash (\texttt{fork}\ \texttt{fun}\ _\ a\texttt{!}3,\ \texttt{recv}\ a) : \texttt{unit} \times \texttt{int} \,\&\, 0}$$

where the same channel a occurs and is used in both components of a pair. The effect of the first component is masked by `fork` and that of the second component satisfies $0 < |\texttt{unit}| = \top$. ■

Example 3.3. Considering again `fibo` in Example 2.1, and assuming that the infix operator + is just the application of the constant + with $\mathsf{TypeOf}(+) = \texttt{int} \to \texttt{int} \to \texttt{int}$, we derive

$$\frac{\vdash + : \mathsf{TypeOf}(+) \,\&\, \bot \qquad \dfrac{\vdots}{a : \texttt{?[int]}^m \vdash \texttt{recv}\ a : \texttt{int} \,\&\, m}}{a : \texttt{?[int]}^m \vdash + (\texttt{recv}\ a) : \texttt{int} \to \texttt{int} \,\&\, m}$$

for the left operand and

$$\frac{\vdots}{b : \texttt{?[int]}^n \vdash \texttt{recv}\ b : \texttt{int} \,\&\, n}$$

for the right operant, therefore we conclude

$$a : \texttt{?[int]}^m, b : \texttt{?[int]}^n \vdash + (\texttt{recv}\ a)\ (\texttt{recv}\ b) : \texttt{int} \,\&\, n$$

provided that $m < n$, which is consistent with the fact that the receive operation on a blocks the receive operation on b. In conclusion, `fibo` is well typed and has type $\texttt{int} \to^{\top,n} \texttt{int}$. ■

3.2 Polymorphic Types

In this section we show how to enrich the type system with support for polymorphism. Ideally we would just introduce *type variables* indicating unknown types so that, for example, the identity function $\texttt{fun}\ x\ x$ can be given the familiar polymorphic type $\forall\alpha.\alpha \to \alpha$. The identity function uses its argument linearly, so the type variable α can be instantiated with *any* type, be it linear or unlimited. In general, however, one must be more careful. Consider, for example, the first projection function

$$f \stackrel{\text{def}}{=} \texttt{fun}\ x\ \texttt{fun}\ _\ x$$

which discards its second argument. Normally a function like this is assigned the type $\forall\alpha\beta.\alpha \to \beta \to \alpha$. In our context, this type is too imprecise, for two reasons. First of all, we would like to specify that β can only be instantiated with unlimited types, for the second argument of the function is discarded and so it cannot be, for instance, a channel. Second, we must be able to say that the priority of the second arrow type is the same as the priority of the first argument. Indeed, the partial application $(f\ a)$ is a function that contains a channel a. Not only this function is linear, but its priority coincides with that of a. In summary, we must be able to reason on, and impose constraints to, the priority of type variables. To this aim, we introduce a category of *priority variables* ranging over unknown priorities, and we allow quantification over priority, as well as type, variables, as we have already suggested in Section 1. In both cases quantification is bounded: type variables range over types with a specific priority, and priority variables range over priority intervals. Below is the extension of types to polymorphism:

type	t	$::=$	$\cdots \mid \alpha$
type scheme	T	$::=$	$t \mid \forall\alpha^\rho.T \mid \forall\imath^\varphi.T$
priority	ρ	$::=$	$\bot \mid n \mid \top \mid \imath \mid \rho \wedge \rho \mid \rho \vee \rho$
interval	φ	$::=$	$[\rho,\rho] \mid (\rho,\rho] \mid [\rho,\rho] \mid (\rho,\rho)$

Types are enriched with *type variables* $\alpha, \beta, \ldots$; type schemes are possibly quantified types: quantification over type variables specifies the priority of types that can instantiate the type variable, while quantification over *priority variables* $\imath, \jmath, \ldots$ specifies an interval φ of priorities over which the priority variable can be instantiated. We use established notation for denoting *intervals* φ, $\ldots$ of $\mathbb{P}$; for example, the closed interval $[\bot, \top]$ denotes the whole $\mathbb{P}$ and the open interval $(\bot, \top)$ denotes the set of finite priorities $\mathbb{Z}$. Having introduced priority variables, we generalize priorities to *priority expressions* and we keep using $\rho, \ldots$ for ranging over these; the operators $\wedge$ and $\vee$ now become part of the syntax.

Using polymorphic types we can assign proper type schemes to the four primitives of our language:

$$\begin{array}{l} \texttt{open} : \forall\imath.\forall\alpha^\imath.\forall\jmath^{(\bot,\imath)}.\texttt{unit} \to \texttt{\#[}\alpha\texttt{]}^\jmath \\ \texttt{recv} : \forall\imath.\forall\alpha^\imath.\forall\jmath^{(\bot,\imath)}.\texttt{?[}\alpha\texttt{]}^\jmath \to^{\top,\jmath} \alpha \\ \texttt{send} : \forall\imath.\forall\alpha^\imath.\forall\jmath^{(\bot,\imath)}.\texttt{![}\alpha\texttt{]}^\jmath \to \alpha \to^{\jmath,\jmath} \texttt{unit} \\ \texttt{fork} : \forall\imath.\forall\jmath^{[\bot,\top]}.(\texttt{unit} \to^{\imath,\jmath} \texttt{unit}) \to \texttt{unit} \end{array}$$

where we omit $(\bot, \top]$ intervals. Note that we can specify the constraint requiring messages to have lower priority with respect to the channel on which they travel.

Since now types may contain type variables and priority expressions may contain priority variables, we must parametrize operations on types and relations between priority expressions by a *priority environment* that keeps track of the priority of type variables and of the intervals on which priority variables range over. *Priority environments* $\Theta, \ldots$ are defined by:

$$\Theta \ ::= \ \emptyset \mid \Theta, \imath \in \varphi \mid \Theta, \alpha :: \rho$$

We write $\mathsf{dom}(\Theta)$ for the domain of Θ, namely the set of type and priority variables for which there is an association in Θ, and $\Theta(\alpha)$ for the priority of α as determined by Θ. We also write Θ_1, Θ_2 for the union of Θ_1 and Θ_2 when they have disjoint domains. To make sure that the associations in priority environments are not ambiguous, we do *not* equate Θ_1, Θ_2 and Θ_2, Θ_1 and we require two basic well-formedness conditions: $\Theta, \imath \in \varphi$ is well formed if all the priority variables in φ are in $\mathsf{dom}(\Theta)$, and $\Theta, \alpha :: \rho$ is well formed if all the priority variables in ρ are in $\mathsf{dom}(\Theta)$. In the following we implicitly assume to work with well-formed priority environments. We write $|t|_\Theta$ for the expression that denotes the priority of t in the priority environment Θ. Formally:

$$|t|_\Theta \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } t = \texttt{B} \\ \Theta(\alpha) & \text{if } t = \alpha \in \mathsf{dom}(\Theta) \\ \rho & \text{if } t = p\texttt{[}s\texttt{]}^\rho \text{ or } t = t_1 \to^{\rho,\sigma} t_2 \\ |t_1|_\Theta \wedge |t_2|_\Theta & \text{if } t = t_1 \times t_2 \end{cases}$$

When checking a relation between priority expressions $\rho\ \mathscr{R}\ \sigma$ or a membership $\rho \in \varphi$, knowing the range of the priority variables occurring in ρ and σ is relevant. For example, the relation $\imath < \jmath$ does not hold for arbitrary values of $\imath$ and $\jmath$ but it holds, say, under the assumptions given by the priority environment $\imath \in (\bot, \top), \jmath \in (\imath, \top]$. From now on we will write $\rho\ \mathscr{R}_\Theta\ \sigma$ if the relation $\mathscr{R}$ holds under the assumptions Θ; similarly, we will write $\rho \in_\Theta \varphi$. We do not provide any detail regarding the inference engine that derives judgments $\rho\ \mathscr{R}_\Theta\ \sigma$ or $\rho \in_\Theta \varphi$, we only assume that the inference

engine is closed by substitutions. That is, if $\rho \; \mathscr{R}_{\Theta,\imath\in\varphi,\Theta'} \; \sigma$ and $\tau \in_\Theta \varphi$, then $\rho\{\tau/\imath\} \; \mathscr{R}_{\Theta,\Theta'\{\tau/\imath\}} \; \sigma\{\tau/\imath\}$.

Following [35], we define two relations for respectively *instantiating* a type scheme to a type and *generalizing* a type to a type scheme. Since our form of quantification is bounded, both relations must be parametric over a priority environment which makes sure that instances of type/priority variables are appropriate (in the case of instantiation) and which keeps track of the type/priority variables that have been generalized (in the case of generalization). We have:

$$t \succ_\Theta t \qquad \frac{|s| =_\Theta \rho \qquad T\{s/\alpha\} \succ_\Theta t}{\forall\alpha^\rho.T \succ_\Theta t} \qquad \frac{\rho \in_\Theta \varphi \qquad T\{\rho/\imath\} \succ_\Theta t}{\forall\imath^\varphi.T \succ_\Theta t}$$

for *instantiation* and

$$t \prec_\emptyset t \qquad \frac{|t| =_\emptyset \top \qquad t \prec_\Theta T}{t \prec_{\alpha::\rho,\Theta} \forall\alpha^\rho.T} \qquad \frac{|t| =_\emptyset \top \qquad t \prec_\Theta T}{t \prec_{\imath\in\varphi,\Theta} \forall\imath^\varphi.T}$$

for *generalization*. Note that we allow the generalization of a type t only if it can be established that its priority is $\top$ in the empty priority environment. The motivation for this constraint is to prevent the priority of a quantified type to depend on the type/priority variable being quantified. For example, we forbid the generalization $\texttt{int} \to^{\imath,\bot} \texttt{int} \prec_{\imath\in(\bot,\top]} \forall\imath.(\texttt{int} \to^{\imath,\bot} \texttt{int})$ because the priority of $\texttt{int} \to^{\imath,\bot} \texttt{int}$ depends on $\imath$. We argue that the impact of such restriction is null: types with a finite (non-$\top$) priority are linear and denote values that can be used only once, so there is no much point in making them polymorphic. On the contrary, thanks to this restriction we can easily generalize the computation of the priority of polymorphic types, which is trivially $\top$. Formally:

$$|\forall\alpha^\rho.T|_\Theta = |\forall\imath^\varphi.T|_\Theta \stackrel{\text{def}}{=} \top$$

Now that we have introduced polymorphic and extended the notion of priority to them, the last ingredient we need is a revised combination operator $+$. In practice the definition is the same as that in equation 4, except that the operator is parametric over a priority environment Θ which is used for establishing whether a type (scheme) T is unlimited or not. More precisely, we revise the first equation in (4) thus:

$$T +_\Theta T \stackrel{\text{def}}{=} T \qquad \text{if } |T|_\Theta =_\Theta \top$$

The extension of $+$ to type environments is just as before, recalling that type environments now associate type schemes to names. In particular:

$$(\Gamma, u : T) +_\Theta (\Gamma', u : S) \stackrel{\text{def}}{=} (\Gamma +_\Theta \Gamma'), u : T +_\Theta S$$

We are now ready to discuss the typing rules of the polymorphic type system. Most of them have exactly the same structure as in the monomorphic type system, except that now judgments for expressions have the form $\Gamma \vdash_\Theta e : t \,\&\, \rho$ and the relations concerning priorities are parametric in the Θ environment. The rules affected by polymorphism are shown in Table 3 (and the complete revised type system is in Appendix A). A judgment $\Gamma \vdash_\Theta e : t \,\&\, \rho$ is well formed if all the free type/priority variables occurring in Γ are in $\mathsf{dom}(\Theta)$. From now on we implicitly assume well formedness of all judgments. Rules [T-NAME POLY] and [T-CONST POLY] are standard: they instantiate the (polymorphic) types of constants and of names in the type environment. Rule [T-LET POLY] implements `let`-polymorphism: the type t of the expression being bound to the variable x can be generalized to the type scheme T provided that all quantified type/priority variables (in Θ') do not occur in the type environment Γ_1. This is guaranteed by the fact that Θ' is disjoint from Θ and that Γ_1 must necessarily contain only type/priority variables in Θ for otherwise the judgment in the conclusion would be ill formed. We also need to revise rule [T-REC] to [T-REC POLY] to support *polymorphic recursion*. The idea is to generalize the type of a recursion variable x and allow each occurrence of a $\texttt{rec}\ x\ e$ to be instantiated to a type s that is appropriate for the context where the recursion variable occurs. Polymorphic recursion is essential for typing most of the examples that follow.

Example 3.4. Given the usual definition of function composition

```
let (∘) f g = fun x (f (g x))
```

we can derive

$$\begin{aligned}(\circ) : \; & \forall\alpha\beta\gamma.\forall\imath.\forall\jmath^{[\bot,\top]}.\forall\kappa.\forall h^{[\bot,\top]}. \\ & (\beta \to^{\imath,\jmath} \gamma) \to (\alpha \to^{\kappa,h} \beta) \to^{\imath,\bot} (\alpha \to^{\imath\wedge\kappa,\jmath\vee h} \gamma)\end{aligned}$$

(type variables with no priority bounds are assumed to range over any type). We can now define a polymorphic forwarder as follows

```
let forward x = (send x) ∘ recv
```

for which we have

$$\texttt{forward} : \forall\imath.\forall\jmath^{(\imath,\top)}.\forall\kappa^{(\jmath,\top)}.\forall\alpha^\kappa.\texttt{?}[\alpha]^\imath \to \texttt{!}[\alpha]^\jmath \to^{\imath,\jmath} \texttt{unit}$$

Note that the priority of the channel on which the message is forwarded must be lower than the priority of the channel from which the message is received, and that the message itself must have lower priority than both channels. ■

Example 3.5. Considering again the functions `ssend` and `asend` from Example 2.2, we derive

$$\begin{aligned}\texttt{ssend} : \; & \forall\imath.\forall\jmath^{(\imath,\top)}.\forall\kappa^{(\imath,\top]}.\forall h^{(\jmath,\top]}.\forall\alpha^\kappa.\forall\beta^h. \\ & \texttt{![}\alpha \times \texttt{?[}\beta\texttt{]}^\jmath\texttt{]}^\imath \to \alpha \to^{\imath,\imath} \texttt{![}\beta\texttt{]}^\jmath \\ \texttt{asend} : \; & \forall\imath.\forall\jmath^{(\imath,\top)}.\forall\kappa^{(\imath,\top]}.\forall h^{(\jmath,\top]}.\forall\alpha^\kappa.\forall\beta^h. \\ & \texttt{![}\alpha \times \texttt{?[}\beta\texttt{]}^\jmath\texttt{]}^\imath \to \alpha \to^{\imath,\bot} \texttt{![}\beta\texttt{]}^\jmath\end{aligned}$$

The only difference between the two types is that `asend` has no latent effect, since the output is performed in an independent thread that is spawned from within `asend`. Note also that there is nothing in the definition of these functions imposing that the returned continuation must have output capability. For instance, `ssend` has also the type $\texttt{![}\alpha \times \texttt{![}\beta\texttt{]}^\jmath\texttt{]}^\imath \to \alpha \to^{\imath,\imath} \texttt{?[}\beta\texttt{]}^\jmath$. As a consequence, some expressions lack a principal typing. ■

3.3 Recursive Types

Let us complete the type system with support for recursive types. This is achieved by adding a conventional $\mu\alpha$ type constructor, but to make recursive types useful in our setting it is necessary to extend types in a slightly more sophisticated way. To see why, consider for example the `link` combinator that we have defined in Example 2.4 and that forwards all messages received from a stream x to another stream y. Since both x and y are *infinite* streams, it is reasonable to expect that their respective types will be recursive. Tentatively, we may think of the assignments

$$x : t \qquad \text{and} \qquad y : \texttt{![int} \times s\texttt{]}^1$$

where t and s are recursive types satisfying the equations:

$$t = \texttt{?[int} \times t\texttt{]}^0 \qquad s = \texttt{?[int} \times s\texttt{]}^2 \tag{5}$$

(for simplicity, while discussing this example we fix both the type `int` of messages and the priorities in types).

Note that each message sent on the output stream y is a pair made of an integer and a continuation channel with *input* capability, since that continuation will be used for receiving the *next* message sent on the stream. Note also that the priority of x must be strictly higher than that of y, since in `link` there is an input operation on x that blocks an output operation on y. The problem is that, by looking at the type schemes associated with `send` and `recv`, we see that any message sent on a channel must have a type with a priority that is strictly lower than that for the message, and this is not true

$$\frac{\top \leq_\Theta |\Gamma| \qquad T \succ_\Theta t}{\Gamma, u : T \vdash_\Theta u : t \,\&\, \bot}\ \text{[T-NAME POLY]} \qquad \frac{\Gamma_1 \vdash_{\Theta,\Theta'} e_1 : t \,\&\, \bot \qquad t \prec_{\Theta'} T \qquad \Gamma_2, x : T \vdash_\Theta e_2 : s \,\&\, \rho}{\Gamma_1 +_\Theta \Gamma_2 \vdash_\Theta \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : s \,\&\, \rho}\ \text{[T-LET POLY]}$$

$$\frac{\top \leq_\Theta |\Gamma| \qquad \mathsf{TypeOf}(\texttt{c}) \succ_\Theta t}{\Gamma \vdash_\Theta \texttt{c} : t \,\&\, \bot}\ \text{[T-CONST POLY]} \qquad \frac{\top \leq_\Theta |\Gamma| \qquad \Gamma, x : T \vdash_{\Theta,\Theta'} e : t \,\&\, \bot \qquad T \succ_\Theta s}{\Gamma \vdash_\Theta \texttt{rec}\ x\ e : s \,\&\, \bot}\ \text{[T-REC POLY]}$$

Table 3. Rules for polymorphism and polymorphic recursion.

for t and s. For example, a message received from a channel with type t has priority $|\texttt{int} \times t| = |t| = 0$, which is the same priority of x. Overall, in order to satisfy the constraint on the priority of message types in the first and every subsequent iteration of `link`, we must use the assignments

$$x : t^{(0)} \qquad \text{and} \qquad y : \texttt{![int} \times t^{(2)}\texttt{]}^1$$

where $t^{(i)}$ satisfies the equation

$$t^{(i)} = \texttt{?[int} \times t^{(i+1)}\texttt{]}^i \tag{6}$$

The problem is that $t^{(i)}$ consists of infinitely many nested channel types with numerically increasing priorities. In other words, $t^{(i)}$ is not a regular type, so it cannot be finitely represented by means of the familiar μ notation [8].

To recover a finite representation of the types $t^{(i)}$ we resort to a new type constructor $\$^\rho t$ which allows us to express the priorities in t as being relative to a finite displacement ρ. The value of a priority σ occurring in t is actually $\rho + \sigma$, where $+$ is the extension of integer addition to priorities, so that $\bot + n = \bot$ and $\top + n = \top$ for every $n \in \mathbb{Z}$. In summary, we extend the syntax of types and priorities in this way:

$$\begin{array}{rrcl} \textbf{type} & t & ::= & \cdots \mid \mu\alpha.t \mid \$^\rho t \\ \textbf{priority} & \rho & ::= & \cdots \mid \rho + \rho \end{array}$$

where $\mu\alpha$ is the standard binder for recursion type variables. Then, in order to make recursions and displacements transparent constructors, we identify types according to the following set of equations:

$$\begin{array}{rcl} \mu\alpha.t & = & t\{\mu\alpha.t/\alpha\} \\ \$^\rho \texttt{B} & = & \texttt{B} \\ \$^\rho(t \times s) & = & (\$^\rho t) \times (\$^\rho s) \\ \$^\rho(t \to^{\sigma,\tau} s) & = & (\$^\rho t) \to^{\rho+\sigma,\rho+\tau} (\$^\rho s) \\ \$^\rho p\texttt{[}t\texttt{]}^\sigma & = & p\texttt{[}\$^\rho t\texttt{]}^{\rho+\sigma} \end{array}$$

The first equation is the standard identification of types modulo folding/unfolding of recursions. The subsequent equations propagate the $\$^\rho$ constructor across compound types and add ρ to every explicit priority annotation. For example, we have $\$^1(\texttt{int} \times p\texttt{[int]}^2) = (\$^1\texttt{int}) \times (\$^1 p\texttt{[int]}^2) = \texttt{int} \times p\texttt{[}\$^1\texttt{int]}^3 = \texttt{int} \times p\texttt{[int]}^3$. Going back to the equation (6), we can now define

$$t \stackrel{\text{def}}{=} \mu\alpha.\texttt{?[int} \times \$^1\alpha\texttt{]}^0$$

and it is easy to see that

$$\begin{array}{rcl} t & = & \texttt{?[int} \times \$^1 t\texttt{]}^0 \\ & = & \texttt{?[int} \times \$^1\texttt{?[int} \times \$^1 t\texttt{]}^0\texttt{]}^0 \\ & = & \texttt{?[int} \times \texttt{?[int} \times \$^2 t\texttt{]}^1\texttt{]}^0 \\ & = & \cdots \end{array}$$

namely that $\$^i t$ is a finite representation for $t^{(i)}$.

Example 3.6. Let us define the type $t\ p\ \texttt{stream}^{\rho,\tau}$ for representing infinite p-streams of messages of type t:

$$t\ p\ \texttt{stream}^{\rho,\sigma} \stackrel{\text{def}}{=} p\texttt{[}t \times \mu\alpha.\texttt{?[}t \times \$^\sigma\alpha\texttt{]}^{\rho+\sigma}\texttt{]}^\rho$$

Each time a message is received from (respectively, sent to) an ?-stream (respectively, a `!`-stream), the priority σ of the stream is incremented by ρ. In particular, we have $t\ \texttt{?stream}^{\rho,\sigma} = \texttt{?[}t \times t\ \texttt{?stream}^{\rho+\sigma,\sigma}\texttt{]}^\rho$ and $t\ \texttt{!stream}^{\rho,\sigma} = \texttt{![}t \times t\ \texttt{?stream}^{\rho+\sigma,\sigma}\texttt{]}^\rho$. The functions `ssend` and `asend` can be specialized to sending messages on streams. For example, `asend` has the type

$$\forall\alpha^\top.\forall\imath^{(\bot,\top)}.\forall\kappa^{(0,\top)}.\alpha\ \texttt{!stream}^{\imath,\kappa} \to \alpha \to^{\imath,\bot} \alpha\ \texttt{!stream}^{\imath+\kappa,\kappa}$$

and `ssend` has an analogous type with $\imath$ in place of $\bot$ in the latent effect of the rightmost arrow.

We are now ready to give a type to the `node` function in Example 2.3. We can derive:

$$\begin{array}{l} \texttt{node} : \forall\alpha.\forall\imath^{(\bot,\top)}.\forall\kappa^{(0,\top)}.\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha\ \texttt{!stream}^{\imath,\kappa} \\ \qquad\qquad \to \alpha\ \texttt{?stream}^{\imath,\kappa} \to^{\imath,\top} \texttt{unit} \end{array}$$

There are two important observations regarding the typing of `node`: first, the local variables x' and y' in the body of `node` are assigned the types

$$x' : \alpha\ \texttt{!stream}^{\imath+\kappa,\kappa} \qquad \text{and} \qquad y' : \alpha\ \texttt{?stream}^{\imath+\kappa,\kappa}$$

which differ from the corresponding types of x and y because of the priorities of streams change each time a message is sent/received. Then, the recursive call of `node` is well typed thanks to polymorphic recursion. The second observation concerns the latent effect of `node`, which cannot be finite (or $\bot$) because `node` performs an unbounded sequence of input operations on channels with (numerically) strictly increasing priorities. Then, the saturated application of `node` yields an expression that has effect $\top$. Because of the constraints on priorities in the typing rules (Table 2) an expression with effect $\top$ cannot be "followed" by any other expression: it cannot be neither the functional part nor the argument of an application (see [T-APP]), it cannot occur within pairs (see [T-PAIR]), nor can it occur in the left part of a `let` (see [T-SPLIT] and [T-LET]). In other words, it can only occur in *tail position*. This is precisely what happens with `node` in Example 2.3. ■

Example 3.7. Table 4 presents the types of the stream combinators used in Example 2.4. All of them take advantage of polymorphic recursion and are non-terminating, tail-recursive functions with latent effect $\top$ just like `node` discussed in Example 3.6. Now we can derive, for example

$$\texttt{mk_fibo_network} : \texttt{unit} \to \texttt{int}\ \texttt{?stream}^{2,5}$$

with the following assignments for the bound names in the body of `mk_fibo_network`:

$$\begin{array}{rl} c_i : \texttt{int}\ \texttt{\#stream}^{i-1,5} & \text{for } 1 \leq i \leq 4 \\ c_5 : \texttt{int}\ \texttt{\#stream}^{9,5} & \\ c_1' : \texttt{int}\ \texttt{!stream}^{10,5} & \\ out : \texttt{int}\ \texttt{\#stream}^{2,5} & \end{array}$$

Note that `mk_fibo_network` itself is a pure function, since all the computation (and communication) is performed in threads spawned within the body of the function. ■

$$\begin{aligned}
\texttt{link} &: \forall\alpha.\forall\imath.\forall\jmath^{(\imath,\top)}.\forall\kappa^{(0,\top)}.\alpha\ \texttt{?stream}^{\imath,\kappa} \to \alpha\ \texttt{!stream}^{\jmath,\kappa} \to^{\imath,\top} \texttt{unit}\\
\texttt{delay} &: \forall\alpha.\forall\imath.\forall\jmath^{(\imath,\top)}.\forall\kappa^{(0,\top)}.\alpha\ \texttt{?stream}^{\imath,\kappa} \to \alpha\ \texttt{!stream}^{\jmath+\kappa,\kappa} \to^{\imath,\top} \texttt{unit}\\
\texttt{copy} &: \forall\alpha.\forall\imath.\forall\jmath^{(\imath,\top)}.\forall h^{(\jmath,\top)}.\forall\kappa^{(0,\top)}.\alpha\ \texttt{?stream}^{\imath,\kappa} \to \alpha\ \texttt{?stream}^{\jmath,\kappa} \to^{\imath,\bot} \alpha\ \texttt{!stream}^{h,\kappa} \to^{\imath,\top} \texttt{unit}\\
\texttt{add} &: \forall\imath.\forall\jmath^{(\imath,\top)}.\forall h^{(\jmath,\top)}.\forall\kappa^{(0,\top)}.\texttt{int}\ \texttt{?stream}^{\imath,\kappa} \to \texttt{int}\ \texttt{?stream}^{\jmath,\kappa} \to^{\imath,\bot} \texttt{int}\ \texttt{!stream}^{h,\kappa} \to^{\imath,\top} \texttt{unit}
\end{aligned}$$

Table 4. Typing of the stream combinators in Example 2.4.

3.4 Properties

The type system is a conservative extension of the Hindley-Milner type system: every program that does not make use of the communication primitives and that is well typed in the Hindley-Milner type system is also well typed in our type system. This follows from the observation that the effect of pure expressions is $\bot$ and unlimited types have priority $\top$ by definition, therefore all the conditions concerning priorities in the typing rules are trivially satisfied.

Let us now discuss the properties of well-typed programs. The first, expected property is subject reduction, namely the fact that the reduction of expressions/processes preserves well typedness. For expressions, this is the familiar preservation of types. For processes, we must take into account the fact that linear channels are *consumed* after each synchronization (last but one reduction in Table 1). This means that when a process P performs a communication on some channel a, the channel must disappear from the type environment used for typing the process. To this aim, we define a partial operation $\Gamma - \ell$ that removes ℓ from Γ, when ℓ is a channel:

$$\Gamma - \tau \stackrel{\text{def}}{=} \Gamma \qquad\qquad (\Gamma, a : \#[t]^{\rho}) - a \stackrel{\text{def}}{=} \Gamma$$

Now, we can formulate subject reduction:

Theorem 3.8. *The following properties hold:*

1. *If* $\Gamma \vdash_{\Theta} e : t \,\&\, \rho$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash_{\Theta} e' : t \,\&\, \rho$*.*
2. *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\ell} P'$*, then* $\Gamma - \ell \vdash P'$*.*

The fact that the type environment changes because of reductions is a common trait of *behavioral type systems*, such as those based on session types [9–12]. Linear channel types are a basic form of behavioral types. There are two things to remark regarding Theorem 3.8. The first one is that the effect ρ of an expression e does not change when e reduces. This is expected since the reductions of expressions in Table 1 solely concern the *pure* fragment of the calculus not involving communication primitives. The second observation is that $\Gamma - a$ is not defined if $a \notin \mathsf{dom}(\Gamma)$. This means that Theorem 3.8 is slightly more than an auxiliary result for proving the soundness of the type system (Theorem 3.11 below). It means that well-typed programs never attempt at using the same channel twice, namely that channels in well-typed programs are indeed *linear channels*. This property has important practical consequences, since it allows the efficient implementation (and deallocation) of channels [20, 32].

Another property granted by linear communications is that computations are deterministic, despite threads may interleave actions in different ways. We express this property as strong confluence of well-typed programs:

Theorem 3.9 (strong confluence)**.** *Let* $\Gamma \vdash P$ *and* $P \xrightarrow{\ell_1} P_1$ *and* $P \xrightarrow{\ell_2} P_2$*. Then either* $P_1 \equiv P_2$ *or there exist* Q *such that* $P_1 \xrightarrow{\ell_2} Q$ *and* $P_2 \xrightarrow{\ell_1} Q$*.*

Theorems 3.8 and 3.9 just reformulate expected or known results [20] concerning the linear π-calculus in the context of a functional language. We now turn the attention to the deadlock freedom properties granted by our typing discipline. Deadlock freedom roughly corresponds to the property that *if* the program terminates, then it has no pending I/O operations left. In our case, the only terminated program without pending operations is (structurally equivalent to) $\{()\}$. We can therefore define deadlock freedom thus:

Definition 3.10 (deadlock freedom [18])**.** We say that P is *deadlock free* if $P \xrightarrow{\tau}{}^{*} Q \not\longrightarrow$ implies $Q \equiv \{()\}$.

The notation $Q \not\longrightarrow$ means, as usual, that Q is unable to reduce further.

Now, every well-typed, closed process is free from deadlocks:

Theorem 3.11 (soundness)**.** *Let* $\emptyset \vdash P$*. Then* P *is deadlock free.*

Theorem 3.11 can be generalized to processes that are well typed in a *balanced* environment Γ, where Γ is balanced if its domain contains only channels with # polarity. The balancing condition, which is trivially satisfied by the empty context $\emptyset$ in the statement of Theorem 3.11, ensures that the program is "complete", in the sense that every linear channel is used for both an input *and* an output operation. This is essential to guarantee that the program does not get stuck on a pending I/O operation which has no complementary one. Nonetheless, one may think that the statement of Theorem 3.11 is rather weak, considering that every process P (even an ill-typed one) can be "fixed" and become part of a deadlock-free system if composed in parallel with the diverging thread $\{\texttt{rec}\ x\ x\}$.

It is not easy to state an interesting property of well-typed *partial programs* – programs that are well-typed in non-balanced environments – or of *partial computations* – computations that have not reached a stable (*i.e.*, irreducible) state. We might think that well-typed programs eventually use all of their channels. This property is false in general, for two reasons. First, our type system does not guarantee the termination of well-typed expressions, so for example a thread like $\{a\texttt{!}(\texttt{rec}\ x\ x)\}$ never uses channel a, because the evaluation of the message does not terminate. Second, there are threads that repeatedly generate (or receive) new channels, so that the set of channels they own is never empty. This happens for instance in Example 2.3 and Example 2.4. What we can prove is that, *assuming* that a well-typed program does not internally diverge, then *each* channel it owns will eventually be used for a communication or it will be sent to the environment in a message. To formalize this property we must generalize reductions to labeled transitions, so that we can reason on the evolution of partial programs. *Labels* $\lambda, \ldots$ of transitions are defined by

$$\lambda \ ::= \ \ell \ \mid\ a?e \ \mid\ a\texttt{!}\mathsf{v}$$

and the transition relation $\xmapsto{\lambda}$ extends reduction with the rules

$$\frac{a \notin \mathsf{bn}(\mathscr{C})}{\mathscr{C}[a\texttt{!}\mathsf{v}] \xmapsto{a\texttt{!}\mathsf{v}} \mathscr{C}[()]} \qquad \frac{a \notin \mathsf{bn}(\mathscr{C}) \qquad \mathsf{fn}(e) \cap \mathsf{bn}(\mathscr{C}) = \emptyset}{\mathscr{C}[\texttt{recv}\ a] \xmapsto{a?e} \mathscr{C}[e]}$$

where $\mathscr{C}$ ranges over *process contexts* $\mathscr{C} ::= \{\mathscr{E}\} \mid (\mathscr{C} \mid P) \mid (P \mid \mathscr{C}) \mid \texttt{new}\ a\ \texttt{in}\ \mathscr{C}$. Messages of input transitions have the form $a?e$ where e is an arbitrary expression instead of a value. This is just to allow a technically convenient formulation of Definition 3.12 below. We formalize the assumption concerning the absence of internal divergences as a property that we call *interactivity*. Note that interactivity is a property of *typed processes*, which we write as pairs $\Gamma \fatsemi P$, since the messages exchanged between a process and the environment in which it executes are not arbitrary in general.

Definition 3.12 (interactivity)**.** Interactivity is the largest predicate on typed processes such that $\Gamma \fatsemi P$ *interactive* implies $\Gamma \vdash P$ and:

1. P has no infinite reduction $P \stackrel{\ell_1}{\longmapsto} P_1 \stackrel{\ell_2}{\longmapsto} P_2 \stackrel{\ell_3}{\longmapsto} \cdots$, and
2. if $P \stackrel{\ell}{\longmapsto} Q$, then $\Gamma - \ell \fatsemi Q$ is interactive, and
3. if $P \stackrel{a!\mathsf{v}}{\longmapsto} Q$ and $\Gamma = \Gamma', a : \mathtt{!}[t]^n$, then $\Gamma'' \fatsemi Q$ is interactive for some $\Gamma'' \subseteq \Gamma'$, and
4. if $P \stackrel{a?x}{\longmapsto} Q$ and $\Gamma = \Gamma', a : \mathtt{?}[t]^n$, then $\Gamma'' \fatsemi Q\{\mathsf{v}/x\}$ is interactive for some v and $\Gamma'' \supseteq \Gamma'$ such that $n < |\Gamma'' \setminus \Gamma'|$.

Clause (1) says that an interactive process does not internally diverge, so it will eventually halt either because it terminates or because it needs interaction with the environment in which it executes. Clause (2) simply states that reductions preserve interactivity. Clause (3) states that a process with a pending output on a channel a *must* reduce to an interactive process after the output is performed. Finally, clause (4) states that a process with a pending input on a channel a *may* reduce to an interactive process after the input of a particular message v is performed. Note that clauses (2) and (3) state provable properties of well-typed processes, and the condition $\Gamma'' \supseteq \Gamma$ where $n < |\Gamma'' \setminus \Gamma'|$ in clause (4) follows from the typing of v, which must have priority $n + 1$ or lower. Therefore, the only additional conditions we are imposing on well-typed programs are *convergence* of P in clause (1) and the *existence* of v in clause (4). It is now possible to prove that well-typed, interactive processes eventually use their channels.

Theorem 3.13 (interactivity)**.** *Let $\Gamma \fatsemi P$ be an interactive typed process such that $a \in \mathsf{fn}(P)$. Then $P \stackrel{\lambda_1}{\longmapsto} P_1 \stackrel{\lambda_2}{\longmapsto} \cdots \stackrel{\lambda_n}{\longmapsto} P_n$ for some $\lambda_1, \ldots, \lambda_n$ such that $a \notin \mathsf{fn}(P)$.*

4. Related Work

Higher-order concurrent languages. Concurrent ML [28, 29] is probably the most studied higher-order language with concurrency primitives. In addition to primitives such as `send` and `recv`, CML also considers first-class events that can be used for building I/O actions in a compositional way, in the style of monadic `IO` actions of Haskell. The approach we have described in this paper can be easily extended to CML events, by annotating event types with priorities. Effect systems for higher-order concurrent languages are discussed in [2, 24]. Effects are used for checking that the communication behavior of a program does not deviate from a desired protocol or for establishing properties (such as *finiteness*) of the communication topology of a concurrent system. Type inference algorithms are presented in [1, 2, 22]. Apart from the fact that we focus on linear communications, a major difference between these and our work is that in [1, 2, 22, 24] the structure of communications is recorded in the effects, while in our case effects simply keep track of the priority of the channels involved in communications and the protocol is encoded in types. More recently, linear typing disciplines not based on effects for higher-order languages equipped with a native notion of sessions have been investigated [5, 10, 34]. In addition to safety, types are used in [10] for estimating bounds in the size of message queues, and in [5] for detecting memory leaks occurring when channel pointers become part of unreachable, cyclic memory structures. The type system in the present paper subsumes the one in [5], for the absence of cyclic dependencies between channels is a necessary condition for deadlock freedom. Also, we simultaneously simplify and generalize the language in [10]: since binary sessions can be encoded in the linear π-calculus [9], our language subsumes that in [10] and in addition our type system treats polymorphism and guarantees deadlock freedom also in presence of session interleaving. Wadler [34] presents a session-based functional language such that well-typed programs are deadlock free without requiring any particular type annotations dedicated to this purpose. However, in his case the syntax of (well-typed) programs prevents the modeling of cyclic network topologies so that, for instance, the networks in Examples 2.3 and 2.4 are ill typed.

Deadlock freedom. Our type system has been partly inspired by previous static analysis techniques guaranteeing deadlock freedom for the π-calculus [18, 25]. There are two main differences between [18] and our own work: first, we work with a higher-order language instead of a process calculus. This requires a richer structure of types, in particular using (latent) effects, since the typing rules do not have visibility of the whole continuation of a program fragment. The second main difference regards the use of polymorphism (and particularly of polymorphic recursion) with respect to channel priorities. This feature has a significant impact on the accuracy of the type system. For example, the suitable encodings of the program in Example 2.3 or the stream network in Example 2.4 are ill typed according to the type system in [18]. In general, the lack of polymorphism in [18] prevents any process combining recursion and interleaving actions on two or more channels from being well typed. Properties stronger than deadlock freedom such as *lock freedom* – the assurance that every pending communication eventually takes place – are considered in [17, 19, 25]. Our interactivity property (Theorem 3.13) is similar in spirit to the *hybrid* type system [19] showing that lock freedom can be characterized as the conjunction of deadlock freedom and termination.

Linearity. Our language makes use of linear channels, whose properties have been previously studied in the setting of the π-calculus [20]. Many practical systems make widespread use of linear channels [13, 14]. It has been shown that structured communications such as those in binary sessions can be modeled solely using binary sessions [9] and a large fraction of *multiparty* sessions – those with an arbitrary, although usually fixed, number of participants – can be modeled using linear channels as well [25]. Concurrency models such as Kahn's process networks [15] or communicating finite-state machines [6] make use of communication channels that are exactly, or very close to, those that can be encoded using linear channels. Incidentally, deadlock detection in these systems has been shown to be undecidable, even without channel mobility and in simple network configurations [6, 7].

Various forms of linearity find widespread use also in sequential languages, where they can be used for the improvement of code optimizations [32], safe access control of shared resources [31], and refined analysis of deallocations [16]. The monadic I/O system of Haskell is a way of enforcing the linear usage of global state [33].

Non-regular types. Limited forms of non-regular types, sometimes called *nested datatypes*, have already been considered [3, 4] also for processes [27]. In all these cases, the non-regularity follows from the very structure of types, whereas in our case it is only the priority annotations that change.

5. Concluding Remarks

Type safety alone is not strong enough to entail deadlock freedom when concurrent threads interleave actions on communication channels. This observation has led to the development of type systems [17–19, 25] specifically aimed at ensuring deadlock freedom. A common trait of all these works is that they are based on the π-calculus [21]. While computationally complete, the π-calculus's only focus is on communication, and programs encoded in the π-calculus are flat sequences of I/O actions. The intrinsic structure of high-level (and possibly higher-order) programs, which include procedures, functions, objects, modules, etc., is therefore lost in the encoding, meaning that the type-based techniques devised for the π-calculus may need significant reworking when ported back to concrete languages. Inspired by the type system for deadlock free-

dom in [25], we have studied this porting to a higher-order language with communication primitives *à la* Concurrent ML [28, 29].

Our type system grants a strong form of deadlock freedom (Theorems 3.11 and 3.13) with a relatively minimal machinery, which has nonetheless a non-trivial impact when integrated with the other features of the language, most notably polymorphism and recursion. The presented type system also has some limitations, of which we discuss here the two that we consider most critical. The first obvious issue regards type reconstruction. As clearly witnessed by the examples in Section 3, the detailed nature of types makes type reconstruction almost mandatory if the type system is meant to be practically useful. We have not yet investigated whether a complete type reconstruction algorithm can be developed. We do have a prototypical reconstruction algorithm for a similar (but simpler) type system for the π-calculus [25] and we are aware of the fact that reconstruction algorithms supporting polymorphic recursion are feasible when this form of polymorphism is restricted to *effects* [1, 2], which is precisely the way we use it. More work is necessary to see whether these good-looking premises lead to a concrete reconstruction algorithm.

Another issue is that some natural program patterns are ill typed. An example is given by `filter_stream` below, which is the stream-oriented counterpart of the well-known `filter` function:

```
1  let rec filter_stream f x y =
2    let (v,x') = recv x in
3    if (f v) then filter_stream f x' (asend y v)
4    else filter_stream f x' y
```

The problem of `filter_stream` is that on line 4 the priority of x' is displaced with respect to that of x, while that of y stays obviously the same and we are not able to differentiate the displacements for the priorities of different arguments. In general, the mechanism of priority displacement $\$^{\rho}$ allows us to deal with programs whose communication behavior follows a regular pattern and does not depend on the *data* being communicated. This is not the case for `filter_stream`. A solution could be to switch to rational (instead of integer) priorities and using more sophisticated priority mapping functions that allow priority "compression": the priority of x' could always be chosen to be half way between that of x and that of y. This extension, however, would invalidate Theorem 3.13, which holds only if the set of priorities is discrete, and would certainly complicate an hypothetical type reconstruction algorithm. A workaround could be the combined use of user-specified recursive types together with iso-recursion, in a way that mimics the usual treatment of recursive algebraic data types.

References

[1] T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *J. Funct. Program.*, 7 (3):321–347, 1997.

[2] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[3] R. Bird and L. Meertens. Nested datatypes. In *MPC'98*, LNCS 1422, pages 52–67. Springer, 1998.

[4] R. Bird and R. Paterson. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

[5] V. Bono, L. Padovani, and A. Tosatto. Polymorphic Types for Leak Detection in a Session-Oriented Functional Language. In *FORTE'13*, LNCS 7892, pages 83–98. Springer, 2013.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[7] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Inf. and Comput.*, 202(2):166–190, 2005.

[8] B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983. .

[9] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.

[10] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[11] K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

[12] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.

[13] A. Igarashi. Type-based analysis of usage of values for concurrent programming languages, 1997. Available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`.

[14] A. Igarashi and N. Kobayashi. Type-based analysis of communication for concurrent programming languages. In *SAS'97*, LNCS 1302, pages 187–201. Springer, 1997.

[15] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[16] N. Kobayashi. Quasi-linear types. In *POPL'99*, pages 29–42. ACM, 1999.

[17] N. Kobayashi. A type system for lock-free processes. *Inf. and Comput.*, 177(2):122–159, 2002.

[18] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, LNCS 4137, pages 233–247. Springer, 2006.

[19] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32 (5), 2010.

[20] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[21] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[22] F. Nielson and H. R. Nielson. Constraints for Polymorphic Behaviours of Concurrent ML. In *CCL'94*, LNCS 845, pages 73–88. Springer, 1994.

[23] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, LNCS 1710, pages 114–136. Springer, 1999.

[24] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL'94*, pages 84–97. ACM Press, 1994.

[25] L. Padovani. Deadlock and lock freedom in the linear π-calculus. Technical report, Università di Torino, 2014. Available at `http://hal.inria.fr/hal-00932356/`.

[26] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. in Comp. Sci.*, 6(5):409–453, 1996.

[27] F. Puntigam. Non-regular process types. In *Euro-Par'99*, LNCS 1685, pages 1334–1343. Springer, 1999.

[28] J. H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI'91*, pages 293–305. ACM, 1991.

[29] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[30] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[31] K. Suenaga and N. Kobayashi. Fractional ownerships for safe memory deallocation. In *APLAS'09*, LNCS 5904, pages 128–143. Springer, 2009.

[32] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *FPCA'95*, pages 1–11. ACM, 1995.

[33] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North Holland, 1990.

[34] P. Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286. ACM, 2012.

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

A. Supplement to Section 3

A.1 Basic definitions

Syntax

t	$::=$		**Type**
		$\mathtt{B}$	(basic type)
	$\mid$	α	(type variable)
	$\mid$	$t \times t$	(product)
	$\mid$	$p[t]^{\rho}$	(channel)
	$\mid$	$t \rightarrow^{\rho,\rho} t$	(arrow)
	$\mid$	$\$^{\rho} t$	(displaced type)
	$\mid$	$\mu\alpha.t$	(recursive type)
T	$::=$		**Type scheme**
		t	(type)
	$\mid$	$\forall\alpha^{\rho}.T$	(type polymorphism)
	$\mid$	$\forall\imath^{\varphi}.T$	(priority polymorphism)
ρ	$::=$		**Priority**
		$\bot$	(bottom)
	$\mid$	n	(finite)
	$\mid$	$\top$	(top)
	$\mid$	$\imath$	(priority variable)
	$\mid$	$\rho \wedge \rho$	(minimum/highest)
	$\mid$	$\rho \vee \rho$	(maximum/lowest)
	$\mid$	$\rho + \rho$	(displacement)
φ	$::=$		**Interval**
		$[\rho, \rho]$	(closed interval)
	$\mid$	$(\rho, \rho]$	(left-open interval)
	$\mid$	$[\rho, \rho)$	(right-open interval)
	$\mid$	(ρ, ρ)	(open interval)
Θ	$::=$		**Priority environment**
		$\emptyset$	(empty environment)
	$\mid$	$\Theta, \imath \in \varphi$	(priority variable binding)
	$\mid$	$\Theta, \alpha :: \rho$	(type variable binding)
Γ	$::=$		**Type environment**
		$\emptyset$	(empty environment)
	$\mid$	$\Gamma, u : T$	(name binding)

Priority of type schemes

$$|T|_{\Theta} \stackrel{\text{def}}{=} \begin{cases} \Theta(\alpha) & \text{if } T = \alpha \in \mathsf{dom}(\Theta) \\ \rho & \text{if } T = p[t]^{\rho} \text{ or } T = t \rightarrow^{\rho,\sigma} s \\ |t|_{\Theta} \wedge |s|_{\Theta} & \text{if } T = t \times s \\ \rho + |s|_{\Theta} & \text{if } T = \$^{\rho} s \\ \top & \text{otherwise} \end{cases}$$

Type (scheme) combination

$$T +_{\Theta} T \stackrel{\text{def}}{=} T \qquad \text{if } \mathsf{un}(t)$$
$$p[t]^{\rho} +_{\Theta} q[t]^{\rho} \stackrel{\text{def}}{=} \#[t]^{\rho} \qquad \text{if } \{p, q\} = \{?, !\}$$

Type environment combination

$$\Gamma +_{\Theta} \Gamma' \stackrel{\text{def}}{=} \Gamma, \Gamma' \qquad \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$$
$$(\Gamma, u : T) +_{\Theta} (\Gamma', u : S) \stackrel{\text{def}}{=} (\Gamma +_{\Theta} \Gamma'), u : T +_{\Theta} S$$

A.2 Restriction of priority environments

We write $\Theta|_{\{\imath_1,\ldots,\imath_n,\alpha_1,\ldots,\alpha_m\}}$ for the smallest well-formed priority environment Θ' included in Θ such that $\imath_i \in \mathsf{dom}(\Theta')$ for every $1 \leq i \leq n$ and $\alpha_j \in \mathsf{dom}(\Theta')$ for every $1 \leq j \leq m$. Note that $\Theta_{\{\imath\}}$ may include priority variables other than $\imath$ because the interval of $\imath$ may contain other priority variables. For example, if $\Theta = \imath \in \mathbb{P}, \jmath \in (\imath, \top]$, then $\Theta|_{\{\jmath\}} = \Theta$.

A.3 Basic properties

The following are standard properties of well-typed expressions and values. In all cases the proofs are simple inductions on the derivation of well typedness.

Lemma A.1. *Let* $\Gamma \vdash_{\Theta} e : t \,\&\, \rho$. *Then* $\mathsf{fn}(e) \subseteq \mathsf{dom}(\Gamma)$ *and* $\mathsf{ftv}(t) \cup \mathsf{fpv}(t) \cup \mathsf{pv}(\rho) \subseteq \mathsf{dom}(\Theta)$.

The following Lemma shows that a value has no effects, which is a common feature of effect systems. We implicitly use this property in the rest of the appendix and omit the priority from the judgments that regard values. Therefore, we will often write $\Gamma \vdash \mathsf{v} : t$ instead of $\Gamma \vdash \mathsf{v} : t \,\&\, \bot$.

Lemma A.2. *Let* $\Gamma \vdash_{\Theta} \mathsf{v} : t \,\&\, \rho$. *Then* $\rho = \bot$.

Lemma A.3. *Let* $\Gamma \vdash_{\Theta} \mathsf{v} : t$ *where* Γ *is ground. Then* $\Gamma|_{\mathsf{fn}(\mathsf{v})} \vdash_{\Theta|_{\mathsf{ftv}(t)\cup\mathsf{fpv}(t)}} \mathsf{v} : t$.

Corollary A.4. *Let* $\Gamma \vdash_{\Theta} \mathsf{v} : t$ *where* t *is closed and* Γ *is ground. Then* $\emptyset \vdash_{\emptyset} \Gamma : \mathsf{v} \,\&\, t$.

For values it is possible to establish an important relationship between the priority of their type and that of the type environments in which they are typed:

Lemma A.5. *If* $\Gamma \vdash_{\Theta} \mathsf{v} : t$, *then* $|t|_{\Theta} =_{\Theta} |\Gamma|_{\Theta}$.

Proof. By induction on v. If $\mathsf{v} = \mathtt{c}$, then $\top =_{\Theta} |\Gamma|_{\Theta}$ and $\mathsf{TypeOf}(\mathtt{c}) \succ_{\Theta} t$ and we conclude by analyzing the priority of constants. If $\mathsf{v} = a$, then $t = p[s]^{\rho}$ and $\Gamma = \Gamma', a : t$ and $\top =_{\Theta} |\Gamma'|_{\Theta}$. We conclude $|t|_{\Theta} = \rho =_{\Theta} |\Gamma'|_{\Theta} \wedge \rho =_{\Theta} \rho$. If $\mathsf{v} = \mathtt{fun}\ x\ e$, then $\Gamma, x : t_1 \vdash_{\Theta} e : t_2 \,\&\, \rho$ and $t = t_1 \rightarrow^{|\Gamma|_{\Theta},\rho} t_2$ and we conclude for reflexivity of $=_{\Theta}$. If $\mathsf{v} = (\mathsf{v}_1, \mathsf{v}_2)$, then $t = t_1 \times t_2$ and $\Gamma = \Gamma_1 +_{\Theta} \Gamma_2$ and $\Gamma_i \vdash_{\Theta} \mathsf{v}_i : t_i$. By induction hypothesis we deduce $|t_i|_{\Theta} =_{\Theta} |\Gamma_i|_{\Theta}$ for $i = 1, 2$. We conclude $|t|_{\Theta} = |t_1|_{\Theta} \wedge |t_2|_{\Theta} =_{\Theta} |\Gamma_1|_{\Theta} \wedge |\Gamma_2|_{\Theta} = |\Gamma|_{\Theta}$. The case for $\mathsf{v} = \mathtt{send}\ a$ is similar to that for a. □

We introduce the category of *quasi values* $\widehat{\mathsf{v}}, \ldots$, namely of expressions that are substituted. These include the values, as expected, but also recursive terms, which are substituted when unfolded:

$$\widehat{\mathsf{v}} ::= \mathsf{v} \mid \mathtt{rec}\ x\ e$$

For quasi values, we have a weaker property than that of Lemma A.5 because a recursive term may, in principle, have a linear type.

Lemma A.6. *If* $\Gamma \vdash_{\Theta} \widehat{\mathsf{v}} : t$, *then* $|t|_{\Theta} \leq_{\Theta} |\Gamma|_{\Theta}$.

Proof. When $\widehat{\mathsf{v}}$ is a value the result follows from Lemma A.5. When $\widehat{\mathsf{v}}$ is a recursion, we conclude using the hypothesis in rule [T-REC POLY] that the environment is unlimited. □

Note that Lemma A.6 does not hold for generic expressions. For example, we have that $a\,!\,3$ has type $\mathtt{unit}$ in the type environment $a : ![\mathtt{int}]^{n}$, however $\top = |\mathtt{unit}| \not\leq n = |![\mathtt{int}]^{n}|$.

A.4 Subject Reduction

This section is dedicated to the proof of Theorem 3.8 (subject reduction). The result is key for proving the soundness of the type system. The sequence of auxiliary results follows the same pattern as in [10], with the necessary adjustments due to the differences between the type systems.

Lemma A.7 (typability of subterms)**.** *If* $\mathscr{D}$ *is a derivation of* $\Gamma \vdash \mathscr{E}[e] : t \,\&\, \rho$, *then there exist* Θ, Γ_1, Γ_2, s, *and* σ *such that* $\Gamma = \Gamma_1 + \Gamma_2$, $\mathscr{D}$ *has a subderivation* $\mathscr{D}'$ *concluding* $\Gamma_1 \vdash_{\Theta} e : s \,\&\, \sigma$, *the position of* $\mathscr{D}'$ *in* $\mathscr{D}$ *corresponds to the position of the hole in* $\mathscr{E}$, *and* $\mathscr{D}'$ *does not end with a generalization or instantiation rule.*

Typing of expressions

$$\text{[T-NAME POLY]}\ \frac{\top \leq_\Theta |\Gamma| \qquad T \succ_\Theta t}{\Gamma, u : T \vdash_\Theta u : t \,\&\, \bot} \qquad \text{[T-LET POLY]}\ \frac{\Gamma_1 \vdash_{\Theta,\Theta'} e_1 : t \,\&\, \bot \qquad t \prec_{\Theta'} T \qquad \Gamma_2, x : T \vdash_\Theta e_2 : s \,\&\, \rho}{\Gamma_1 +_\Theta \Gamma_2 \vdash_\Theta \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : s \,\&\, \rho}$$

$$\text{[T-CONST POLY]}\ \frac{\top \leq_\Theta |\Gamma| \qquad \mathsf{TypeOf}(\texttt{c}) \succ_\Theta t}{\Gamma \vdash_\Theta \texttt{c} : t \,\&\, \bot} \qquad \text{[T-REC POLY]}\ \frac{\top \leq_\Theta |\Gamma|_\Theta \qquad \Gamma, x : T \vdash_{\Theta,\Theta'} e : t \,\&\, \bot \qquad \top \leq_\Theta |t|_\Theta \qquad t \prec_{\Theta'} T \qquad T \succ_\Theta s}{\Gamma \vdash_\Theta \texttt{rec}\ x\ e : s \,\&\, \bot}$$

$$\text{[T-FUN]}\ \frac{\Gamma, x : t \vdash_\Theta e : s \,\&\, \rho}{\Gamma \vdash_\Theta \texttt{fun}\ x\ e : t \rightarrow^{|\Gamma|_\Theta, \rho} s \,\&\, \bot} \qquad \text{[T-APP]}\ \frac{\Gamma_1 \vdash_\Theta e_1 : t \rightarrow^{\rho,\sigma} s \,\&\, \tau_1 \qquad \Gamma_2 \vdash_\Theta e_2 : t \,\&\, \tau_2 \qquad \tau_1 <_\Theta |\Gamma_2|_\Theta \qquad \tau_2 <_\Theta \rho}{\Gamma_1 + \Gamma_2 \vdash_\Theta e_1 e_2 : s \,\&\, \sigma \vee \tau_1 \vee \tau_2}$$

$$\text{[T-PAIR]}\ \frac{\Gamma_i \vdash_\Theta e_i : t_i \,\&\, \rho_i\ ^{(i=1,2)} \qquad \rho_1 <_\Theta |\Gamma_2|_\Theta \qquad \rho_2 <_\Theta |t_1|_\Theta}{\Gamma_1 + \Gamma_2 \vdash_\Theta (e_1, e_2) : t_1 \times t_2 \,\&\, \rho_1 \vee \rho_2} \qquad \text{[T-SPLIT]}\ \frac{\Gamma_1 \vdash_\Theta e_1 : t_1 \times t_2 \,\&\, \rho \qquad \Gamma_2, x : t_1, y : t_2 \vdash_\Theta e_2 : s \,\&\, \sigma \qquad \rho <_\Theta |\Gamma_2|_\Theta}{\Gamma_1 + \Gamma_2 \vdash_\Theta \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : s \,\&\, \rho \vee \sigma}$$

Typing of processes

$$\text{[T-THREAD]}\ \frac{\Gamma \vdash_\emptyset e : \texttt{unit} \,\&\, \rho}{\Gamma \vdash \{e\}} \qquad \text{[T-PAR]}\ \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q} \qquad \text{[T-NEW]}\ \frac{\Gamma, a : \texttt{\#}[t]^n \vdash P}{\Gamma \vdash \texttt{new}\ a\ \texttt{in}\ P}$$

Table 5. Complete typing rules for expressions and processes.

Proof. By induction on $\mathscr{E}$. □

Lemma A.8 (replacement)**.** *If*

1. *$\mathscr{D}$ is a derivation of $\Gamma_0 + \Gamma \vdash_\Theta \mathscr{E}[e] : t_0 \,\&\, \rho$,*
2. *$\mathscr{D}'$ is a subderivation of $\mathscr{D}$ concluding $\Gamma \vdash_{\Theta'} e : t \,\&\, \sigma$,*
3. *the position of $\mathscr{D}'$ in $\mathscr{D}$ corresponds to the position of $[\,]$ in $\mathscr{E}$,*
4. *$\Gamma' \vdash_{\Theta'} e' : t \,\&\, \sigma'$ where $\sigma' \leq_\Theta \sigma$,*
5. *$\Gamma_0 + \Gamma'$ is defined,*

then $\Gamma_0 + \Gamma' \vdash_\Theta \mathscr{E}[e'] : t \,\&\, \rho'$ and $\rho' \leq_\Theta \rho$. Furthermore, $\sigma' =_\Theta \sigma$ implies $\rho' =_\Theta \rho$.

Proof. By induction on $\mathscr{E}$. □

Lemma A.9 (instantiation)**.** *If $\Gamma \vdash_{\Theta,\Theta'} \widehat{\mathsf{v}} : t$ and Γ is ground and $t \prec_{\Theta'} T$ and $T \succ_\Theta s$, then $\Gamma \vdash_\Theta \widehat{\mathsf{v}} : s$.*

Lemma A.10 (substitution)**.** *If*

1. *$\Gamma_0, x : T \vdash_\Theta e : s \,\&\, \rho$, and*
2. *$\Gamma_1 \vdash_{\Theta,\Theta'} \widehat{\mathsf{v}} : t$, and*
3. *$\Gamma_0 + \Gamma_1 =_\Theta \Gamma$, and*
4. *$t \prec_{\Theta'} T$,*

then $\Gamma \vdash_\Theta e\{\widehat{\mathsf{v}}/x\} : s \,\&\, \rho$.

Proof. The proof is almost entirely conventional. The only critical aspect is the fact that x may occur non linearly within e, hence multiple copies of $\widehat{\mathsf{v}}$ may be necessary. In these cases, however, the type scheme T of x must be unlimited, as by definition of type combination, and from Lemma A.6 one deduces that Γ_1 is also unlimited, meaning that $\Gamma_1 + \Gamma_1$ is defined and equal to Γ_1 itself. □

Lemma A.11 (subject reduction for expressions)**.** *Let $\Gamma \vdash_\Theta e : t \,\&\, \rho$ and $e \longrightarrow e'$. Then $\Gamma \vdash_\Theta e' : t \,\&\, \rho$.*

Proof. By induction on the derivation of $e \longrightarrow e'$.

$\boxed{e = \texttt{let}\ x = \mathsf{v}\ \texttt{in}\ e'' \longrightarrow e''\{\mathsf{v}/x\} = e'}$ From [T-LET POLY] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma_1 + \Gamma_2 =_\Theta \Gamma$ and $\Gamma_1 \vdash_{\Theta,\Theta'} \mathsf{v} : s$ and $s \prec_{\Theta'} T$ and $\Gamma_2, x : T \vdash_\Theta e'' : t \,\&\, \rho$. We conclude $\Gamma \vdash_\Theta e' : t \,\&\, \rho$ with an application of Lemma A.10.

$\boxed{e = \texttt{let}\ (x, y) = (\mathsf{v}_1, \mathsf{v}_2)\ \texttt{in}\ e'' \longrightarrow e''\{\mathsf{v}_1, \mathsf{v}_2/x, y\} = e'}$ From [T-SPLIT] we deduce that there exist Γ_1 and Γ_2 such that $+_\Theta \Gamma_1 \Gamma_2 \Gamma$ and $\Gamma_1 \vdash_\Theta (\mathsf{v}_1, \mathsf{v}_2) : t_1 \times t_2$ and $\Gamma_2, x : t_1, y : t_2 \vdash_\Theta e'' : t \,\&\, \rho$. From [T-PAIR] we deduce that there exist Γ_{11} and Γ_{12} such that $\Gamma_1 = \Gamma_{11} +_\Theta \Gamma_{12}$ and $\Gamma_{1i} \vdash_\Theta \mathsf{v}_i : t_i$ for every $i = 1, 2$. By two applications of Lemma A.10 we conclude $\Gamma \vdash_\Theta e''\{\mathsf{v}_1, \mathsf{v}_2/x, y\} : t \,\&\, \rho$.

$\boxed{e = (\texttt{fun}\ x\ e'')\mathsf{v} \longrightarrow e''\{\mathsf{v}/x\} = e'}$ From [T-APP] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 +_\Theta \Gamma_2$ and $\Gamma_1 \vdash_\Theta \texttt{fun}\ x\ e'' : s \rightarrow^{\sigma,\rho} t$ and $\Gamma_2 \vdash_\Theta \mathsf{v} : s$. From [T-FUN] we deduce $\Gamma_1, x : s \vdash_\Theta e'' : t \,\&\, \rho$ and $\sigma = |\Gamma_1|_\Theta$. We conclude $\Gamma \vdash_\Theta e''\{\mathsf{v}/x\} : t \,\&\, \rho$ with an application of Lemma A.10.

$\boxed{e = \texttt{rec}\ x\ e'' \longrightarrow e''\{\texttt{rec}\ x\ e''/x\} = e'}$ From [T-REC POLY] we deduce $\top \leq_\Theta |\Gamma|$ and $\Gamma, x : T \vdash_{\Theta,\Theta'} e'' : s$ and $s \prec_{\Theta'} T$ and $T \succ_\Theta t$ and $\rho = \bot$. We conclude $\Gamma \vdash_\Theta e' : t \,\&\, \bot$ with an application of Lemma A.10.

$\boxed{e = \mathscr{E}[e_0] \longrightarrow \mathscr{E}[e_0'] = e' \text{ where } e_0 \longrightarrow e_0'}$ By Lemma A.7 we deduce that there exist Γ_1, Γ_2, Θ', s, and σ such that $\Gamma = \Gamma_1 +_\Theta \Gamma_2$ and $\Gamma_1 \vdash_{\Theta'} e : s \,\&\, \sigma$. By induction hypothesis we deduce $\Gamma_1 \vdash_{\Theta'} e' : s \,\&\, \sigma$. By Lemma A.8 we conclude $\Gamma \vdash_\Theta \mathscr{E}[e'] : t \,\&\, \rho$. □

Lemma A.12. *Let $\Gamma \vdash P$ and $P \equiv Q$. Then $\Gamma \vdash Q$.*

Proof. Straightforward induction on the derivation of $P \equiv Q$. □

Theorem A.13 (subject reduction for processes)**.** *Let $\Gamma \vdash P$ and $P \xrightarrow{\ell} P'$. Then $\Gamma - \ell \vdash P'$.*

Proof. By induction on the derivation of $P \xrightarrow{\ell} P'$.

$\boxed{P = \{\mathscr{E}[a\,!\,\mathsf{v}]\} \mid \{\mathscr{E}'[\texttt{recv}\ a]\} \xrightarrow{a} \{\mathscr{E}[()]\} \mid \{\mathscr{E}'[\mathsf{v}]\} = P'}$ From [T-PAR] and [T-THREAD] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash \mathscr{E}[a\,!\,\mathsf{v}] : \texttt{unit} \,\&\, \rho_1$ and $\Gamma_2 \vdash$

$\mathscr{E}'[\texttt{recv}\ a] : \texttt{unit}\ \&\ \rho_2$. By Lemma A.7 we deduce that there exist Θ_i and Γ_{ij} for $i, j \in \{1, 2\}$ such that $\Gamma_i = \Gamma_{i1} + \Gamma_{i2}$ for $i = 1, 2$ and $\Gamma_{12} \vdash_{\Theta_1} a\texttt{!}\mathsf{v} : \texttt{unit}\ \&\ n$ and $\Gamma_{22} \vdash_{\Theta_2} \texttt{recv}\ a : t\ \&\ n$. From the type of $\texttt{send}$ and the fact that Γ is ground we deduce $\Gamma_{12} = \Gamma'_{12} +_{\Theta_1} a : \texttt{![}t\texttt{]}^n$ and $\Gamma'_{12} \vdash_{\Theta_1} \mathsf{v} : t$ where t is closed. By Corollary A.4 we deduce that $\Gamma'_{12} \vdash \mathsf{v} : t$. Let $\Gamma'_2 = \Gamma'_{12} + \Gamma_{21}$. By Lemma A.8 we deduce $\Gamma_{11} \vdash \mathscr{E}[\texttt{()}] : \texttt{unit}\ \&\ \sigma_1$ and $\Gamma'_2 \vdash \mathscr{E}'[\mathsf{v}] : \texttt{unit}\ \&\ \sigma_2$ where $\sigma_i \leq \rho_i$ for $i = 1, 2$. We conclude with [T-THREAD] and [T-PAR].

$\boxed{P = \{\mathscr{E}[\texttt{fork}\ \mathsf{v}]\} \xrightarrow{\tau} \{\mathscr{E}[\texttt{()}]\} \mid \{\mathsf{v}\texttt{()}\} = P'}$ From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{fork}\ \mathsf{v}] : \texttt{unit}\ \&\ \rho$. By Lemma A.7 we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash_\Theta \texttt{fork}\ \mathsf{v} : \texttt{unit}$. From [T-APP] and $\mathsf{TypeOf}(\texttt{fork})$ we deduce $\Gamma_1 \vdash_\Theta \mathsf{v} : \texttt{unit} \rightarrow^{\rho_1, \rho_2} \texttt{unit}$. By Lemma A.8 and an application of [T-THREAD] we derive $\Gamma_2 \vdash \{\mathscr{E}[\texttt{()}]\}$. By [T-APP] we derive $\Gamma_1 \vdash \mathsf{v}\texttt{()} : \texttt{unit}\ \&\ \rho_2$. We conclude with one application of [T-THREAD] and one of [T-PAR].

$\boxed{P = \{\mathscr{E}[\texttt{open()}]\} \xrightarrow{\tau} \texttt{new}\ a\ \texttt{in}\ \{\mathscr{E}[a]\} = P' \text{ and } a \notin \mathsf{fn}(\mathscr{E})}$ From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{open()}] : \texttt{unit}\ \&\ \rho$. By Lemma A.7 we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash_\Theta \texttt{open()} : \texttt{\#[}t\texttt{]}^\sigma$. Furthermore, it must be the case that $\mathsf{un}(\Gamma_1)$ because $\texttt{open()}$ is a closed expression. Using [T-NAME] we derive $\Gamma_1, a : \texttt{\#[}t\texttt{]}^\sigma \vdash_\Theta a : \texttt{\#[}t\texttt{]}^\sigma$. From the hypothesis $a \notin \mathsf{fn}(\mathscr{E})$ we deduce $a \notin \mathsf{dom}(\Gamma_2)$ hence $\Gamma, a : \texttt{\#[}t\texttt{]}^\sigma = (\Gamma_1, a : \texttt{\#[}t\texttt{]}^\sigma) + \Gamma_2$. By Lemma A.8 we derive $\Gamma, a : \texttt{\#[}t\texttt{]}^\sigma \vdash \mathscr{E}[a] : \texttt{unit}\&\rho$. We conclude $\Gamma \vdash \texttt{new}\ a\ \texttt{in}\ \{\mathscr{E}[a]\}$ with one application of [T-THREAD] and one of [T-NEW].

$\boxed{P = \{e\} \xrightarrow{\tau} \{e'\} = P' \text{ where } e \longrightarrow e'}$ From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[e] : \texttt{unit}\ \&\ \rho$. By Lemma A.11 we deduce $\Gamma \vdash e' : \texttt{unit}\ \&\ \rho$. We conclude with one application of [T-THREAD].

$\boxed{P = P_1 \mid P_2 \xrightarrow{\ell} P'_1 \mid P_2 = P' \text{ where } P_1 \xrightarrow{\ell} P'_1}$ From [T-PAR] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_i \vdash P_i$ for every $i = 1, 2$. By induction hypothesis we have $\Gamma_1 - \ell \vdash P'_1$ and we conclude $\Gamma - \ell \vdash P'$ with one application of [T-PAR] and observing that $\Gamma - \ell = (\Gamma_1 - \ell) + \Gamma_2$.

$\boxed{P = \texttt{new}\ a\ \texttt{in}\ Q \xrightarrow{\ell} \texttt{new}\ a\ \texttt{in}\ Q' = P' \text{ where } Q \xrightarrow{\ell} Q' \text{ and } \ell \neq a}$ From [T-NEW] we deduce that $\Gamma, a : \texttt{\#[}t\texttt{]}^n \vdash Q$. By induction hypothesis and using the fact that $\ell \neq a$ we deduce $(\Gamma - \ell), a : \texttt{\#[}t\texttt{]}^n \vdash Q'$. We conclude $\Gamma - \ell \vdash Q'$ with one application of [T-NEW].

$\boxed{P = \texttt{new}\ a\ \texttt{in}\ Q \xrightarrow{\tau} P' \text{ where } Q \xrightarrow{a} P'}$ From [T-NEW] we deduce $\Gamma, a : \texttt{\#[}t\texttt{]}^n \vdash Q$. We conclude $\Gamma \vdash P'$ by induction hypothesis and observing that $(\Gamma, a : \texttt{\#[}t\texttt{]}^n) - a = \Gamma$.

$\boxed{P \equiv Q \xrightarrow{\ell} Q' \equiv P'}$ Straightforward application of Lemma A.12 and the induction hypothesis. □

A.5 Confluence

This section is devoted to the proof of Theorem 3.9. The first auxiliary result is a conventional lemma about canonical forms, inferring the shape of a value having a given type.

Lemma A.14 (canonical forms)**.** *Let* $\Gamma \vdash_\Theta \mathsf{v} : t$*. Then:*

- *if* $t = \texttt{unit}$*, then* $\mathsf{v} = \texttt{()}$*;*
- *if* $t = p\texttt{[}s\texttt{]}^\rho$*, then* v *is a channel;*
- *if* $t = t_1 \times t_2$*, then* $\mathsf{v} = \texttt{(}\mathsf{v}_1\texttt{,}\ \mathsf{v}_2\texttt{)}$*;*
- *if* $t = t_1 \rightarrow^{\rho,\sigma} t_2$*, then* v *is either an abstraction or* $(\texttt{send}\ a)$ *or one of the constants in the set* $\{\texttt{fork}, \texttt{open}, \texttt{send}, \texttt{recv}\}$*;*

Proof. By a trivial case analysis on the typing rules of Table 5. □

Next, we define the notion of *reducible expression*, namely of an expression that, when occurring within an evaluation context, it may potentially trigger a reduction of the thread in which it occurs, possibly in combination with another thread in case of a synchronization.

Definition A.15 (redex)**.** A *redex* r is an expression having one of the following forms:

$$
\begin{array}{rcl}
\mathsf{r} & ::= & (\texttt{fun}\ x\ e)\mathsf{v} \\
& \mid & \texttt{rec}\ x\ e \\
& \mid & \texttt{let}\ x\ \texttt{=}\ \mathsf{v}\ \texttt{in}\ e \\
& \mid & \texttt{let}\ \texttt{(}x\texttt{,}\ y\texttt{)}\ \texttt{=}\ \texttt{(}\mathsf{v}\texttt{,}\ \mathsf{w}\texttt{)}\ \texttt{in}\ e \\
& \mid & \texttt{fork}\ \mathsf{v} \\
& \mid & \texttt{open()} \\
& \mid & a\texttt{!}\mathsf{v} \\
& \mid & \texttt{recv}\ a
\end{array}
$$

Note that we consider $a\texttt{!}\mathsf{v}$ and $\texttt{recv}\ \mathsf{v}$ to be redexes, even though they are not able to reduce independently, but only if they occur within two parallel threads. The next result shows that each well-typed expression has at most one redex.

Lemma A.16. *Let* $\mathsf{red}(e) \stackrel{def}{=} \{\mathscr{E} \mid e = \mathscr{E}[\mathsf{r}]$ *and* r *is a redex*$\}$ *and* $\Gamma \vdash_\Theta e : t\ \&\ \rho$ *where* Γ *is ground. The following properties hold:*

1. *if* e *is a value, then* $\mathsf{red}(e) = \emptyset$*;*
2. *if* e *is not a value, then* $\mathsf{red}(e)$ *is a singleton.*

Proof. We prove both properties simultaneously, by induction on e and by cases on its structure.

If e is a constant, a channel, or an abstraction then it is a value and we conclude by observing that $\mathsf{red}(e) = \emptyset$. Note that e cannot be a variable, because Γ is assumed to be ground.

Suppose $e = \texttt{rec}\ x\ e'$. Then e is not a value and we conclude by observing that $\mathsf{red}(e) = \{[\,]\}$.

Suppose $e = \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$. Then e is not a value and we must show that $\mathsf{red}(e)$ is a singleton. From [T-LET] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 +_\Theta \Gamma_2$ and $\Gamma_1 \vdash_{\Theta,\Theta'} e_1 : s\ \&\ \sigma$. We distinguish two subcases: if e_1 is a value, then by induction hypothesis $\mathsf{red}(e_1) = \emptyset$ and we conclude by observing that $\mathsf{red}(e) = \{[\,]\}$; if e_1 is not a value, then by induction hypothesis $\mathsf{red}(e_1)$ is a singleton and we conclude by observing that $\mathsf{red}(e) = \{\texttt{let}\ x\ \texttt{=}\ \mathscr{E}\ \texttt{in}\ e_2 \mid \mathscr{E} \in \mathsf{red}(e_1)\}$.

Suppose $e = \texttt{let}\ \texttt{(}x\texttt{,}\ y\texttt{)}\ \texttt{=}\ e_1\ \texttt{in}\ e_2$. Then e is not a value and from [T-SPLIT] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 +_\Theta \Gamma_2$ and $\Gamma_1 \vdash_\Theta e_1 : t_1 \times t_2\ \&\ \sigma$. We distinguish two subcases: if e_1 is a value, then by induction hypothesis we have $\mathsf{red}(e_1) = \emptyset$ and by Lemma A.14 we deduce that it must have the form $\texttt{(}\mathsf{v}\texttt{,}\ \mathsf{w}\texttt{)}$, so we conclude by observing that $\mathsf{red}(e) = \{[\,]\}$; if e_1 is not a value, then by induction hypothesis we have that $\mathsf{red}(e_1)$ is a singleton and we conclude by observing that $\mathsf{red}(e) = \{\texttt{let}\ \texttt{(}x\texttt{,}\ y\texttt{)}\ \texttt{=}\ \mathscr{E}\ \texttt{in}\ e_2 \mid \mathscr{E} \in \mathsf{red}(e_1)\}$.

Suppose $e = \texttt{(}e_1\texttt{,}\ e_2\texttt{)}$. From [T-PAIR] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma_i \vdash_\Theta e_i : t_i\ \&\ \rho_i$ for $i = 1, 2$. We distinguish three subcases: if both e_1 and e_2 are values, then by induction hypothesis $\mathsf{red}(e_1) = \mathsf{red}(e_2) = \emptyset$ and we conclude by ovserving that e is also a value and $\mathsf{red}(e) = \emptyset$; if e_1 is a value but e_2 is not, then by induction hypothesis we deduce that $\mathsf{red}(e_1) = \emptyset$ and $\mathsf{red}(e_2)$ is a singleton and we conclude by observing that $\mathsf{red}(e) = \{\texttt{(}e_1\texttt{,}\ \mathscr{E}\texttt{)} \mid \mathscr{E} \in \mathsf{red}(e_2)\}$; if e_1 is not a value, then by induction hypothesis we deduce that $\mathsf{red}(e_1)$ is a singleton and we conclude by observing that $\mathsf{red}(e) = \{\texttt{(}\mathscr{E}\texttt{,}\ e_2\texttt{)} \mid \mathscr{E} \in \mathsf{red}(e_1)\}$.

Suppose $e = e_1 e_2$. Then e is not a value and from [T-APP] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma_1 \vdash_\Theta e_1 : s \rightarrow^{\sigma,\tau} t\,\&\,\rho_1$ and $\Gamma_2 \vdash_\Theta e_2 : s\ \&\ \rho_2$. We distinguish three subcases but we only discuss the subcase in which e_1 and e_2 are both values, since the other subcases are similar to those already discussed for pairs.

By induction hypothesis we deduce that $\mathsf{red}(e_1) = \mathsf{red}(e_2) = \emptyset$, hence we must conclude that e is a redex. By Lemma A.14 we deduce that e_1 is either an abstraction or the application $\mathtt{send}\ a$ or one of the constants in the set $\{\mathtt{fork}, \mathtt{open}, \mathtt{send}, \mathtt{recv}\}$. In each case we are able to conclude that e is a redex, possibly using Lemma A.14 again for deducing that $\mathtt{open}$ is applied to $\mathtt{()}$ and the arguments of $\mathtt{send}$ and $\mathtt{recv}$ have the right shape. □

Corollary A.17. *The only well-typed thread without redexes is* $\{\mathtt{()}\}$.

Proof. Let $\Gamma \vdash \{e\}$ where e has no redex. By Lemma A.16 we deduce that e is a value. By Lemma A.14, the only value of type $\mathtt{unit}$ is $\mathtt{()}$. □

Theorem A.18 (Theorem 3.9)**.** *Let $\Gamma \vdash P$ and $P \xrightarrow{\ell_1} P_1$ and $P \xrightarrow{\ell_2} P_2$. Then either $P_1 \equiv P_2$ or there exist Q such that $P_1 \xrightarrow{\ell_2} Q$ and $P_2 \xrightarrow{\ell_1} Q$.*

Proof. Without loss of generality we may assume that $P \equiv \prod_{i \in I} Q_i$ where the Q_i are threads different from $\{\mathtt{()}\}$. Indeed, if there are restrictions these can always be brought up at the top level using structural congruence and the inner process is still well typed. Also, $\{\mathtt{()}\}$ threads can be removed again by structural congruence. Now, from Lemma A.16 we know that each Q_i has exactly one redex, so it can reduce in at most one way, either independently of the other threads or in combination with another thread Q_j if a communication takes place. In the latter case, Q_j is uniquely identified because a linear channel cannot occur in more than two threads. In every case it is easy to see that the statement holds. □

A.6 Deadlock Freedom

This section is devoted to the proof of the soundness theorem, namely Theorem 3.11. The first auxiliary result states that the effect of an expression is (numerically) larger than the priority of a channel that is the subject of a communication in a well-typed expression. That is, the effect is a conservative approximation of the priority of all the I/O operations performed during the evaluation of a well-typed expression.

Lemma A.19. *Let Γ be ground and either $\Gamma \vdash_\Theta \mathscr{E}[\mathtt{recv}\ a] : t \,\&\, \rho$ or $\Gamma \vdash_\Theta \mathscr{E}[a\,\mathtt{!}\mathsf{v}] : t \,\&\, \rho$. Then $|\Gamma(a)| \leq \rho$.*

Proof. This is a straightforward induction on $\mathscr{E}$, using the fact that the typing rules accumulate effects in the conclusion. □

The second auxiliary result states the relationship between the priority of the subject of a communication that is a redex, and the priority of any other channel that may occur in a well-typed expression. In particular, the subject of the communication is the channel with the *highest* (*i.e.*, numerically smallest) priority occurring free in the expression.

Lemma A.20. *Let Γ be ground and either*

- $\Gamma \vdash_\Theta \mathscr{E}[\mathtt{recv}\ a] : t \,\&\, \rho$ *and* $b \in \mathsf{fn}(\mathscr{E})$, *or*
- $\Gamma \vdash_\Theta \mathscr{E}[a\,\mathtt{!}\mathsf{v}] : t \,\&\, \rho$ *and* $b \in \mathsf{fn}(\mathscr{E}) \cup \mathsf{fn}(\mathsf{v})$.

Then $|\Gamma(a)| < |\Gamma(b)|$.

Proof. We prove the result assuming $\Gamma \vdash_\Theta \mathscr{E}[a\,\mathtt{!}\mathsf{v}] : t \,\&\, \rho$ and $b \in \mathsf{fn}(\mathscr{E}) \cup \mathsf{fn}(\mathsf{v})$. With the other hypothesis the proof is analogous. We proceed by induction on $\mathscr{E}$ and by cases on its shape.

Suppose $\mathscr{E} = [\ ]$. Then it must be the case that $b \in \mathsf{fn}(\mathsf{v})$. From [T-APP] and the type of $\mathtt{send}$ we deduce that there exist Γ_1 and Γ_2 such that $\Gamma_1 +_\Theta \Gamma_2 = \Gamma$ and $\Gamma_1 \vdash_\Theta \mathtt{send}\ a : s \to^{n,n} \mathtt{unit}$ and $\Gamma_2 \vdash_\Theta \mathsf{v} : s$ and $\Gamma(a) = \mathtt{!}[s]^n$ and $n < |\Gamma_2|$. We conclude $|\Gamma(a)| = n < |\Gamma_2| \leq |\Gamma_2(b)| = |\Gamma(b)|$.

Suppose $\mathscr{E} = (\mathscr{E}'\mathtt{,}\ e)$. From [T-PAIR] we deduce that there exist Γ_1 and Γ_2 and t_1 and t_2 such that $\Gamma_1 +_\Theta \Gamma_2 = \Gamma$ and $\Gamma_1 \vdash_\Theta \mathscr{E}'[\mathtt{recv}\ a] : t_1 \,\&\, \rho_1$ and $\Gamma_2 \vdash_\Theta e : t_2 \,\&\, \rho_2$ and $\rho_1 < |\Gamma_2|$. We distinguish two subcases. If $b \in \mathsf{fn}(\mathscr{E}')$, then by induction hypothesis we deduce $|\Gamma_1(a)| < |\Gamma_1(b)|$ and we conclude by observing that $|\Gamma(a)| = |\Gamma_1(a)|$ and $|\Gamma(b)| = |\Gamma_1(b)|$. If $b \in \mathsf{fn}(e)$, then we conclude

$$\begin{array}{rcll} |\Gamma(a)| & = & |\Gamma_1(a)| & \text{by definition of } +_\Theta \\ & \leq & \rho_1 & \text{by Lemma A.19} \\ & < & |\Gamma_2| & \text{from [T-PAIR]} \\ & \leq & |\Gamma_2(b)| & \text{by definition of } |\Gamma_2| \\ & = & |\Gamma(b)| & \text{by definition of } +_\Theta \end{array}$$

Suppose $\mathscr{E} = (\mathsf{v}'\mathtt{,}\ \mathscr{E}')$. From [T-PAIR] we deduce that there exist Γ_1 and Γ_2 such that $\Gamma_1 +_\Theta \Gamma_2 = \Gamma$ and $\Gamma_1 \vdash_\Theta \mathsf{v}' : t_1$ and $\Gamma_2 \vdash_\Theta \mathscr{E}'[a\,\mathtt{!}\mathsf{v}] : t_2 \,\&\, \rho$ and $\rho < |t_1|$. We distinguish two subcases. If $b \in \mathsf{fn}(\mathsf{v}')$, then we conclude

$$\begin{array}{rcll} |\Gamma(a)| & = & |\Gamma_2(a)| & \text{by definition of } +_\Theta \\ & \leq & \rho & \text{by Lemma A.19} \\ & < & |t_1| & \text{from [T-PAIR]} \\ & = & |\Gamma_1| & \text{by Lemma A.5} \\ & \leq & |\Gamma_1(b)| & \text{by definition of } |\Gamma_1| \\ & = & |\Gamma(b)| & \text{by definition of } +_\Theta \end{array}$$

If $b \in \mathsf{fn}(\mathscr{E}')$, then we conclude $|\Gamma(a)| = |\Gamma_2(a)| < |\Gamma_2(b)| = |\Gamma(b)|$ using the definition of $+_\Theta$ and the induction hypothesis.

The remaining cases are similar. □

We are now approaching the core of the proof. Before establishing the crucial lemma, we introduce some convenient terminology regarding type environments.

Definition A.21 (even and odd type environment)**.** We say that Γ is *even* if for every $u \in \mathsf{dom}(\Gamma)$ we have that $\Gamma(u)$ is a channel type with # polarity. We say that it is *odd* if, for every $u \in \mathsf{dom}(\Gamma)$ we have that $\Gamma(u)$ is a channel type with either ? or ! polarity.

Note that the empty type environment is both even and odd. In the following, we use Γ_{even} (respectively, Γ_{odd}) to range over even (respectively, odd) type environments. Note that any ground type environment Γ can be split into a pair $\Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}}$. A fundamental property of well-typed, stable processes, those that cannot reduce any further, is that they must be typed in an environment where the odd part contains at least one channel whose priority is higher (*i.e.*, numerically smaller) than the priority of all the channels in the even part. That is, the process cannot be blocked on a I/O operation for a channel that occurs with both polarities in the very same process.

Lemma A.22. *Let $\Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}} \vdash P$ and $P \not\xrightarrow{\tau}$ and $\Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}}$ is ground and all channels in Γ_{odd} have odd polarity and all channels in Γ_{even} have even polarity. Then $|\Gamma_{\mathsf{odd}}| \leq |\Gamma_{\mathsf{even}}|$.*

Proof. We do an induction on the number of restrictions in P.

In the base case P has no restrictions. Then, from the hypothesis $P \not\xrightarrow{\tau}$ it must be the case that

$$P \equiv \prod_{i \in I} \{\mathscr{E}_i[\mathtt{recv}\ a_i]\} \mid \prod_{j \in J} \{\mathscr{E}'_j[b_j\,\mathtt{!}\mathsf{v}_j]\}$$

Suppose, by contradiction, that $|\Gamma_{\mathsf{even}}| < |\Gamma_{\mathsf{odd}}|$ and let $c \in \mathsf{dom}(\Gamma_{\mathsf{even}})$ such that $|\Gamma_{\mathsf{even}}(c)| = |\Gamma_{\mathsf{even}}|$. By Lemma A.20 we deduce that $c \notin \mathsf{fn}(\mathscr{E}_i)$ for every $i \in I$ and $c \notin \mathsf{fn}(\mathscr{E}'_j) \cup \mathsf{fn}(\mathsf{v}_j)$ for every $j \in J$, because c has minimum priority hence it cannot be blocked by other operations on channels with greater or equal priority. Then, since $c \in \mathsf{dom}(\Gamma_{\mathsf{even}})$ and all channel types in Γ_{even} have even polarity, it must be the case that $c = a_i = b_j$ for some $i \in I$ and $j \in J$. This contradicts the hypothesis that $P \not\xrightarrow{\tau}$.

We conclude that the assumption $|\Gamma_{\mathsf{even}}| < |\Gamma_{\mathsf{odd}}|$ is absurd, hence $|\Gamma_{\mathsf{odd}}| \leq |\Gamma_{\mathsf{even}}|$.

In the inductive case, we have $P \equiv \texttt{new}\ a\ \texttt{in}\ Q$ where Q has fewer restrictions than P. From the hypothesis $\Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}} \vdash P$ and [T-NEW] we deduce $\Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}}, a : \texttt{\#}[t]^n \vdash Q$. By induction hypothesis we have $|\Gamma_{\mathsf{odd}}| \leq |\Gamma_{\mathsf{even}}, a : \texttt{\#}[t]^n|$. We conclude by observing that $|\Gamma_{\mathsf{even}}, a : \texttt{\#}[t]^n| = |\Gamma_{\mathsf{even}}| \wedge n \leq |\Gamma_{\mathsf{even}}|$. □

An easy consequence of Lemma A.22 is that every stable process that is well typed in an even environment must be structurally equivalent to $\{()\}$.

Lemma A.23. *Let Γ_{even} be ground and $\Gamma_{\mathsf{even}} \vdash P$ and $P \not\xrightarrow{\tau}$. Then $P \equiv \{()\}$.*

Proof. We do an induction on the number of restrictions in P. If P has no restrictions, then from Lemma A.22 we deduce that $\Gamma_{\mathsf{even}} = \emptyset$, namely that P is a closed process. We conclude observing that every well-typed, closed process is structurally congruent to $\{()\}$. If $P \equiv \texttt{new}\ a\ \texttt{in}\ Q$, then there exists Γ'_{even} such that $\Gamma'_{\mathsf{even}} \vdash Q$. By induction hypothesis we deduce $Q \equiv \{()\}$. This contradicts the hypothesis that P is well typed, since a must occur free in Q, hence this case is impossible. □

Soundness of the type system is then an easy corollary of the previous Lemmas.

Theorem A.24 (Theorem 3.11)**.** *Let $\emptyset \vdash P$. Then P is deadlock free.*

Proof. Straightforward consequence of Theorem A.13 and of Lemma A.23. □

A.7 Interactivity

The first auxiliary result we need states that reductions can only decrease the priority of a type environment.

Lemma A.25. *If $\Gamma - \ell$ is defined, then $|\Gamma| \leq |\Gamma - \ell|$.*

Proof. Immediate consequence of the definition of $\Gamma - \ell$. □

Then, we show that every interactive typed process has a sequence of transitions (possibly involving channels with odd type) that leads to a new state in which the priority of channels is strictly lower than in the original process.

Lemma A.26. *Let $\Gamma \fatsemi P$ be an interactive typed process such that $|\Gamma| < \top$. Then there exist $\lambda_1, \ldots, \lambda_n$ and Γ' such that*

$$P \xmapsto{\lambda_1} \cdots \xmapsto{\lambda_n} P'$$

and $\Gamma' \fatsemi P'$ is an interactive typed process and $|\Gamma| < |\Gamma'|$.

Proof. Consider a maximal reduction

$$P \xrightarrow{\ell_1} P_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_m} P_m \not\xrightarrow{\tau}$$

We know that such reduction exists and is finite because $\Gamma \fatsemi P$ is interactive. Let $\Gamma'' = (\cdots((\Gamma - \ell_1) - \ell_2) - \cdots - \ell_m)$. From Theorem A.13 we know that $\Gamma'' \vdash P_m$ and from Lemma A.25 we also know $|\Gamma| \leq |\Gamma''|$. If $|\Gamma| < |\Gamma''|$ there is nothing left to prove, so suppose $|\Gamma| = |\Gamma''|$ and let $\Gamma'' = \Gamma_{\mathsf{odd}}, \Gamma_{\mathsf{even}}$ where Γ_{odd} contains only channel with odd polarity and Γ_{even} contains only channels with even polarity. From the hypothesis $P_m \not\xrightarrow{\tau}$ and $\Gamma'' \vdash P_m$ and Lemma A.22 we deduce that $|\Gamma_{\mathsf{odd}}| \leq |\Gamma_{\mathsf{even}}|$. From the hypothesis $|\Gamma| < \top$ we also know that $\mathsf{dom}(\Gamma_{\mathsf{odd}})$ contains at least one channel. Let $a_1, \ldots, a_k$ be the channels in $\mathsf{dom}(\Gamma_{\mathsf{odd}})$ such that $|\Gamma_{\mathsf{odd}}(a_i)| = |\Gamma_{\mathsf{odd}}|$, that is the a_i are the channels with highest priority in Γ''. Then it must be the case

$$P_m \equiv \texttt{new}\ \tilde{a}\ \texttt{in} \left(Q \mid \prod_{i=1..k} R_i \right)$$

where each R_i is blocked on a_i, because no a_i can be blocked by another channel with lower (or equal) priority. Using the hypothesis that $\Gamma'' \fatsemi P_m$ we know that there exists a Γ' and k labels λ_i each of the form $a_i\texttt{?}\mathsf{v}_i$ or $a_i\texttt{!}\mathsf{v}_i$ such that

$$P_m \xmapsto{\lambda_1} \cdots \xmapsto{\lambda_k} P'$$

and $\Gamma' \fatsemi P'$ is an interactive typed process where none of the a_i occurs in Γ'. Also, by Definition 3.12 we know that every output action can only remove channels from processes, and every input action can only add channels with priority strictly lower than $|\Gamma_{\mathsf{odd}}|$. Since the a_i were *all* the channels with priority $|\Gamma_{\mathsf{odd}}|$, we conclude $|\Gamma''| < |\Gamma'|$. □

Interactivity is then a consequence of the fact that the priority of any given channel is at finite distance from that of the whole type environment used for typing a process.

Theorem A.27 (Theorem 3.13)**.** *Let $\Gamma \fatsemi P$ be an interactive typed process such that $\Gamma \vdash P$ and $a \in \mathsf{fn}(P)$. Then*

$$P \xmapsto{\lambda_1} P_1 \xmapsto{\lambda_2} \cdots \xmapsto{\lambda_n} P_n$$

for some $\lambda_1, \ldots, \lambda_n$ such that $a \notin \mathsf{fn}(P_n)$.

Proof. We proceed by induction on $|\Gamma(a)| - |\Gamma|$. By Lemma A.26, we know that there exist $\lambda_1, \ldots, \lambda_m$ and Γ'' such that

$$P \xmapsto{\lambda_1} \cdots \xmapsto{\lambda_m} P'$$

and $\Gamma'' \fatsemi P'$ is an interactive typed process and $|\Gamma| < |\Gamma''|$. We have two possibilities. If $|\Gamma(a)| < |\Gamma''|$, then we conclude $a \notin \mathsf{fn}(P')$. If $|\Gamma''| \leq |\Gamma(a)|$, then $|\Gamma(a)| - |\Gamma''| < |\Gamma(a)| - |\Gamma|$ and we conclude by induction hypothesis. □