Deadlock and lock freedom in the linear π-calculus

Luca Padovani

Deadlock and lock freedom in the linear π-calculus

Luca Padovani – *Dipartimento di Informatica, Università di Torino, Italy*

***Abstract*—We study two refinements of the linear π-calculus that ensure *deadlock freedom* (the absence of stable states with pending linear communications) and *lock freedom* (the eventual completion of pending linear communications). The simple setting given by the linear π-calculus allows us to devise a new form of channel polymorphism that affects significantly the accuracy of the type systems, in particular with respect to recursive processes communicating in cyclic network topologies.**

I. Introduction

The linear π-calculus [18] is a resource-aware model of communicating processes that distinguishes between linear and unlimited channels. Unlimited channels can be used without restrictions, while linear channels are meant to be used for *exactly one* communication. This intrinsic restriction is rewarded by several benefits, including specialized behavioral equivalences for reasoning about communication optimizations, the efficient implementation of linear channels, and the fact that communications on linear channels enjoy desirable properties such as determinism and confluence. The value of these benefits is amplified given that a significant fraction of channels in several actual systems happen to be linear.

From an operational standpoint, the linear π-calculus only guarantees that well-typed processes never communicate twice on the same linear channel. In practice, one may be interested in stronger guarantees, such as *deadlock freedom* [16] – the absence of stable states with pending communications on linear channels – or *lock freedom* [14], [19] – the possibility to complete pending communications on linear channels. A paradigmatic example of deadlock is illustrated by the process

$$a?(x).b!\langle x\rangle \mid b?(y).a!\langle y\rangle \tag{1}$$

where the left subprocess forwards on b the message x received from a, and the right subprocess forwards on a the message y received from b. The process (1) is well typed when a and b are linear channels, but none of the pending communications on a and b can complete because of the mutual dependencies between corresponding inputs and outputs. An example of lock which is not a deadlock is illustrated by the process

$$c!\langle a\rangle \mid *c?(x).c!\langle x\rangle \mid a!\langle 1984\rangle \tag{2}$$

where one occurrence of the linear channel a is repeatedly sent over the unlimited channel c but never used for receiving the 1984 message. Note that (2), unlike (1), reduces forever, but the pending communication on a cannot be completed.

In this work we propose two refinements of the linear π-calculus such that well-typed processes are (dead)lock free. The techniques we put forward have been inspired by previous ideas presented in [14]–[16], [19], except that by narrowing the focus on the linear π-calculus we obtain type systems that are not just technically simpler but also more accurate in establishing (dead)lock freedom of relevant processes.

In a nutshell, we consider channel types of the form $p^{\iota}[t]^{n,m}$ where the *polarity* p (either input `?` or output `!`), the type of message (t), and the *multiplicity* (unlimited ω or linear 1), are just like in the linear π-calculus. In addition, we annotate channel types with an integer number n called *priority* and a natural number m called *rank*. We use *priorities* to enforce an order on the use of channels, so that when a process owns two or more channels at the same time, channels with higher priority must be used *before* channels with lower priority (beware that we adopt the convention that "higher priority" means "smaller number"). For example, an input $u?(x).P$ is well typed if u has higher priority than all the channels occurring in P and an output $u!\langle v\rangle$ is well typed if u has higher priority than v. This mechanism makes (1) ill typed: it is not possible to assign two priorities h and k to a and b, for the structure of the process requires the simultaneous satisfiability of the two constraints $h < k$ and $k < h$. The *rank* is an upper bound to the number of times a channel can be sent in a message: $u!\langle x\rangle$ is well typed if the rank of x is strictly positive; each time a channel is sent as a message, its rank decreases by 1; a channel with null rank cannot be sent in a message and must be used for performing a communication. This mechanism makes (2) ill typed regardless the rank of a, because a is sent on c infinitely many times.

The technique described thus far prevents deadlocks (if we just consider priorities) and locks (if we also consider ranks). Unfortunately, as observed in [16], it rejects most recursive processes. For example, the usual encoding of the factorial below where, for convenience, we have annotated linear names with their priority, is ill typed:

$$\begin{array}{l} *\mathit{fact}?(x, y^h). \\ \quad \texttt{if } x = 0 \texttt{ then } y^h!\langle 1\rangle \\ \quad \texttt{else } (\nu a^k)(\mathit{fact}!\langle x-1, a^k\rangle \mid a^k?(z).y^h!\langle x \times z\rangle) \end{array} \tag{3}$$

Since the newly created channel a is used in the same position as y in the recursive invocation of fact, it should be the case that a and y have the same type hence the same priority $h = k$. This clashes with the input on a that blocks the output on y, requiring $k < h$. We observe a symmetric phenomenon in

$$*\mathit{stream}?(x, y^h).(\nu a^k)(y^h!\langle x, a^k\rangle \mid \mathit{stream}!\langle x+1, a^k\rangle) \tag{4}$$

which generates a stream of integers. Here too a and y are used in the same position and should have the same priority $h = k$, but the output $y!\langle x, a\rangle$ also requires $h < k$.

We need some way to overcome these difficulties. Let us digress for a moment from fact and stream and consider

$$F_1 \stackrel{\text{def}}{=} c?(x^h).y^k!\langle x^h\rangle \qquad \text{and} \qquad F_2 \stackrel{\text{def}}{=} c?(x^h, y^k).y^k!\langle x^h\rangle$$

both of which forward a linear channel x received from c to the linear channel y. In both cases it must be $k < h$, but there is a key difference between F_1 and F_2: the priority of x in F_1 has a fixed lower bound $k+1$ because y is free, while the priority of x in F_2 is arbitrary, provided that $k < h$, because y is bound. Rephrased in technical jargon, c is *monomorphic* in F_1 and *polymorphic* in F_2 with respect to priorities. The fact that a channel like c is monomorphic or polymorphic depends on the presence or absence of free linear channels in the continuation that follows the input(s) on c. Usually this information is not inferrable solely from the type of c, but it turns out that we can readily approximate it in some cases: in the linear π-calculus a replicated process cannot have free linear channels because it is not known how many times it will run. So, unlimited channels that are *always* used for replicated inputs (*i.e.*, *replicated channels* [18]), are polymorphic. This is a very convenient circumstance, because replicated channels are the prime mechanism for implementing recursion.

Now, going back to *fact* and *stream*, we see that they are replicated channels, that is they are polymorphic. This means that the mismatches between the priorities h of y and k of a in (3) and (4) can be compensated by this form of polymorphism and these processes can be declared well typed.

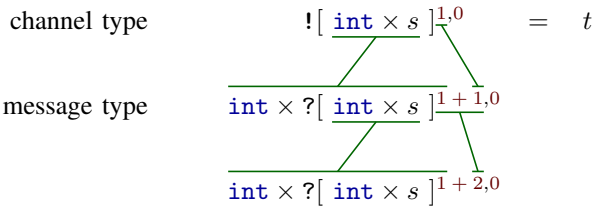
The interplay between recursion and polymorphism leads to a technical problem. Recall that in (4) there are two occurrences of a, one in $\mathit{stream}!\langle x+1, a\rangle$ having the same type as y but lower priority and another in $y!\langle x, a\rangle$ with a similar type, but opposite (input) polarity. Overall we realize that the types t of y and t_1 of the second occurrence of a should satisfy the equations

$$t = \mathtt{!}[\mathtt{int} \times t_1]^{1,0} \qquad \begin{array}{l} t_1 = \mathtt{?}[\mathtt{int} \times t_2]^{2,0} \\ t_2 = \mathtt{?}[\mathtt{int} \times t_3]^{3,0} \\ \vdots \end{array} \tag{5}$$

(the specific choice of 1 as priority in t and of $i+1$ as priority in t_i is irrelevant for the issue being discussed). The problem is that t_1 is not a regular type and therefore cannot be finitely represented by means of the well-known μ notation [6]. To recover a finite representation for t, we *relativize* priorities. In particular, we postulate that t should satisfy the equations

$$t = \mathtt{!}[\mathtt{int} \times s]^{1,0} \qquad s = \mathtt{?}[\mathtt{int} \times s]^{1,0}$$

and that messages sent on a channel of type t must have type $\mathtt{int} \times s$ (as written in t) but with priorities shifted by 1 (the priority of t). Below we illustrate such shifting, which turns relative priorities (within channel types) into absolute ones:



Note that $\mathtt{int}$ is not subject to any shifting because it is not a linear type and that overall the channel types in this sequence are the same $t_1, t_2, \ldots$ that we initially guessed in (5), except that now s is regular and hence finitely representable.

Summary of contributions. We strengthen the notion of *linearity* in the linear π-calculus by defining two type systems ensuring that linear channels are used *exactly* once as opposed to *at most* once. In this setting, we improve the accuracy of similar type systems [14]–[16] thanks to an original form of channel polymorphism that allows us to deal with recursive processes interleaving actions on different linear channels and communicating in cyclic network topologies. Such configurations arise often in the implementation of parallel and distributed algorithms or in session-based systems, for which our type systems provide unprecedented accuracy at the cost of minimal annotations.

Outline. In Section II we quickly review syntax and semantics of the π-calculus and give formal definitions of deadlock and lock freedom. The type systems are described in Section III and illustrated on several examples that highlight the features and accuracy of our approach. Section IV contains a more detailed and technical comparison with related work and particularly with [14]–[16]. Section V concludes and hints at ongoing and future developments. Additional technical material and proofs of the results can be found in the appendix.

II. Language

We use integer numbers $h, k, \ldots, m, n, \ldots$, a countable set of *variables* $x, y, \ldots$, and a countable set of *channels* $a, b, \ldots$; *names* $u, v, \ldots$ are either channels or variables. *Expressions* $e, \ldots$ and *processes* $P, Q, \ldots$ are defined by:

$$\begin{array}{l} e ::= n \mid u \mid (e,e) \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \\ P ::= \mathbf{0} \mid u?(x).P \mid *u?(x).P \mid u!\langle e\rangle \mid (P \mid Q) \mid (\nu a)P \mid \\ \qquad \mathtt{let}\ x,y = e\ \mathtt{in}\ P \mid \mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}} \end{array}$$

Expressions are integers, names, pairs of expressions, or expressions injected using either $\mathtt{inl}$ or $\mathtt{inr}$. *Values* v, w, $\ldots$ are expressions without variables. Processes are the usual terms of the π-calculus enriched with pattern matching for pairs and injected values. In particular, $\mathtt{let}\ x,y = e\ \mathtt{in}\ P$ deconstructs the value of e, which must be a pair, and continues as P where x and y have been respectively replaced by the first and second component of the pair. A process $\mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ evaluates e, which must result into a value $(i\ \mathsf{v})$ where $i \in \{\mathtt{inl}, \mathtt{inr}\}$, and continues as P_i where x_i has been replaced by v. Binders are as expected, in particular $\mathtt{let}\ x,y = e\ \mathtt{in}\ P$, binds both x and y in P, and $\mathtt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ binds x_i in P_i. The notions of *free* and *bound names* of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are defined consequently. In the following we write $(\nu\tilde{a})$ for sequences of restrictions and we use polyadic inputs $u?(x_1, \ldots, x_n)$ as an abbreviation for monadic input of (possibly nested) pairs followed by (possibly nested) pair deconstructions using $\mathtt{let}$'s. For example, $u?(x,y).P$ abbreviates $u?(z).\mathtt{let}\ x,y = z\ \mathtt{in}\ P$ for some fresh z.

The operational semantics is defined by a combination of *structural congruence* $\equiv$ and a *reduction relation* $\rightarrow$. The former is the same as in the π-calculus, except that we do *not* include the law $*P \equiv P \mid *P$ because we treat replicated inputs in a special way. Reduction is defined by the rules below

$$\begin{aligned} a!\langle \mathsf{v} \rangle \mid a?(x).P &\rightarrow P\{\mathsf{v}/x\} \\ a!\langle \mathsf{v} \rangle \mid *a?(x).P &\rightarrow P\{\mathsf{v}/x\} \mid *a?(x).P \\ \texttt{let}\ x,y = \mathsf{v},\mathsf{w}\ \texttt{in}\ P &\rightarrow P\{\mathsf{v},\mathsf{w}/x,y\} \\ \texttt{case}\ k\ \mathsf{v}\ \{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}} &\rightarrow P_k\{\mathsf{v}/x_k\} \end{aligned}$$

and closed by *reduction contexts* $\mathscr{C} ::= [\,] \mid (\mathscr{C} \mid P) \mid (\nu a)\mathscr{C}$ and structural congruence. The rules are unremarkable and $P\{\mathsf{v}/x\}$ is the usual capture-avoiding substitution of v in place of the free occurrences of x in P. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$ and $P \not\rightarrow$ if there is no Q such that $P \rightarrow Q$. We say that P is *stable* if $P \not\rightarrow$.

To formulate deadlock and lock freedom, we need a few predicates that describe the pending communications of a process P with respect to some channel a:

$$\begin{aligned} \mathsf{in}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[a?(x).Q] \wedge a \notin \mathsf{bn}(\mathscr{C}) \\ *\mathsf{in}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[*a?(x).Q] \wedge a \notin \mathsf{bn}(\mathscr{C}) \\ \mathsf{out}(a,P) &\overset{\text{def}}{\iff} P \equiv \mathscr{C}[a!\langle e \rangle] \wedge a \notin \mathsf{bn}(\mathscr{C}) \\ \mathsf{sync}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \vee *\mathsf{in}(a,P)) \wedge \mathsf{out}(a,P) \\ \mathsf{wait}(a,P) &\overset{\text{def}}{\iff} (\mathsf{in}(a,P) \vee \mathsf{out}(a,P)) \wedge \neg\mathsf{sync}(a,P) \end{aligned}$$

In words, $\mathsf{in}(a,P)$ holds if there is a sub-process Q within P that is waiting for a message from a (it is similar to the *live* predicate in [2]). Note that, by definition of reduction context, the input is not guarded by other actions. The condition $a \notin \mathsf{bn}(\mathscr{C})$ means that a occurs free in P. The predicates $*\mathsf{in}(a,P)$ and $\mathsf{out}(a,P)$ are similar, but they regard replicated inputs and outputs. Whenever one of $\mathsf{in}(a,P)$ or $\mathsf{out}(a,P)$ holds, there is some pending input/output operation. On the contrary, we do not interpret $*\mathsf{in}(a,P)$ as a pending communication, because we do not require that a replicated input process should run infinitely often. This discussion explains the $\mathsf{sync}(a,P)$ and $\mathsf{wait}(a,P)$ predicates: the first one denotes the fact that there are pending input/output operations on a, but a synchronization on a is *immediately* possible; the second one denotes the fact that there is a pending output or a pending non-replicated input on a, but no immediate synchronization on a is possible. Note that pending outputs trigger the wait predicate even when they regard unlimited channels. This is because such outputs may carry messages containing linear channels and in order to guarantee deadlock freedom we must be sure that *all* outputs, including those on unlimited channels, can be completed and the messages delivered. For this reason we treat unlimited channels as replicated channels and enforce input receptiveness for them. On the contrary, replicated inputs, which can be though of as persistently available servers, do not trigger the wait predicate because, by definition, they are not allowed to have free linear channels (the type system will enforce this property).

We define deadlock freedom as the absence of stable states with pending communications and lock freedom as the ability to eventually complete any pending communication. Formally:

Definition II.1 (deadlock [16] and lock freedom [19])**.**

1) We say that P is *deadlock free* if for every Q such that $P \rightarrow^* (\nu\tilde{a})Q \not\rightarrow$ we have $\neg\mathsf{wait}(a,Q)$ for every a.
2) We say that P is *lock free* if for every Q such that $P \rightarrow^* (\nu\tilde{a})Q$ and $\mathsf{wait}(a,Q)$ there exists R such that $Q \rightarrow^* R$ and $\mathsf{sync}(a,R)$.

For example, both $(\nu a)a!\langle 1984 \rangle$ and $(\nu a)a?(x).P$ are deadlocks, whereas $(\nu a)*a?(x).P$ is lock free. In principle, we should also require that let and case processes are always able to reduce and that expressions in top-level outputs are always evaluated. These properties are already guaranteed (for closed processes) by conventional type systems for the π-calculus, so we keep Definition II.1 focused on pending communications. Note that lock freedom implies deadlock freedom, but not viceversa (process (2) is deadlock free but not lock free).

Example II.2. Many parallel/distributed algorithms (Jacobi and Gauss-Seidel, leader election, vertex coloring, just to mention a few) are implemented using iterative processes where at each iteration, each process exchanges some messages with its neighbors using *full-duplex* communication. This means that processes simultaneously send messages to each other to maximize parallelism. For instance, the process

$$Node_{\mathrm{A}} \overset{\text{def}}{=} *c_{\mathrm{A}}?(x,y).(\nu a)(x!\langle a \rangle \mid y?(z).c_{\mathrm{A}}!\langle a,z \rangle)$$

uses a channel x for sending messages to, and another channel y for receiving messages from, a neighbor process. Each message sent on x carries a payload (omitted) as well as a continuation channel a with which $Node_{\mathrm{A}}$ communicates with its neighbor at the next iteration. Symmetrically, each message received from y contains the neighbor's payload (omitted) and continuation z. The use of fresh continuations at each iteration makes sure that messages are received in the desired order. An alternative modeling using *half-duplex* communication is

$$Node_{\mathrm{B}} \overset{\text{def}}{=} *c_{\mathrm{B}}?(x,y).y?(z).(\nu a)(x!\langle a \rangle \mid c_{\mathrm{B}}!\langle a,z \rangle)$$

where the process first waits for the message from its neighbor, and only then sends its own information. It may then be relevant to understand whether a given configuration consisting of mixed nodes, some of type A and others of type B, is lock free. Below we represent three of them

$$\begin{aligned} L_1(\mathrm{X}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,e \rangle \\ L_2(\mathrm{X},\mathrm{Y}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,f \rangle \mid c_{\mathrm{Y}}!\langle f,e \rangle \\ L_3(\mathrm{X},\mathrm{Y},\mathrm{Z}) &\overset{\text{def}}{=} Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid c_{\mathrm{X}}!\langle e,f \rangle \mid c_{\mathrm{Y}}!\langle g,e \rangle \mid c_{\mathrm{Z}}!\langle f,g \rangle \end{aligned}$$

where each of X, Y, and Z can be either A and B to yield a different configuration. For example, we have

$$\begin{aligned} L_2(\mathrm{B},\mathrm{B}) \rightarrow^* Node_{\mathrm{A}} \mid Node_{\mathrm{B}} \mid f?(z).(\nu a)(e!\langle a \rangle \mid c_{\mathrm{B}}!\langle a,z \rangle) \\ \mid e?(z).(\nu b)(f!\langle a \rangle \mid c_{\mathrm{B}}!\langle b,z \rangle) \not\rightarrow \end{aligned}$$

and the final configuration satisfies the in predicate but not out for both e and f, namely $L_2(\mathrm{B},\mathrm{B})$ is *not* lock free. We will see later that a given configuration L_i is lock free if and only if at least one of its nodes is of type A. ■

III. Type system

Notation. *Polarities* p, q, ... are subsets of $\{\mathtt{?}, \mathtt{!}\}$. We abbreviate $\{\mathtt{?}\}$ with $\mathtt{?}$, $\{\mathtt{!}\}$ with $\mathtt{!}$, and $\{\mathtt{?}, \mathtt{!}\}$ with $\mathtt{\#}$ and we say that $\emptyset$ and $\mathtt{\#}$ are *even polarities*. *Multiplicities* ι, ... are either 1 or ω. We also use a countable set of *type variables* α, ... *Types* t, s, ... are defined by the grammar below:

$$t \ ::= \ \mathtt{int} \mid \alpha \mid t \times s \mid t \oplus s \mid p^{\iota}[t]^{n,m} \mid \mu\alpha.t$$

The types $\mathtt{int}$, $t \times s$, and $t \oplus s$ respectively denote integers, pairs inhabited by values (v, w) where v has type t and w has type s, and the disjoint sum of t and s inhabited by values $(\mathtt{inl}\ \mathsf{v})$ when v has type t or $(\mathtt{inr}\ \mathsf{w})$ when w has type s. The type $p^{\iota}[t]^{n,m}$ denotes a channel to be used with polarity p and multiplicity ι for exchanging messages of type t. The polarity determines the operations allowed on the channel: $\emptyset$ means none, $\mathtt{?}$ means input, $\mathtt{!}$ means output, and $\mathtt{\#}$ means both. The multiplicity determines how many times the channel can or must be used: 1 means that the channel must be used exactly once (for each element in p), while ω means that the channel can be used any number of times. The numbers n and m are respectively the *priority* and *rank* of the channel. As we have anticipated in Section I, priorities are used for imposing an order on the usage of channels, so that an action on a channel with low priority cannot block another action on a channel with higher priority. Recall that smaller numbers denote higher priorities, so for example an input on a channel with priority 4 can block an input on a channel with priority 5, but not viceversa. The rank of a channel is an upper limit to the number of times the channel can be sent in a message. For example, a channel with rank 3 can be sent *at most* three times in a message, but it can be used at any time for an input/output operation according to its polarity. A channel with rank 0 cannot be sent in a message. Ranks are always non-negative. We omit priority and rank when $\iota = \omega$ and the multiplicity when it is 1. Also, we often omit the rank when it is zero. We use type variables and μ's for building recursive types. Notions of free and bound type variables are as expected.

Contractiveness. For simplicity, we forbid non-contractive types in which recursion variables are not guarded by a channel type. For example, $\mu\alpha.\alpha$ and $\mu\alpha.(\alpha \times \alpha)$ are illegal while $\mu\alpha.p^{\iota}[\alpha]^{n,m}$ is allowed. In this way, we will be able to define functions by induction on the structure of types, when such functions do not recur within channel types. Contractiveness can be weakened without compromising the results that follow, but some functions on types must be defined more carefully. The formal definition of contractiveness can be found in long version of the paper. We identify two types modulo renaming of bound type variables and if they have the same infinite unfolding, that is if they denote the same (regular) tree [6]. In particular, we have $\mu\alpha.t = t\{\mu\alpha.t/\alpha\}$.

Priorities and ranks. The priority of a channel gives a measure to the urgency with which the channel must be used: the higher the priority is, the sooner the channel must be used. This measure can be extended from channels to arbitrary types. The idea is that the priority of a type is the highest of the priorities of the channel types occurring therein. To compute the priority of a type, we define an auxiliary function $|\cdot|$ such that $|t|$ is an element of $\mathbb{Z} \cup \{\bot, \top\}$ where $\bot < n < \top$ for every $n \in \mathbb{Z}$. The function is defined inductively thus:

$$|t| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } t = p^{\omega}[s] \text{ and } \mathtt{?} \in p \\ n & \text{if } t = p[s]^{n,m} \text{ and } p \neq \emptyset \\ \min\{|t_1|, |t_2|\} & \text{if } t = t_1 \times t_2 \text{ or } t = t_1 \oplus t_2 \\ \top & \text{otherwise} \end{cases} \tag{6}$$

Intuitively, unlimited channels with input polarity have the highest priority $\bot$ (first equation) because their use cannot be postponed by any means to guarantee input receptiveness. Numbers, unlimited channels with only output polarity, and linear channels with empty polarity can be used with the lowest priority $\top$ (last equation) because they do not affect (dead)lock freedom in any way. Linear channels with pending operations must be used according to their priority (second equation). The priority of compound types is the highest (numerically speaking, the minimum) of the priorities of the component types (third equation). For instance, $|\mathtt{!}[t]^{3,1} \times \mathtt{?}[s]^{2,4}| = \min\{|\mathtt{!}[t]^{3,1}|, |\mathtt{?}[s]^{2,4}|\} = \min\{3, 2\} = 2$.

We say that a (value with) type t is *unlimited* if $|t| = \top$, that it is *linear* if $|t| \in \mathbb{Z}$, that it is *relevant* if $|t| = \bot$.

We define another auxiliary function $\$^{h,k}$ to *shift* priorities and ranks: $\$^{h,k}\, t$ has the same structure of t, except that all priorities/ranks in the topmost linear channel types of t have been transposed by h and k respectively. Formally:

$$\$^{h,k}\, t \stackrel{\text{def}}{=} \begin{cases} p[s]^{n+h,m+k} & \text{if } t = p[s]^{n,m} \text{ and } p \neq \emptyset \\ (\$^{h,k}\, t_1) \times (\$^{h,k}\, t_2) & \text{if } t = t_1 \times t_2 \\ (\$^{h,k}\, t_1) \oplus (\$^{h,k}\, t_2) & \text{if } t = t_1 \oplus t_2 \\ t & \text{otherwise} \end{cases} \tag{7}$$

As an example, we have $\$^{1,1}\,(\mathtt{int} \times \mathtt{?}[\mathtt{int}]^3) = (\$^{1,1}\,\mathtt{int}) \times (\$^{1,1}\,\mathtt{?}[\mathtt{int}]^3) = \mathtt{int} \times \mathtt{?}[\mathtt{int}]^{4,1}$. Note that positive/negative shifting respectively corresponds to decreasing/increasing the priority. We write $\$^n$ for $\$^{n,0}$.

A key property used in the proof of subject reduction is that shifting behaves like the identity on unlimited types:

Lemma III.1. *If t is unlimited, then $\$^{h,k}\, t = t$ for any h, k.*

Monomorphic and polymorphic channels. A distinguishing feature of our type systems is that message types are "relative" to the priority of the channels on which they travel. We use shifting to compute the absolute type of messages that can be sent on a channel. For example, a channel whose type is $\mathtt{!}[s]^h$ accepts messages of type $\$^h\, s$, that is the type s (in the type t of the channel) where all top-level priorities have been shifted by h (the priority of the channel). As we have seen in Section I, this feature allows us to provide finite representations for recursive types having infinitely many different priorities. For example, a recursive type that satisfies the equation $t = \mathtt{!}[t]^1$ denotes a linear channel on which it is possible to send a message of type $\mathtt{!}[t]^2$. In turn, on such a channel it is possible to send a message of type $\mathtt{!}[t]^3$, and so on and so forth.

Shifting is also used to implement polymorphism. For example, an unlimited (*i.e.*, polymorphic) channel of type

$!^{\omega}[s]$ accepts messages of type $\$^h s$ *for any* h. So, for linear (*i.e.*, monomorphic) channels, the shifting is fixed and determined by the priority of the channel, while for unlimited (*i.e.*, polymorphic) channels, the shifting is arbitrary.

It should be noted that this particular way of realizing polymorphism, as opposed to, for example, devising "priority variables", is sometimes more restrictive than necessary. For example, by virtue of shifting, a channel of type

$$!^{\omega}[?[\mathtt{int}]^1 \times !^{}[?[\mathtt{int}]^1]^0]$$

accepts messages of type

$$\$^h\,(?[\mathtt{int}]^1 \times !^{}[?[\mathtt{int}]^1]^0) = ?[\mathtt{int}]^{h+1} \times !^{}[?[\mathtt{int}]^1]^h$$

for every h, but not messages of type

$$?[\mathtt{int}]^{h+2} \times !^{}[?[\mathtt{int}]^1]^h$$

where the first component of the pair has been shifted by $h+1$ and the second component of the pair only by h. A message of the latter type could be safely sent on the unlimited channel c appearing in the process F_2 of Section I. We leave more general forms of polymorphism as future work.

Type environments. We check that processes are well typed in *type environments* $\Gamma, \ldots$, which are finite maps from names to types written $u_1 : t_1, \ldots, u_n : t_n$. We write $\mathsf{dom}(\Gamma)$ for the *domain* of Γ, namely the set of names for which there is a binding in Γ, and Γ, Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. In general we need a more flexible way of composing type environments, taking into account the linearity and relevance of types and the fact that we can split channel types by distributing different polarities to different processes. Following [18], we define a partial composition operator $+$ between types, thus:

$$\begin{array}{rll} t + t = t & & \text{if } t \text{ is unlimited} \\ p^{\omega}[t] + q^{\omega}[t] = (p \cup q)^{\omega}[t] & & \\ p[s]^{n,h} + q[s]^{n,k} = (p \cup q)[s]^{n,h+k} & & \text{if } p \cap q = \emptyset \end{array} \tag{8}$$

Informally, unlimited types compose with themselves without restrictions. The composition of two unlimited/relevant channel types has the union of their polarities. Two linear channel types can be composed only if they have the same priority and disjoint polarities, and the composition has the union of their polarities and the sum of their ranks. This partial loss of information about which component had which rank does not compromise the soundness of the type system. We extend $+$ to type environments, thus:

$$\begin{array}{rll} \Gamma + \Gamma' \stackrel{\text{def}}{=} \Gamma, \Gamma' & & \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset \\ (\Gamma, u : t) + (\Gamma', u : s) \stackrel{\text{def}}{=} (\Gamma + \Gamma'), u : t + s & & \end{array} \tag{9}$$

Note that $\Gamma + \Gamma'$ is undefined if there is $u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma')$ such that $\Gamma(u) + \Gamma'(u)$ is undefined and that $\mathsf{dom}(\Gamma + \Gamma') = \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma')$. We write $\mathsf{un}(\Gamma)$ if all the types in the range of Γ are unlimited. We let $|\Gamma|$ denote the maximum priority of the types in the range of Γ, that is $|\Gamma| = \min\{|\Gamma(u)| \mid u \in \mathsf{dom}(\Gamma)\}$.

TABLE 1: Type rules for expressions and processes.

Expressions

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash n : \mathtt{int}}\ [\text{T-CONST}] \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma, u : t \vdash u : t}\ [\text{T-NAME}]$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma' \vdash e' : s}{\Gamma + \Gamma' \vdash (e, e') : t \times s}\ [\text{T-PAIR}] \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{inl}\ e : t \oplus s}\ [\text{T-INL}] \qquad \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inr}\ e : t \oplus s}\ [\text{T-INR}]$$

Processes

$$\frac{\Gamma, x : \$^{n,0}\, t \vdash_k P \qquad n < |\Gamma|}{\Gamma + u : ?[t]^{n,m} \vdash_k u?(x).P}\ [\text{T-IN}] \qquad \frac{\Gamma, x : t \vdash_k P \qquad \mathsf{un}(\Gamma)}{\Gamma + u : ?^{\omega}[t] \vdash_k *u?(x).P}\ [\text{T-IN*}]$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_k \mathbf{0}}\ [\text{T-IDLE}] \qquad \frac{\Gamma \vdash e : \$^{n,k}\, t \qquad 0 < |t|}{\Gamma + u : !^{}[t]^{n,m} \vdash_k u!\langle e \rangle}\ [\text{T-OUT}] \qquad \frac{\Gamma \vdash e : \$^{n,k}\, t \qquad \bot < |t|}{\Gamma + u : !^{\omega}[t] \vdash_k u!\langle e \rangle}\ [\text{T-OUT*}]$$

$$\frac{\Gamma, a : p^{\iota}[t]^{n,m} \vdash_k P \qquad p \text{ even}}{\Gamma \vdash_k (\nu a) P}\ [\text{T-NEW}] \qquad \frac{\Gamma \vdash e : t \times s \qquad \Gamma', x : t, y : s \vdash_k P}{\Gamma + \Gamma' \vdash_k \mathtt{let}\ x, y = e\ \mathtt{in}\ P}\ [\text{T-LET}]$$

$$\frac{\Gamma_i \vdash_k P_i^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash_k P_1 \mid P_2}\ [\text{T-PAR}] \qquad \frac{\Gamma \vdash e : t \oplus s \qquad \Gamma', x : t \vdash_k P \qquad \Gamma', y : s \vdash_k Q}{\Gamma + \Gamma' \vdash_k \mathtt{case}\ e\ \{\mathtt{inl}\ x \Rightarrow P, \mathtt{inr}\ y \Rightarrow Q\}}\ [\text{T-CASE}]$$

Typing rules. The typing rules for expressions and processes are presented in Table 1. A judgment $\Gamma \vdash e : t$ denotes that e is well typed and has type t in Γ and a judgment $\Gamma \vdash_k P$ denotes that P is well typed in Γ. The two type systems for deadlock and lock freedom are very similar, to the point that we devise just one set of rules with a parameter k which is either 0 (for deadlock freedom) or 1 (for lock freedom). This parameter affects only the rules for outputs, and is always propagated unchanged by all rules.

Rules for expressions as well as [T-IDLE], [T-PAR], [T-LET], and [T-CASE] are standard for the linear π-calculus, with the usual splitting of environments and the requirement that unused environments in axioms must be unlimited.

Rule [T-IN] is used for typing linear inputs $u?(x).P$, where u must have type $?[t]^{n,m}$. The continuation P is typed in an environment where the input polarity of u has been removed and the received message x has been added. Note that the type of x is not just t, but t shifted by n, consistently with the relative interpretation of priorities of message types. The rank m is irrelevant since u is used for an input operation, not as the content of a message. The condition $n < |\Gamma|$ verifies that the input on u does not block operations on other channels with equal or higher priority. In particular, Γ cannot contain relevant channels. Below are some typical examples of ill-typed processes that violate this condition:

- $a : ?[\mathtt{int}]^1, b : !^{}[\mathtt{int}]^0 \vdash_k a?(x).b!\langle x \rangle$ is not derivable because $1 \not< 0$: the input on a blocks the output on b, but b has higher priority than a;
- $a : \#[\mathtt{int}]^h \vdash_k a?(x).a!\langle x \rangle$ is a degenerate case of the previous example, where the input on a blocks the very

output that should synchronize with it. Note that this process is well typed in the traditional linear π-calculus because $\#[\mathtt{int}] = ?[\mathtt{int}] + ![\mathtt{int}]$;

- $a : ?[\mathtt{int}]^h, c : ?^\omega[\mathtt{int}] \vdash_k a?(x).{*}c?(y)$ is not derivable because $|?^\omega[\mathtt{int}]| = \bot$. To guarantee input receptiveness, replicated inputs cannot be guarded by any operation on linear channels.

Rule [T-OUT] is used for typing linear outputs on channels of type $![t]^{n,m}$. The type of the message e must be t (as specified in the type of the channel u) shifted by n again since t is relative to the priority of u. The rank is shifted by 1, but only for lock freedom ($k = 1$). The meaning is that, by sending e as a message, one unit from every finite rank in the type of e is consumed. This prevents the degenerate phenomenon of infinite delegation that we have seen in (2). The condition $0 < |t|$ verifies that the priority of the message is lower (*i.e.*, numerically greater) than that of the channel on which it travels. Below are two examples:

- The judgment $a : ![?[\mathtt{int}]^{1,0}]^{2,0}, b : ?[\mathtt{int}]^{3,1} \vdash_1 a!\langle b\rangle$ is derivable because $?[\mathtt{int}]^{3,1} = \$^{2,1}\, ?[\mathtt{int}]^{1,0}$. Note in particular that the channel to be sent on a must have rank 0, which is in fact the residual rank of b after 1 unit is consumed as the effect of b being sent in a message.
- The judgment $a : ?[![\mathtt{int}]^{2,0}]^{0,0}, b : ![![\mathtt{int}]^{2,0}]^{1,0} \vdash_1 a?(x).b!\langle x\rangle$ is not derivable, because x received from a has null rank so it cannot be forwarded on b.
- Let $t = \mu\alpha.?[\alpha]^0$ and observe that $\#[t]^1 = ![t]^1 + ?[t]^1$. The judgment $a : \#[t]^1 \vdash_0 a!\langle a\rangle$ is not derivable, despite the message a has the "right" type $?[t]^1 = \$^1\, t$, because the condition $0 < |t| = 0$ is not satisfied. According to Definition II.1, a process like $(\nu a)a!\langle a\rangle$ is deadlock.

Rule [T-IN*] is used for typing replicated inputs ${*}u?(x).P$. This rules differs from [T-IN] in three important ways. First of all, u must be an unlimited channel with input polarity. Second, the residual environment Γ must be unlimited, because the continuation P can be spawned an arbitrary number of times. Third, it may be the case that $u \in \mathsf{dom}(\Gamma)$, because $?^\omega[t] + !^\omega[t] = \#^\omega[t]$ according to (8) and $!^\omega[t]$ is unlimited. This means that replicated input processes may invoke themselves, as we have seen in many examples.

Rule [T-OUT*] is used for outputs on unlimited channels. There are two key differences with respect to [T-OUT]. First, the condition $\bot < |t|$, where t is the type of x, implies that only relevant names cannot be communicated, but a process like $a!\langle a\rangle$ is well typed when a is unlimited, for example by giving the inner a type $\mu\alpha.!^\omega[\alpha]$. Second, the type of the message need not match exactly the type t in the channel, but its priority can be shifted by an arbitrary amount n. This is the technical realization of polymorphism. In particular, each distinct output on u can shift the type of the message by a different amount, therefore allowing polymorphic recursion. We will this feature at work in later examples.

Rule [T-NEW] is used for restricting new (linear and unlimited) channels. Only channels with an even polarity can be restricted: either the channel comes with full polarity #, or with no polarity $\emptyset$ (this is only useful for subject reduction).

We write $\vdash_{\mathtt{DF}} P$ if $\emptyset \vdash_0 P$ and $\vdash_{\mathtt{LF}} P$ if $\emptyset \vdash_1 P$ and all priorities in the derivation are non-negative. Intuitively, the type system for deadlock freedom ignores ranks because infinite delegations (like in (2)) are not deadlocks, while the type system for lock freedom requires a well-founded order on priorities (hence the restriction to natural numbers) and an upper bound on delegations (hence the importance of ranks) so that in the soundness proof we are able to find a *finite* reduction that completes any pending communication.

Properties. The relative interpretation of priorities makes it possible to shift whole derivations and preserve typing. More precisely, let $\$^n\,\Gamma$ be the environment obtained by shifting all the types in the range of Γ by n, that is $(\$^n\,\Gamma)(u) = \$^n\,\Gamma(u)$ for every $u \in \mathsf{dom}(\Gamma)$. We have:

Lemma III.2 (shifting)**.** *The following properties hold:*

1) *If* $\Gamma \vdash e : t$*, then* $\$^n\,\Gamma \vdash e : \$^n\,t$.
2) *If* $\Gamma \vdash_0 P$*, then* $\$^n\,\Gamma \vdash_0 P$.
3) *If* $\Gamma \vdash_1 P$ *and* $n \geq 0$*, then* $\$^n\,\Gamma \vdash_1 P$.

Note that the converse of 3) does not hold. For example, the derivation for $u : ![\mathtt{int}]^{1,0} \vdash_1 (\nu a)(a!\langle 1984\rangle \mid a?(x).u!\langle x\rangle)$ relies on the fact that u has priority 1, for otherwise it would not be possible to prefix $u!\langle x\rangle$ with an input operation on a.

Typing is preserved by reductions and the type environment may change as a consequence of communications on linear channels, just like in the linear π-calculus [18]:

Lemma III.3 (subject reduction)**.** *If* $\Gamma \vdash_k P$ *and* $P \to P'$*, then* $\Gamma' \vdash_k P'$ *for some* Γ'.

The proof is essentially standard, except for the interesting case concerning the communication on unlimited (*i.e.*, polymorphic) channels sketched below. Consider the derivation

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{v} : \$^n\,t}{a : !^\omega[t], \Gamma \vdash_k a!\langle \mathsf{v}\rangle} \qquad \dfrac{\Gamma', x : t \vdash_k P \quad \mathsf{un}(\Gamma')}{a : ?^\omega[t], \Gamma' \vdash_k {*}a?(x).P}}{a : \#^\omega[t], \Gamma + \Gamma' \vdash_k a!\langle \mathsf{v}\rangle \mid {*}a?(x).P}$$

and observe that the actual type $\$^n\,t$ of the message v being sent does *not* match exactly the type t expected by the receiver, but is shifted by some arbitrary amount n. In order to type the reduct $P\{\mathsf{v}/x\}$, we proceed like this. First of all we shift the derivation of $\Gamma', x : t \vdash_k P$ using Lemma III.2, and obtain:

$$\$^n\,\Gamma', x : \$^n\,t \vdash_k P$$

Then, we observe that Γ' is unlimited because it is the type environment used for typing a replicated process. This means that $\$^n\,\Gamma' = \Gamma'$, since shifting is the identity on unlimited types (Lemma III.1). Therefore we deduce:

$$\Gamma', x : \$^n\,t \vdash_k P$$

Now, the actual and expected types of the message v coincide and we know from the initial derivation that $\Gamma + \Gamma'$ is defined, so we can apply the conventional substitution lemma for the linear π-calculus [18] and conclude:

$$\Gamma + \Gamma' \vdash_k P\{\mathsf{v}/x\}$$

TABLE 2: Typing derivation for $Node_{\mathrm{B}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{a : \$^{n,2}\, s \vdash a : \$^{n,2}\, s \qquad 0 < m}{x : t, a : \$^{n,2}\, s \vdash_1 x!\langle a\rangle}\ [\text{T-OUT}] \qquad \dfrac{z : \$^{m,1}\, s, a : \$^{m,1}\, t \vdash (a, z) : \$^{m,1}\, (t \times s)}{c_{\mathrm{B}} : !^{\omega}[t \times s], z : \$^{m,1}\, s, a : \$^{m,1}\, t \vdash_1 c_{\mathrm{B}}!\langle a, z\rangle}\ [\text{T-OUT*}]}{c_{\mathrm{B}} : !^{\omega}[t \times s], x : t, z : \$^{m,1}\, s, a : \#[\$^{0,1}\, s]^{m+n,3} \vdash_1 x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a, z\rangle}\ [\text{T-PAR}]}{c_{\mathrm{B}} : !^{\omega}[t \times s], x : t, z : \$^{m,1}\, s \vdash_1 (\nu a)(x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a, z\rangle)}\ [\text{T-NEW}] \qquad m < n}{c_{\mathrm{B}} : !^{\omega}[t \times s], x : t, y : s \vdash_1 y?(z).(\nu a)(x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a, z\rangle)}\ [\text{T-IN}]$$

$$\dfrac{c_{\mathrm{B}} : !^{\omega}[t \times s], x : t, y : s \vdash_1 y?(z).(\nu a)(x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a, z\rangle)}{c_{\mathrm{B}} : \#^{\omega}[t \times s] \vdash_1 *c_{\mathrm{B}}?(x, y).y?(z).(\nu a)(x!\langle a\rangle \mid c_{\mathrm{B}}!\langle a, z\rangle)}\ [\text{T-IN*}]$$

Since the type systems refine the one in [18], all the properties of the linear π-calculus (partial confluence, linear usage of channels, etc.) still hold. The added value is that the refined type systems guarantee deadlock/lock freedom.

Theorem III.4 (soundness)**.** *The following properties hold:*

1) *If* $\vdash_{\mathrm{DF}} P$*, then* P *is deadlock free.*
2) *If* $\vdash_{\mathrm{LF}} P$*, then* P *is lock free.*

The proof of 1) is shown by contradiction, for the existence of a well-typed, stable process with pending input or output operations on linear channels would imply the existence of an infinite chain of channels with strictly decreasing priorities, contradicting the fact that there are only finitely many distinct channels in the process. The proof of 2) does an induction on a *measure* that includes, among other information, priority and rank of the linear channel for which we are completing the communication (see Definition II.1(2)). The induction is well founded if so is the domain of priorities, whence the restriction to natural numbers for the priorities in the type system for lock freedom. The details can be found in the appendix.

We conclude this section showing the type systems at work on a variety of examples. All the examples are meant to highlight the distinguishing features of our type systems, in particular the key role of polymorphism for coping with recursive processes in possibly cyclic network topologies.

Example III.5. Regarding the processes in Example II.2, let

$$t \stackrel{\text{def}}{=} ![s']^{n,0} \quad \text{and} \quad s \stackrel{\text{def}}{=} ?[s']^{m,0} \quad \text{and} \quad s' \stackrel{\text{def}}{=} \mu\alpha.?[\alpha]^{m,1}$$

and observe that $s' = \$^{0,1}\, s = ?[\$^{0,1}\, s]^{m,1}$. For $Node_{\mathrm{B}}$ we obtain the typing derivation shown in Table 2 if and only if $0 < m < n$. An analogous derivation is obtained for $Node_{\mathrm{A}}$ if and only if $0 < n, m$. We deduce that $L_1(\mathrm{A})$, $L_2(\mathrm{A}, \mathrm{A})$, $L_2(\mathrm{A}, \mathrm{B})$, and $L_2(\mathrm{B}, \mathrm{A})$ are all well typed. Since these constraints on priorities are the most general ones, we also deduce that there are no derivations for $L_1(\mathrm{B})$ and $L_2(\mathrm{B}, \mathrm{B})$, suggesting that these configurations are locked. ■

Example III.6. Below is the encoding of the function that computes the n-th number in the sequence of Fibonacci. Instead of showing the typing derivation, which is conventional for the linear π-calculus, we just annotate each linear name occurring in the term with its priority and leave it to the reader to verify that such annotations are consistent with the constraints expressed in the typing rules for deadlock freedom.

$$\begin{array}{l} *\mathit{fibo}?(x, y^{0}). \\ \quad \texttt{if } x \leq 1 \texttt{ then } y^{0}!\langle x\rangle \\ \quad \texttt{else } (\nu a^{-1})(\nu b^{-2})\big(\mathit{fibo}!\langle x-1, a^{-1}\rangle \mid \mathit{fibo}!\langle x-2, b^{-2}\rangle \mid \\ \qquad\qquad b^{-2}?(z_2).a^{-1}?(z_1).y^{0}!\langle z_1 + z_2\rangle\big) \end{array}$$

Note the use of negative priorities associated with the fresh continuation channels a and b, due to the fact that the inputs on these channels block the output on the continuation y. Observe also the role of polymorphic recursion in typing this process: the two recursive invocations of *fibo* are applied to continuation channels a and b with different priorities. ■

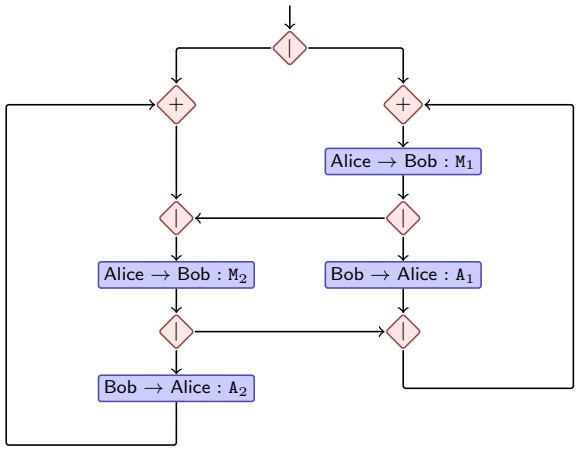


Fig. 1: Alternating Bit Protocol [9].

Example III.7. The following variation of the process $Node_{\mathrm{A}}$ gives an example of dynamic lock-free system:

$$\begin{array}{l} \mathit{Ring} \stackrel{\text{def}}{=} *c?(x^{0,0}, y^{0,0}). \\ \quad (\nu a^{1,3})\big(x^{0,0}?(z^{1,1}).(\nu b^{1,2})(c!\langle z^{1,1}, b^{1,1}\rangle \mid c!\langle b^{1,1}, a^{1,1}\rangle) \mid \\ \qquad y^{0,0}!\langle a^{1,2}\rangle\big) \end{array}$$

Each message on c creates two duplicates connected by a new channel b. As a consequence, the process $(\nu e)(\mathit{Ring} \mid c!\langle e, e\rangle)$ grows as a ring of processes, each sending a message to its neighbour, and the ring doubles its diameter each time a round of communications is completed. ■

Example III.8. Figure 1 depicts a variant of the Alternating Bit Protocol [9] in which Alice sends alternated messages

$\mathtt{M}_1$ and $\mathtt{M}_2$ to Bob, and Bob answers with corresponding acknowledgments $\mathtt{A}_1$ and $\mathtt{A}_2$. The diagram describes communications (the rectangles) between the two parties and their dependencies (the arcs), combined with merging points (+ nodes), forks (| nodes with two outgoing arcs) and joins (| nodes with two incoming arcs). The particularity of this modeling of the protocol is that Alice concurrently waits for both $\mathtt{A}_1$ and $\mathtt{A}_2$ before sending the next corresponding $\mathtt{M}_i$, but always alternating between $\mathtt{M}_1$ and $\mathtt{M}_2$ messages.

We model Bob as the process below, which is parametric in two channels x and y from which Bob respectively receives messages $\mathtt{M}_1$ and $\mathtt{M}_2$ along with the continuations on which he sends the corresponding $\mathtt{A}_i$ (we omit the actual payloads and focus on channels and their continuations):

$$\begin{aligned}&\mathsf{Bob} \stackrel{\text{def}}{=} *bob?(x^0, y^2).\\ &\quad (\nu a^{3,3} b^{5,3})\big(x^0?(\bar{x}^1).(\bar{x}^1!\langle a^{3,2}\rangle \mid \\ &\qquad\qquad y^2?(\bar{y}^3).(\bar{y}^3!\langle b^{5,2}\rangle \mid bob!\langle a^{3,1}, b^{5,1}\rangle)))\big)\end{aligned}$$

We model Alice as two parallel threads Alice_1 and Alice_2, each handling messages and acknowledgments with matching number, so that Alice_1 corresponds to the rightmost vertical flow in Figure 1, and Alice_2 to the leftmost one. To respect the protocol and prevent that two messages with the same index are sent in a row, Alice uses a third channel z (unknown to Bob) with which she synchronizes the two threads:

$$\begin{aligned}&\mathsf{Alice}_1 \stackrel{\text{def}}{=} *alice_1?(x^0, z^1).\\ &\quad (\nu a^{1,1} c^{2,1})\big(x^0!\langle a^{1,1}\rangle \mid z^1!\langle c^{2,1}\rangle \mid \\ &\qquad\qquad a^1?(\bar{x}^{3,1}).c^2?(\bar{z}^{4,1}).alice_1!\langle \bar{x}^{3,1}, \bar{z}^{4,1}\rangle\big)\end{aligned}$$

$$\begin{aligned}&\mathsf{Alice}_2 \stackrel{\text{def}}{=} *alice_2?(y^2, z^1).\\ &\quad (\nu b^{3,1} c^{4,3})\big(z^1?(\bar{z}^2).(y^2!\langle b^{3,1}\rangle \mid \bar{z}^2!\langle c^{4,2}\rangle \mid \\ &\qquad\qquad b^3?(\bar{y}^{5,1}).alice_2!\langle \bar{y}^{5,1}, c^{4,1}\rangle)\big)\end{aligned}$$

The whole protocol is modeled as the term $(\nu ab)(\mathsf{Bob} \mid \mathsf{Alice}_1 \mid \mathsf{Alice}_2 \mid bob!\langle a, b\rangle \mid (\nu c)(alice_1!\langle a, c\rangle \mid alice_2!\langle b, c\rangle))$. Note the interleaving of possibly blocking actions on different channels in both Alice and Bob. ■

IV. Related work

The articles [14]–[16], [19] have been a source of inspiration for our work so we devote most of this section to them. The discussion is quite technical and some familiarity with these works is assumed.

Deadlock freedom. [16] defines a type system for ensuring deadlock freedom in the π-calculus. Up to syntactic details, channels are typed by pairs $[t]/U$ where t is the type of messages and U – called *usage* – is a term of a process algebra that includes prefixing, parallel composition, and replication and that specifies how the channel is accessed. For example, a channel of type $[\mathtt{int}]/(!^m_n.?^h_k \mid ?^n_m.!^k_h)$ can be used concurrently by two processes, one that first sends and then receives an integer and the other that does the opposite. Actions like $!^m_n$ are annotated with an *obligation* m and a *capability* n to prevent mutual dependencies between channels: inputs on a channel with capability n can only block operations on other channels whose obligation is strictly larger than n. The special value ∞ is allowed for denoting various forms of unlimited channels.

There is a significant overlapping between [16] and our work, not only because usages can describe linear channels as a particular case, but also because linearized channels (those that can be accessed multiple times, but only sequentially) can be encoded using linear channels and continuation passing. Also, priorities have been directly inspired by obligations and capabilities and are used similarly, although having obligations and capabilities instead of a single priority sometimes allows the typability of processes that are ill typed in our discipline. For example, the process

$$*c?(x, y).x?(z).y?(z) \mid *c?(x, y).y?(z).x?(z) \qquad (10)$$

can be typed in [16] by giving both x and y obligation 1 and capability 0. The null capability means that, assuming that outputs on x and y are available in the environment immediately (capability 0), then either of these processes guarantees that the both inputs will be performed after at most one reduction (obligation 1). We believe that it is possible to split priorities into obligations and capabilities to recover processes like these. The main distinguishing features of our type systems are given by polymorphism and the relative interpretation of priorities in message types. To illustrate how these features affect the accuracy of our type systems, we revisit Example II.2 in the setting of [16] assuming to extend usages with recursion (recursive usages are admitted in [16], but only for type reconstruction purposes). The following modeling of $L_2(\mathrm{A}, \mathrm{A})$ uses no explicit continuation passing and therefore is the most favorable to the type system in [16]:

$$*c?(x, y).(x!\langle 3\rangle \mid y?(z).c!\langle x, y\rangle) \mid c!\langle e, f\rangle \mid c!\langle f, e\rangle \qquad (11)$$

The process is tentatively typed using the assignments

$$\begin{aligned}e &: [\mathtt{int}]/(\mu\alpha.!^m_n.\alpha \mid \mu\alpha.?^n_m.\alpha)\\ f &: [\mathtt{int}]/(\mu\alpha.?^h_k.\alpha \mid \mu\alpha.!^k_h.\alpha)\end{aligned}$$

where the fact that obligations and capabilities are swapped in complementary actions of the usages of e and f is necessary to satisfy the *reliability* condition [16]. In our setting, reliability corresponds to the property that channel types with opposite polarities and same priority can be combined together (last equation of (8)). The structure of (11) requires the simultaneous satisfiability of $k < m$ (there is an input on f with capability k that blocks e whose topmost obligation is m) and of $m < k$ (there is an input on e with capability m that blocks f whose topmost obligation is k), therefore (11) cannot be proven deadlock free with the type discipline in [16]. The problem is that obligations and capabilities, unlike priorities, are statically assigned to actions and do not change according to the unfolding of recursive usages: e and f always have the same capability and obligation regardless of how many iterations have been performed. By contrast, in Example III.5 the channels e and f and their continuations are assigned a numerically increasing sequence of priorities. There are two features in [16] that partially overcome the limitations due to statically assigned obligations and capabilities. The first one

is that the behavioral nature of usages allows the typing of recursive processes involving one channel. For example, the following variant of $L_2(\mathtt{A},\mathtt{B})$

$$*c?(y).(y!\langle\rangle \mid c!\langle y\rangle) \mid *d?(y).y?(z).d!\langle y\rangle \mid c!\langle e\rangle \mid d!\langle e\rangle$$

is well typed in [16]. The point is that in [16] an input process $u?(x).P$ is well typed if the capability of u is smaller than the obligation of all the channels in P *except* u *itself*. The second feature stems from the observation that the strict order $<$ used for comparing the capability of a channel u and the obligation of another channel v can be safely weakened to $\leq$ if u has been created later than v. This property, denoted by the relation $u \prec v$ in [16], can be inferred by looking at the syntactic nesting of restrictions in a process. This makes it possible to deal with processes like (3), where the input on the newer channel a is allowed to block operations on the older channel y despite the fact that a and y have the same type (hence the same obligations and capabilities). As noted in [16], the very same feature would also work using the opposite order $\succ$, but one would have to choose between either $\prec$ or $\succ$, since using their union would be unsound. The opposide order $\succ$ however arises naturally with continuation passing, as we have seen in (4) and in most of the examples throughout the paper.

In our approach, we stick to the idea that causal dependencies between channels are reflected by a strict ordering of their priorities. We tame the difficulties posed by processes like (3), (4), (11) with polymorphism that in the end grants better accuracy with recursive behaviors.

Outside the domain of linear(ized) channels, usages provide more precise information than unlimited channel types. For example, the lock-free modeling of the dining philosophers in [14] cannot be captured by our type system if only because it needs unlimited channels for representing shared resources (the forks) accessed by competing processes (the philosophers). It is plausible to imagine a composition of the two approaches (usages for unlimited channels and priorities for linear ones) to achieve the best of both.

Lock freedom. Type systems for lock freedom have been presented in [14], [15]. These works essentially rely on the same types and usages already discussed for deadlock freedom and are subject to the same problems illustrated while discussing (11). It is important to observe that the partial order $\prec$ between channels is unsound for lock freedom, as witnessed by the fact that the encoding of the factorial function (3) diverges if applied to a negative number. This is reflected also in our type system for lock freedom, where negative priorities are banned. Lock-free typability of (3) (restricted to natural numbers) and many other recursive functions is recovered in [19], where it is observed that lock freedom can be characterized as the conjunction of deadlock freedom and *termination*. Such characterization turns out to be more accurate on recursive processes with function-like behaviors, partly because the type system for deadlock freedom imposes fewer constraints, partly thanks to the accuracy of the techniques for proving termination. In fact, the hybrid approach [19] is parametric on the particular techniques for verifying deadlock freedom or termination. This means that, when instantiated with the deadlock free analysis of [16], it suffers from the same shortcomings that we have discussed earlier for processes like (11). It also means that it should be applicable in our setting, in which case it would benefit from the better accuracy of our type system for deadlock freedom. By the way, the limit we impose on the number of times a channel can be sent in a message is akin to ensuring a form of termination (in fact, the eventual use of the channel for performing a communication) so in a sense our type system for lock freedom can already be seen as an instance of the hybrid framework described in [19].

Session types. Session types [11], [12] are behavioral types akin to sequential usages in which each single input/output action is annotated with the type of a message. For example, the session type $!\langle\mathtt{int}\rangle.?\langle\mathtt{bool}\rangle$ denotes a channel on which it is possible to first sent an integer, and then receive a boolean. *Duality* relates session types associated with corresponding endpoints of a binary session and generalizes the composition operator between linear types (see (8)). For example, the session type above is dual of $?\langle\mathtt{int}\rangle.!\langle\mathtt{bool}\rangle$. Duality ensures communication safety and also lock freedom, but the latter property is guaranteed only *within* sessions, not when (blocking) actions from different sessions are interleaved with each other. In some works [2], [23] a plain session type discipline ensures deadlock freedom also when sessions are interleaved, but only because the syntax of (well-typed) processes prevents the modeling of cyclic network topologies.

Global types. A global type [3], [9], [10], [13] describes the interactions that are going to occur within a session from a vantage point of view. For example, the interaction between the three processes that compose the system $L_3(\mathtt{A},\mathtt{B},\mathtt{B})$ in Example III.5 can be described by the global type G_3 that satisfies the equation $G_3 = \mathtt{A} \to \mathtt{B}.\mathtt{B} \to \mathtt{C}.\mathtt{C} \to \mathtt{A}.G_3$ which gives names $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$ to the three *participants* and specifies the order of their interactions. In general, global types also allow the specification of the type of the exchanged messages and include composition operators other than prefixing, such as choice and parallel composition. Global types that satisfy some well-formedness conditions are *projected* into session types and processes that conform to these projections are guaranteed to be type safe, to implement the protocol described by the global type, and to be lock free. Therefore, global types are an effective way of extending the lock freedom property from binary to n-ary sessions. This approach relies on the advance planning of the whole network topology, making it more difficult to build and maintain systems *compositionally* (although some modularity and compositionality aspects have recently been addressed in [8] and [20]). For instance, $L_2(\mathtt{A},\mathtt{B})$ and $L_3(\mathtt{A},\mathtt{B},\mathtt{B})$ in Example II.2 are assembled in a modular way using the same constituents, but the global types $G_2 = \mathtt{A} \to \mathtt{B}.\mathtt{B} \to \mathtt{A}.G_2$ and G_3 above that describe the interactions in these systems are quite different. It may also be difficult to describe "unstructured" interactions using global types. For instance, it takes the most sophisticated global type language available to date [9] to describe accurately the dependencies

between Alice's threads in Example III.8. For these reasons, global types are ideal for describing confined interactions within sessions, but do not scale well as a universal mechanism for the enforcement of (dead)lock freedom. These observations motivate mixed approaches [1], [4], [5] that take advantage of the fact that lock freedom is granted *by design* within sessions and focus on the order in which different sessions interleave with the purpose of preventing mutual dependencies between sessions. Unfortunately, these mixed approaches turn out to be fairly restrictive with respect to recursion (*e.g.*, recursive processes can access one session only [5]) and require heavyweight and multi-layer type systems.

V. Conclusion

We have proposed two type systems that strengthen the guarantees regarding the usage of linear channels. The type systems exploit a form of polymorphism that affects their accuracy, in particular with respect to recursive processes in cyclic network topologies. All previous type-based approaches for ensuring deadlock and lock freedom of communicating processes make use of rich behavioral types (usages [14], [15], [17], [19], session types [21], global types [1], [4], [5], [9], [10], [13], conversation types [22]), and when they are able to deal with recursive processes it is because they take advantage of such richness (see Section IV), almost suggesting that complex types are necessary to ensure properties as strong as deadlock and lock freedom. We have shown that this is not necessarily the case: our type systems solely rely on simple pairs of numbers priority/rank and are unperturbed by the encoding of complex conversations using explicit continuations.

Our type systems can be used for reasoning on deadlock and lock freedom of session-based langauges, also in presence of session interleaving. Indeed, the linear π-calculus is the "assembly language" that lies beneath binary sessions [7], [17] and in the long version of the paper we show that a large fraction of multiparty sessions can also be compiled into the linear π-calculus. So, one can establish deadlock and lock freedom of a process by typing its compilation using our type systems. Alternatively, one can lift with little effort the approach described in this paper to type systems based on session and/or global types: just like we annotate *channel types* with priorities and ranks, it is possible to annotate in the same way the single *actions* within session and global types. This experiment was already attempted in [21], [22], but in these works the annotations are "static", much like obligations/capabilities, and so they suffer from the same limitations we have discussed for usages in (11).

Whether and how the approach can be extended to more general languages with non-linear channels [14]–[16] remains to be established. At least, it can improve the accuracy of hybrid approaches [19] applied to the linear π-calculus.

The form of polymorphism we have considered allows the shifting of priorities with maps of the form $z \mapsto z + n$. This prevents, for example, typing an output $a!\langle x^0, y^1\rangle$ if a has type $!^{\omega}[?[\mathtt{int}]^1 \times ![\mathtt{int}]^3]$, because a expects x and y to have priorities at distance 2, while they are at distance 1. In fact, the type system for deadlock freedom can be generalized to rational numbers for priorities and *positive affine maps* for transforming them. In the example above, the map $z \mapsto \frac{1}{2}z - \frac{1}{2}$ turns the priorities 1 and 3 in the type of a into 0 and 1, namely the priorities of x and y. The type system for lock freedom, on the other hand, is somewhat less flexible, because its soundness proof requires priorities to be part of a well-founded set.

Work on type reconstruction for our type systems is well under way. Intuitively, the reconstruction algorithm accumulates all constraints between priorities and ranks imposed by the type system (Table 1). These constraints can always be represented as an integer programming problem, which is solvable using known methods.

References

[1] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR'08*, LNCS 5201, pages 418–433, 2008.

[2] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, LNCS 6269, pages 222–236, 2010.

[3] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On Global Types and Multi-Party Sessions. *Log. Meth. in Comp. Sci.*, 8:1–45, 2012.

[4] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION'13*, LNCS 7890, pages 45–59. Springer, 2013.

[5] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. in Comp. Sci.*, to appear.

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

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

[8] R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR'12*, LNCS 7454, pages 272–286. Springer, 2012.

[9] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12*, LNCS 7211, pages 194–213. Springer, 2012.

[10] P.-M. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Log. Meth. in Comp. Sci.*, 8(4), 2012.

[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] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

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

[15] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.

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

[17] N. Kobayashi. Type systems for concurrent programs. Technical report, Tohoku University, 2007. Short version appeared in 10th Anniversary Colloquium of UNU/IIST, 2002.

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

[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] F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR'13*, LNCS 8052, pages 425–439. Springer, 2013.

[21] L. Padovani. From Lock Freedom to Progress Using Session Types. In *PLACES'13*, EPTCS 137, pages 3–19, 2013.

[22] H. T. Vieira and V. T. Vasconcelos. Typing progress in communication-centred systems. In *COORDINATION'13*, LNCS 7890, pages 236–250. Springer, 2013.

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

APPENDIX A

Although the reduction relation defined in Section II is perfectly adequate to explain the operational semantics of the language and the deadlock and lock freedom properties, it lacks some important information that is instrumental for the proof of subject reduction. In particular, it is necessary to keep track of the *channel* on which a synchronization occurs, so that we can suitably update the type of that channel in case it is linear (recall that, by definition, only one synchronization is supposed to occur on a linear channel). To this aim, we define a *labeled variant* of the reduction relation which keeps track of this information. Labels $act, \ldots$ are either channels or the special symbol τ indicating an unobservable reduction. The labeled reduction relation is inductively defined by the rules below:

$$\text{[LR-COMM]}\quad a!\langle \mathsf{v}\rangle \mid a?(x).P \xrightarrow{a} P\{\mathsf{v}/x\} \qquad \text{[LR-COMM*]}\quad a!\langle \mathsf{v}\rangle \mid *a?(x).P \xrightarrow{a} P\{\mathsf{v}/x\} \mid *a?(x).P$$

$$\text{[LR-LET]}\quad \texttt{let}\ x,y = \mathsf{v},\mathsf{w}\ \texttt{in}\ P \xrightarrow{\tau} P\{\mathsf{v},\mathsf{w}/x,y\} \qquad \text{[LR-CASE]}\quad \texttt{case}\ k\ \mathsf{v}\ \{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}} \xrightarrow{\tau} P_k\{\mathsf{v}/x_k\}$$

$$\text{[LR-PAR 1]}\ \frac{P \xrightarrow{act} P'}{P \mid Q \xrightarrow{act} P' \mid Q} \qquad \text{[LR-PAR 2]}\ \frac{Q \xrightarrow{act} Q'}{P \mid Q \xrightarrow{act} P \mid Q'} \qquad \text{[LR-NEW 1]}\ \frac{P \xrightarrow{act} Q \qquad act \neq a}{(\nu a)P \xrightarrow{act} (\nu a)Q} \qquad \text{[LR-NEW 2]}\ \frac{P \xrightarrow{a} Q}{(\nu a)P \xrightarrow{\tau} (\nu a)Q}$$

$$\text{[LR-STRUCT]}\ \frac{P \equiv P' \qquad P' \xrightarrow{act} Q' \qquad Q' \equiv Q}{P \xrightarrow{act} Q}$$

The two reduction semantics are perfectly equivalent:

Theorem A.1. *The predicates $P \to Q$ and $\exists act : P \xrightarrow{act} Q$ are equivalent.*

Proof: A straightforward induction on the derivation of the reduction. □

APPENDIX B

A. Definitions

Below is the formal definition of contractive types: we say that a type is *contractive* if $\emptyset \vdash \mathsf{con}(t)$ is inductively derivable by the rules

$$A \vdash \mathsf{con}(\texttt{int}) \qquad A \setminus \{\alpha\} \vdash \mathsf{con}(\alpha) \qquad \frac{\emptyset \vdash \mathsf{con}(t)}{A \vdash \mathsf{con}(p^\iota[t])} \qquad \frac{A \vdash \mathsf{con}(t)}{A \vdash \mathsf{con}(\mathtt{M}\,t)}$$

$$\frac{A \vdash \mathsf{con}(t_i)\ ^{(i=1,2)}}{A \vdash \mathsf{con}(t_1 \times t_2)} \qquad \frac{A \vdash \mathsf{con}(t_i)\ ^{(i=1,2)}}{A \vdash \mathsf{con}(t_1 \oplus t_2)} \qquad \frac{A \cup \{\alpha\} \vdash \mathsf{con}(t)}{A \vdash \mathsf{con}(\mu\alpha.t)}$$

where A is a set of type variables. Note that $\mu\alpha.p^\iota[\alpha]$ is contractive but neither $\mu\alpha.(\alpha \times \alpha)$ nor $\mu\alpha.(\alpha \oplus \alpha)$ are. The last one, in particular, seems preventing the definition of any recursive algebraic type. The definition of contractiveness we adopt is convenient for the rest of the technical development but it can be relaxed to allow α be guarded by *any* constructor, at the cost of a slightly more complicated definition of the function $|\cdot|$ (see (6)). In this article we do not investigate the issue further and we always assume to work with contractive types.

B. Basic properties

Below are two auxiliary results regarding the type systems, namely the shifting property and the substitution lemma for expressions and processes. Both results can be easily proved with an induction on the derivation of the judgment in the premise.

Lemma B.1 (Lemma III.2)**.** *The following properties hold:*

1) *If $\Gamma \vdash e : t$, then $\$^n\,\Gamma \vdash e : \$^n\,t$.*
2) *If $\Gamma \vdash_0 P$, then $\$^n\,\Gamma \vdash_0 P$.*
3) *If $\Gamma \vdash_1 P$ and $n \geq 0$, then $\$^n\,\Gamma \vdash_1 P$.*

Lemma B.2 (substitution)**.** *Let $\Gamma' \vdash \mathsf{v} : t$ and $\Gamma + \Gamma'$ be defined. Then:*

1) *$\Gamma, x : t \vdash e : s$ implies $\Gamma + \Gamma' \vdash e\{\mathsf{v}/x\} : s$.*
2) *$\Gamma, x : t \vdash_k P$ implies $\Gamma + \Gamma' \vdash_k P\{\mathsf{v}/x\}$.*

We will make extensive use of Lemma III.2 in the proofs that follow, to the point that it is convenient to extend the type system presented in Table 1 with the (admissible) rule

$$\frac{\Gamma \vdash_k P \qquad k = 1 \Rightarrow n \geq 0}{\$^n\,\Gamma \vdash_k P}\ [\text{T-SHIFT}]$$

C. Subject reduction

In order to formulate and prove the subject reduction result, we need to determine a precise correspondence between the type environment *before* a reduction and the one *after* the reduction. We do so by defining a reduction relation for type environments using the same labels as those in the labeled reduction for processes. Let $\xrightarrow{act}$ be the least relation defined by the rules

$$\Gamma, a : \#[t]^{n,m} \xrightarrow{a} \Gamma, a : \emptyset[t]^{n,m} \qquad \Gamma, a : \#^{\omega}[t] \xrightarrow{a} \Gamma, a : \#^{\omega}[t] \qquad \Gamma, a : \$^{0,1}\,t \xrightarrow{\tau} \Gamma, a : t$$

The first rule consumes a linear channel a from the environment and is used when a synchronization on a occurs (recall that linear channels can be used for one synchronization only by definition). The middle rule leaves the environment unchanged when a synchronization on some unlimited channel a occurs. The last rule decreases the rank in the type of a and is used each time a is sent in a message. In the following we write $\stackrel{\tau}{\Longrightarrow}$ for the reflexive, transitive closure of $\xrightarrow{\tau}$ and $\stackrel{act}{\Longrightarrow}$ for the composition $\stackrel{\tau}{\Longrightarrow}\xrightarrow{act}\stackrel{\tau}{\Longrightarrow}$.

Lemma B.3. *Let $\Gamma \vdash \mathsf{v} : \$^{0,1}\,t$. Then there exists Γ' such that $\Gamma \stackrel{\tau}{\Longrightarrow} \Gamma'$ and $\Gamma' \vdash \mathsf{v} : t$.*

Proof: By induction on the derivation of $\Gamma \vdash \mathsf{v} : \$^{0,1}\,t$. We only show two cases, correponding to the channel and pair expressions:

- ($\mathsf{v} = a$) Then $\Gamma = \Gamma'', a : p^{\iota}[s]^{n,m}$ and $\mathsf{un}(\Gamma'')$. We distinguish two subcases:
  - ($\iota = 1$) Then $\$^{0,1}\,t = p[s]^{n,m}$ and $m > 0$ and $t = p[s]^{n,m-1}$ and we conclude by taking $\Gamma' = \Gamma'', a : t$ and observing that $\Gamma \xrightarrow{\tau} \Gamma''$.
  - ($\iota = \omega$) Then $\$^{0,1}\,t = t = p^{\omega}[s]$ and we conclude by taking $\Gamma' = \Gamma$.
- ($\mathsf{v} = (\mathsf{v}_1, \mathsf{v}_2)$) Then $\Gamma = \Gamma_1 + \Gamma_2$ and $\$^{0,1}\,t = (\$^{0,1}\,t_1) \times (\$^{0,1}\,t_2)$ and $\Gamma_i \vdash \mathsf{v}_i : \$^{0,1}\,t_i$ for $i = 1, 2$. By induction hypothesis there exist Γ_1' and Γ_2' such that $\Gamma_i \stackrel{\tau}{\Longrightarrow} \Gamma_i'$ and $\Gamma_i' \vdash \mathsf{v}_i : t_i$ for $i = 1, 2$. Observe that the $\xrightarrow{\tau}$ relation between type environments can only decrease the object ranks of values. This means that $\Gamma_1' + \Gamma_2'$ is defined and moreover $\Gamma \stackrel{\tau}{\Longrightarrow} \Gamma_1' + \Gamma_2'$. We conclude by taking $\Gamma' = \Gamma_1' + \Gamma_2'$.

□

The following result states that the type of an expression is the same as that of its value. Clearly, if the language of expressions is extended with more constructors and/or operators, it is expected that they obey subject reduction.

Lemma B.4. *Let $\Gamma \vdash e : t$ and $e \downarrow \mathsf{v}$. Then $\Gamma \vdash \mathsf{v} : t$.*

Proof: Trivial induction on the structure of e, using the fact that $e \downarrow \mathsf{v}$ implies that e is closed. □

The following result states that well-typing of processes is preserved by reductions. Note though that the type environment may change.

Lemma B.5 (Lemma III.3)**.** *If $\Gamma \vdash_k P$ and $P \xrightarrow{act} P'$, then $\Gamma' \vdash_k P'$ for some Γ' such that $\Gamma \stackrel{act}{\Longrightarrow} \Gamma'$.*

Proof: By induction on the derivation of $P \xrightarrow{act} P'$ and by cases on the last rule applied. We only show the most relevant cases for $k = 1$.

[R-COMM] Then $P = a!\langle e\rangle \mid a?(x).Q \xrightarrow{a} Q\{\mathsf{v}/x\} = P'$ where $e \downarrow \mathsf{v}$ and $act = a$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash_k a!\langle e\rangle$ and $\Gamma_2 \vdash_k a?(x).Q$. From [T-OUT] we deduce $\Gamma_1 = \Gamma_1' + a : \,!\,[t]^{n,k}$ and $\Gamma_1' \vdash e : \$^{n,k}\,t$. From [T-IN] we deduce $\Gamma_2 = \Gamma_2' + a : \,?[t]^{n,h}$ and $\Gamma_2', x : \$^n\,t \vdash_k Q$. The fact that the types associated with the two a's are complementary follows from the definition of $\Gamma_1 + \Gamma_2$. By Lemma B.4 we obtain $\Gamma_1' \vdash \mathsf{v} : \$^{n,k}\,t$. By Lemma B.3 we deduce that there exists Γ_1'' such that $\Gamma_1'' \vdash \mathsf{v} : \$^n\,t$ and $\Gamma_1' \stackrel{\tau}{\Longrightarrow} \Gamma_1''$. Note that $\Gamma_1'' + \Gamma_2'$ is defined. Then by Lemma B.2(2) we obtain $\Gamma_1'' + \Gamma_2' \vdash_k Q\{\mathsf{v}/x\}$. Let $\Gamma' = \Gamma_1'' + \Gamma_2' + a : \emptyset[t]^{n,k}$. From [T-PAR] we deduce $\Gamma' \vdash_k P'$. We conclude by observing that $\Gamma \stackrel{act}{\Longrightarrow} \Gamma'$.

[R-COMM*] Then $P = a!\langle e\rangle \mid *a?(x).Q \xrightarrow{a} Q\{\mathsf{v}/x\} \mid *a?(x).Q = P'$ where $e \downarrow \mathsf{v}$ and $act = a$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash_k a!\langle e\rangle$ and $\Gamma_2 \vdash_k *a?(x).Q$. From [T-OUT*] we deduce $\Gamma_1 = \Gamma_1' + a : \,!^{\omega}[t]$ and $\Gamma_1' \vdash e : \$^{n,k}\,t$. From [T-IN*] we deduce $\Gamma_2 = \Gamma_2' + a : \,?^{\omega}[t]$ and $\Gamma_2', x : t \vdash_k Q$ and $\mathsf{un}(\Gamma_2')$. By Lemma B.4 we obtain $\Gamma_1' \vdash \mathsf{v} : \$^{n,k}\,t$. By Lemma B.3 we deduce that there exists Γ_1'' such that $\Gamma_1'' \vdash \mathsf{v} : \$^n\,t$ and $\Gamma_1' \stackrel{\tau}{\Longrightarrow} \Gamma_1''$. By Lemma III.2 we deduce $\$^n\,\Gamma_2', x : \$^n\,t \vdash_k Q$. From $\mathsf{un}(\Gamma_2')$ we deduce $\Gamma_2' = \$^n\,\Gamma_2'$, therefore $\Gamma_1' + \$^n\,\Gamma_2'$ is defined. By Lemma B.2(2) we deduce $\Gamma_1'' + \Gamma_2' \vdash_k Q\{\mathsf{v}/x\}$. From [T-IN*] and [T-PAR] we obtain $\Gamma_1'' + \Gamma_2 \vdash_k Q\{\mathsf{v}/x\} \mid *a?(x).Q$. We conclude by taking $\Gamma' = \Gamma_1'' + \Gamma_2$ and observing that $\Gamma \stackrel{act}{\Longrightarrow} \Gamma'$.

[R-LET] Then $P = \texttt{let}\ x, y = e\ \texttt{in}\ Q$ and $e \downarrow (\mathsf{v}, \mathsf{w})$ and $P' = Q\{\mathsf{v}, \mathsf{w}/x, y\}$. From [T-LET] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash e : t \times s$ and $\Gamma_2, x : t, y : s \vdash_k Q$. From Lemma B.4 we deduce $\Gamma_1 \vdash \mathsf{v}, \mathsf{w} : t \times s$. From [T-PAIR] we deduce $\Gamma_1 = \Gamma_{11} + \Gamma_{12}$ where $\Gamma_{11} \vdash \mathsf{v} : t$ and $\Gamma_{12} \vdash \mathsf{w} : s$. By Lemma B.2 we deduce $\Gamma \vdash_k Q\{\mathsf{v}, \mathsf{w}/x, y\}$ and we conclude by taking $\Gamma' = \Gamma$.

[R-CASE] Then $P = \texttt{case}\ e\ \{\texttt{inl}\ x_1 \Rightarrow P_1, \texttt{inr}\ x_2 \Rightarrow P_2\}$ and either $e \downarrow \texttt{inl}\ \mathsf{v}_1$ or $e \downarrow \texttt{inr}\ \mathsf{v}_2$ and $P \to P_i\{\mathsf{v}_i/x_i\} = P'$. From [T-CASE] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash e : t_1 \oplus t_2$ and $\Gamma_2, x_i : t_i \vdash_k P_i$. From Lemma B.4 and either [T-INL] or [T-INR] we deduce $\Gamma_1 \vdash \mathsf{v}_i : t_i$. From Lemma B.2 we deduce $\Gamma \vdash_k P_i\{\mathsf{v}_i/x_i\}$ and we conclude by taking $\Gamma' = \Gamma$.

[R-NEW 1] Then $P = (\nu a)Q$ and $Q \xrightarrow{act} Q'$ and $P' = (\nu a)Q'$ and $act \neq a$. From the hypothesis $\Gamma \vdash_k P$ we deduce $\Gamma, a : p^{\iota}[t]^{n,m} \vdash_k Q$ where p is even. By induction hypothesis we derive that there exists Γ' and $h \leq m$ such that $\Gamma', a : p^{\iota}[t]^{n,h} \vdash_k Q'$ and $\Gamma \xRightarrow{act} \Gamma'$. We conclude with an application of [T-NEW]. □

D. Soundness

Definition B.6 (balanced environment)**.** We say that Γ is *balanced* if $\Gamma = \{a_i : p_i^{\iota_i}[t_i]^{n_i,m_i}\}_{i \in I}$ and p_i is even for every $i \in I$.

Lemma B.7 (deadlock freedom)**.** *Let* $\vdash_{\mathtt{DF}} P$ *and* $P \not\to$*. Then* $P \equiv (\nu \tilde{a})\left(\prod_{i \in I} *a_i?(x_i).P_i\right)$.

Proof: From the hypothesis that P is well typed and does not reduce we know that $P \equiv (\nu \tilde{a})Q$ where

$$Q \equiv \prod_{i \in I} *a_i?(x_i).P_i \mid \prod_{j \in J} b_j?(y_j).Q_j \mid \prod_{k \in K} c_k!\langle e_k \rangle$$

hence, from [T-NEW], we deduce $\Gamma \vdash_0 Q$ for some Γ that is balanced. We proceed showing that $J = K = \emptyset$ by contradiction. In particular, we show that if $J \cup K \neq \emptyset$, then it is possible to find an infinite chain of channels with strictly decreasing priority. This is absurd since Q is finite, so there is a finite number of distinct channels in it.

We begin doing a few generic considerations about the channels occurring in these processes:

- all the a_i's are unlimited channels, because they are used for replicated inputs;
- all the b_j are linear channels, because they are used for linear inputs;
- all the c_k's are linear channels, because $P \not\to$ whereas if they were unlimited they would be able to synchronize with one of the replicated inputs (recall that the type system guarantees input receptiveness);
- from [T-IN*] we know that none of the P_i's has free linear channels.

Now suppose $j \in J$ and $\Gamma(b_j) = ?[t]^{n,m}$. Since b_j is a linear channel, there must be another occurrence of b_j with output polarity. We observe that:

- b_j is not any of the b_l for $l \in J \setminus \{j\}$, for the occurrence of b_j we are looking for must have output polarity;
- b_j is not any of the c_k's, because $P \not\to$ and all the e_k must be closed and therefore evaluated.

We analyze the remaining possibilities:

- ($b_j \in \mathsf{fn}(Q_l)$ for some $l \in J$) Then from [T-IN] we deduce $\Gamma(b_l) = ?[s]^{n',m'}$ where $n' < n$.
- ($b_j \in \mathsf{fn}(e_k)$ for some $k \in K$) Then from [T-OUT] we deduce $\Gamma(c_k) = ![s]^{n',m'}$ where $n' < n$.

Now suppose $k \in K$ and $\Gamma(c_k) = ![t]^{n,m}$. Since c_k is a linear channel, there must be another occurrence of c_k with input polarity. We observe that:

- c_k is not any of the b_j's, because $P \not\to$ and e_k must be closed and therefore evaluated.
- c_k is not any of the c_l for $l \in K \setminus \{j\}$, for the occurrence of c_k we are looking for must have input polarity.

We analyze the remaining possibilities:

- ($c_k \in \mathsf{fn}(Q_j)$ for some $j \in J$) Then from [T-IN] we deduce $\Gamma(b_j) = ?[s]^{n',m'}$ where $n' < n$.
- ($c_k \in \mathsf{fn}(e_l)$ for some $l \in K$) Then from [T-OUT] we deduce $\Gamma(c_l) = ![s]^{n',m'}$ where $n' < n$.

In conclusion, for every possibility that is not ruled out by simple considerations, from any channel in $\{b_j \mid j \in J\} \cup \{c_k \mid k \in K\}$ we have been able to find another channel in the same set with a strictly smaller priority. This contradicts the fact that the very same set must be finite. □

In order to prove that the type system for lock freedom is sound we need to define the *measure* of a channel. Intuitively, the measure of a represents an upper bound to the (finite) number of reduction steps that must occur before a synchronization on a occurs. The measure of a channel depends on both its type and the positions in which it occurs in a process. We only consider processes that are well typed in a balanced environment.

We let $\mathsf{rank}(t)$ denote the rank of a type, which is the analogous generalization of priorities for types in (6) except that we consider m in the second equation.

Definition B.8. Let $\Gamma \vdash_1 P$ where Γ is balanced and $a : t \in \Gamma$. Then $\mathsf{measure}(a, \Gamma, P) \stackrel{\text{def}}{=} (|t|, \mathsf{rank}(t), [\![P]\!]_a)$ where $[\![P]\!]_a$ is inductively defined in Table 3.

TABLE 3: Depth of a channel.

$$
\begin{array}{rcll}
[\![a]\!]_a & = & 0 & \\
[\![u]\!]_a & = & \bot & a \neq u \\
[\![e_1, e_2]\!]_a & = & [\![e_1]\!]_a \vee [\![e_2]\!]_a & \\
[\![\texttt{inl}\ e]\!]_a = [\![\texttt{inr}\ e]\!]_a & = & [\![e]\!]_a & \\
 & & & \\
[\![\mathbf{0}]\!]_a & = & \bot & \\
[\![u?(x).P]\!]_a & = & [\![u]\!]_a \vee ([\![P]\!]_a + 1) & \\
[\![*P]\!]_a & = & [\![P]\!]_a & \\
[\![u!\langle e\rangle]\!]_a & = & [\![u]\!]_a \vee [\![e]\!]_a & \\
[\![P \mid Q]\!]_a & = & [\![P]\!]_a \vee [\![Q]\!]_a & \\
[\![\texttt{let}\ x, y = e\ \texttt{in}\ P]\!]_a & = & [\![e]\!]_a \vee [\![P]\!]_a & \\
[\![\texttt{case}\ e\ \{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}}]\!]_a & = & [\![e]\!]_a \vee [\![P_{\texttt{inl}}]\!]_a \vee [\![P_{\texttt{inr}}]\!]_a & \\
[\![(\nu a)P]\!]_a & = & \bot & \\
[\![(\nu b)P]\!]_a & = & [\![P]\!]_a & a \neq b
\end{array}
$$

Given $\Gamma \vdash_1 P$ where Γ is balanced and $a \in \mathsf{dom}(\Gamma)$, the measure of a is a triple consisting of the priority $|\Gamma(a)|$ of a, the rank $\mathsf{rank}(\Gamma(a))$ of a, and the maximum "depth" of a in P, namely the maximum number of input prefixes that guard an occurrence of a in P. Observe that, when P is well typed, the depth $[\![P]\!]_a$ is always finite. We write $<$ for the usual lexicographic order on triples returned by $\mathsf{measure}(a, \Gamma, P)$. Note that $<$ is well founded, *i.e.* there are no infinite descending chains of triples related by $<$, because the type system for lock freedom restricts priorities to natural numbers.

Lemma B.9. *Let* **(1)** $\Gamma \vdash_1 P$ *and* **(2)** Γ *balanced and* **(3)** *either* $\mathsf{in}(a, P)$ *or* $\mathsf{out}(a, P)$. *Then there exists* Q *such that* $P \to^* Q$ *and* $\mathsf{sync}(a, Q)$.

Proof. Without loss of generality we can assume that P has no restrictions at the top level. Indeed, if $P \equiv (\nu b)P'$, from **(1)** and [T-NEW] we deduce $\Gamma, b : \#[t]^{n,m} \vdash_1 P'$ where $\Gamma, b : \#[t]^{n,m}$ is balanced, so we could reason on P'. We can also assume that P has no `let` or `case` at the top level. If this is the case, then from **(1)** and the hypothesis that the evaluation of expressions always terminates we can always reduce P to a well-typed process that contains no top level `let`'s or `case`'s.

We proceed by induction, showing that if the result holds for every triple b, Γ', P' such that $\mathsf{measure}(b, \Gamma', P') < \mathsf{measure}(a, \Gamma, P)$, then it holds also for the triple a, Γ, P. We analyze the various possibilities and we reason by cases on the place where a can occur. If $\mathsf{sync}(a, P)$, then we conclude immediately by taking $Q = P$. Next, we analyze a few cases that are impossible given the typing rules of processes:

- $P \equiv \mathscr{C}[a?(x).R]$ and $a \in \mathsf{fn}(R)$. Then $a : \#[t]^{n,m} \in \Gamma$. According to [T-IN] we have $\Gamma', x : \$^n\, t \vdash_1 R$ and $n < |\Gamma'|$. Then $a \notin \mathsf{dom}(\Gamma')$, hence this case is impossible.
- $P \equiv \mathscr{C}[a!\langle e\rangle]$ and $a \in \mathsf{fn}(e)$ and $a : \#^\omega[t] \in \Gamma$ and $\neg{*}\mathsf{in}(a, P)$. Then from [T-OUT*] we deduce $\Gamma' \vdash e : \$^{n,1}\, t$ and $0 \leq |t|$. From $\neg{*}\mathsf{in}(a, P)$ we know that e must contain the endpoint a with input capability, for otherwise it would occur at the top level of P. Hence $|t| = \bot$, which contradicts $0 \leq |t|$. Hence this case is impossible.
- $P \equiv \mathscr{C}[a!\langle e\rangle]$ and $a \in \mathsf{fn}(e)$ and $a : \#[t]^{n,m} \in \Gamma$. Then from [T-OUT] we deduce $\Gamma' \vdash e : \$^{n,1}\, t$ and $0 < |t|$. We deduce

$$
\begin{array}{rcll}
n & < & n + |t| & \text{from } 0 < |t| \\
 & = & |\$^{n,1}\, t| & \text{by definition of } |\cdot| \\
 & \leq & n & \text{from } a \in \mathsf{fn}(e)
\end{array}
$$

  which is absurd, so this case is also impossible.

Finally, we consider the cases in which there is a pending operation on a but no immediate synchronization is possible. In every case we are able to extend the reduction of P to a state where the measure of a is strictly smaller than $\mathsf{measure}(a, \Gamma, P)$. This is enough for establishing the result, given that the order $<$ on measures is well founded. Note also that in each of these cases we are using the assumption that the evaluation of expressions always terminates.

- $P \equiv \mathscr{C}[b!\langle e\rangle \mid *b?(x).R]$ and $a \in \mathsf{fn}(e)$. Then $P \to \mathscr{C}[R\{\mathsf{v}/x\} \mid *b?(x).R] = P'$ and $e \downarrow \mathsf{v}$ and $a \in \mathsf{fn}(\mathsf{v})$. By Lemma III.3 we deduce that $\Gamma' \vdash_1 P'$ for some Γ' such that $\mathsf{rank}(\Gamma'(a)) < \mathsf{rank}(\Gamma(a))$, therefore $\mathsf{measure}(a, \Gamma', P') < \mathsf{measure}(a, \Gamma, P)$. We conclude by induction hypothesis.
- $P \equiv \mathscr{C}[b!\langle e\rangle]$ where $a \in \mathsf{fn}(e)$ and b is a linear channel. From [T-OUT] we deduce $|\Gamma(b)| < |\Gamma(a)|$. By induction hypothesis we derive $P \to^* \mathscr{C}'[b!\langle e\rangle \mid b?(x).R] \to \mathscr{C}'[R\{\mathsf{v}/x\}] = P'$ where $e \downarrow \mathsf{v}$ and $a \in \mathsf{fn}(\mathsf{v})$. By Lemma III.3 there exists Γ' such that $\Gamma' \vdash_1 P'$ and $\mathsf{rank}(\Gamma'(a)) < \mathsf{rank}(\Gamma(a))$. We conclude by induction hypothesis.
- $P \equiv \mathscr{C}[b?(x).R]$ where $a \in \mathsf{fn}(R) \setminus \{b\}$. From [T-IN] we deduce $|\Gamma(b)| < |\Gamma(a)|$. By induction hypothesis we derive $P \overset{\tau}{\Longrightarrow} \mathscr{C}'[b!\langle e\rangle \mid b?(x).R] \overset{b}{\longrightarrow} \mathscr{C}'[R\{\mathsf{v}/x\}] = P'$ for some v such that $e \downarrow \mathsf{v}$. By Lemma III.3 we deduce $\Gamma' \vdash_1 P'$ for some Γ' such that $\Gamma \overset{b}{\Longrightarrow} \Gamma'$ and $\mathsf{measure}(a, \Gamma', P') < \mathsf{measure}(a, \Gamma, P)$ because a is guarded by one less input prefix. We conclude by induction hypothesis. □

$$\Delta, x : \tau \vdash x : \tau \ \&\ 0, 0$$

$$\frac{\Delta, x : \tau \vdash M : \sigma \ \&\ m, n}{\Delta \vdash \lambda x.M : \tau \xrightarrow{m,n} \sigma \ \&\ 0, 0}$$

$$\frac{\Delta \vdash M : \tau \xrightarrow{m,n} \sigma \ \&\ h, k \qquad \Delta \vdash N : \tau \ \&\ h', k'}{\Delta \vdash MN : \sigma \ \&\ \max\{h + 2, h' + 1, m\}, n + 1}$$

□

APPENDIX C
EXTENDED EXAMPLE: LOCK-FREE λ-CALCULUS

In this section we identify a class of terms of the simply-typed λ-calculus whose encoding into the linear π-calculus using the parallel call-by-value evaluation strategy produces well-typed processes according to our type discipline for lock freedom. The encoding provides evidence of the expressiveness of the type system for lock freedom in several respects: first of all, it shows that the dicotomy between unlimited and linear channels retains a significant expressive power; second, in this encoding the lock freedom property actually corresponds to termination; third, the image of the encoding is an example of dynamic system, where the number of active parallel processes grows and shrinks as the system reduces. It should be noted that, differently from the encoding of λ-terms presented in [16], we do not consider recursion and not all well-typed terms of the simply-typed λ-calculus translate into well-typed processes. However, the terms resulting from the encoding in [16] are shown to be well typed with respect to a type system for processes that guarantees *deadlock freedom*, which is strictly weaker than the lock freedom property we refer to in our encoding.

Terms M, N, ... of the λ-calculus and their encoding are defined below:

$$\begin{array}{rcll} M & ::= & x & [\![x]\!]^u \stackrel{\text{def}}{=} u!\langle x\rangle \\ & | & \lambda x.M & [\![\lambda x.M]\!]^u \stackrel{\text{def}}{=} (\nu f)(*f?(x,y).[\![M]\!]^y \mid u!\langle f\rangle) \\ & | & MN & [\![MN]\!]^u \stackrel{\text{def}}{=} (\nu a)(\nu b)([\![M]\!]^a \mid [\![N]\!]^b \mid a?(x).b?(y).x!\langle y,u\rangle) \end{array}$$

A term M is encoded as a process $[\![M]\!]^u$, where u is the *continuation channel* on which the value of M is sent. A variable x is encoded as the output of x on u; an abstraction $\lambda x.M$ is encoded as a replicated input on a channel f that accepts an argument x and another channel y and evaluates M using y as continuation; an application MN evaluates M and N in parallel using two fresh continuations a and b, collects their respective values, and invokes the function denoted by M using the value of N as argument.

Observe how the encoding uses channels in two fundamentally different ways: values are persistent resources that can be used without restrictions and as such they are represented by unlimited channels; the evaluation of a term produces exactly one value which, for this reason, is communicated over a linear channel. For example, a function of type $\texttt{int} \to \texttt{int}$ becomes a channel of type $!^\omega[\texttt{int} \times ![\texttt{int}]]$, where $![\texttt{int}]$ is the type of the continuation on which the function outputs the result of an application. However, we see from the definition of $[\![MN]\!]^u$ that continuation channels are also used for input operations and sent as messages. Therefore, in order to produce well-typed processes as the result of the encoding of well-typed λ-terms, we must determine appropriate subject and object ranks of continuation channels that satisfy the conditions of rules [T-IN] and [T-OUT]. We do so with the help of a simple *effect system* for the λ-calculus. More specifically, types τ, σ, ... are defined by the grammar

$$\tau \ ::= \ \texttt{int} \ \big| \ \tau \xrightarrow{m,n} \sigma$$

and the typing rules are on the right. The judgments of the effect system have the form $\Delta \vdash M : \tau \ \&\ m, n$, where the type environment Δ associates variables with types, τ is the type of M, and m and n are subject and object ranks of the channel on which the value of M is sent. The effect system does not alter in any way the class of well-typed terms: for every type derivation in the classical simply-typed λ-calculus there is an isomorphic one in our effect system, and vice versa. Looking at the encoding function $[\![\cdot]\!]^{\cdot}$ it is clear how the numbers m and n in a judgment $\Delta \vdash M : \tau \ \&\ m, n$ are determined: they are both 0 for variables and abstractions because in these cases the term is already evaluated and the continuation is used immediately. For abstractions, however, we record the ranks associated with the body of the function on the arrow, because they will be necessary when the function is applied and its body evaluated. For applications $[\![MN]\!]^u$, the subject rank m of u must be greater than the subject ranks of both a and b because u is blocked by input actions on these two channels, and the object rank n of u is 1 plus the number of times u is delegated during the evaluation of M, because u is sent along with y in $x!\langle y, u\rangle$.

Types are encoded taking into account the effects recorded over arrows:

$$[\![\texttt{int}]\!] \stackrel{\text{def}}{=} \texttt{int} \qquad [\![\tau \xrightarrow{m,n} \sigma]\!] \stackrel{\text{def}}{=} !^{\omega}[[\![\tau]\!] \times !^{m,n}[[\![\sigma]\!]]$$

The encoding of terms preserves typing:

Theorem C.1. *$\emptyset \vdash M : \tau \;\&\; m, n$ implies $a : ![[\![\tau]\!]]^{m,n} \vdash_1 [\![M]\!]^a$.*

Proof. We prove a sligthly more general result, namely that if $\Delta \vdash M : \tau \;\&\; m, n$ and $u \notin \mathsf{dom}(\Delta)$, then $[\![\Delta]\!], u : ![[\![\tau]\!]]^{m,n} \vdash_1 [\![M]\!]^u$. We proceed by induction on M and by cases on its shape.

$\boxed{M = x}$ Then $\Delta = \Delta', x : \tau$ and $m = n = 0$. We conclude

$$\dfrac{\dfrac{}{[\![\Delta']\!], x : [\![\tau]\!] \vdash x : [\![\tau]\!]}\,\text{[T-NAME]}}{[\![\Delta']\!], x : [\![\tau]\!], a : ![[\![\tau]\!]] \vdash_1 a!\langle x\rangle}\,\text{[T-OUT]}$$

noting that $\mathsf{un}([\![\Delta']\!])$ holds because $[\![\cdot]\!]$ only produces unlimited types.

$\boxed{M = \lambda x.N}$ Then $\tau = \tau_1 \xrightarrow{h,k} \tau_2$ and $\Delta, x : \tau_1 \vdash N : \tau_2 \;\&\; h, k$ and $m = n = 0$. Note that $[\![\tau]\!] = !^{\omega}[[\![\tau_1]\!] \times !^{h,k}[[\![\tau_2]\!]]$. We conclude with the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{[\![\Delta]\!], x : [\![\tau_1]\!], y : ![[\![\tau_2]\!]]^{h,k} \vdash_1 [\![N]\!]^y}\,\text{IND.HYP.}}{[\![\Delta]\!], f : ?^{\omega}[[\![\tau_1]\!] \times !^{h,k}[[\![\tau_2]\!]]] \vdash_1 *f?(x,y).[\![N]\!]^y}\,\text{[T-IN*]} \quad \dfrac{\dfrac{}{f : [\![\tau]\!] \vdash f : [\![\tau]\!]}\,\text{[T-NAME]}}{f : [\![\tau]\!], u : ![[\![\tau]\!]] \vdash_1 u!\langle f\rangle}\,\text{[T-OUT]}}{[\![\Delta]\!], f : \#^{\omega}[[\![\tau_1]\!] \times !^{h,k}[[\![\tau_2]\!]]^{h,k}], u : ![[\![\tau]\!]] \vdash_1 *f?(x,y).[\![N]\!]^y \mid u!\langle f\rangle}\,\text{[T-PAR]}}{[\![\Delta]\!], u : ![[\![\tau]\!]] \vdash_1 (\nu f)(*f?(x,y).[\![N]\!]^y \mid u!\langle f\rangle)}\,\text{[T-NEW*]}$$

where [T-IN*] uses the fact that $\mathsf{un}([\![\Delta]\!])$ holds and [T-OUT] uses $0 < \top = |([\![\tau]\!])$.

$\boxed{M = M_1 M_2}$ Then $\Delta \vdash M_1 : \tau_1 \xrightarrow{m',n'} \tau \;\&\; h, k$ and $\Delta \vdash M_2 : \tau_1 \;\&\; h', k'$ and $m = \max\{h+2, h'+1, m'\}$ and $n = n'+1$. Let a and b two fresh names and let $\sigma = \tau_1 \xrightarrow{m',n'} \tau$. We derive $\mathscr{A}$

$$\dfrac{\dfrac{\vdots}{[\![\Delta]\!], a : ![[\![\sigma]\!]]^{h,k} \vdash_1 [\![M_1]\!]^a}\,\text{IND.HYP.}}{}$$

as well as $\mathscr{B}$

$$\dfrac{\dfrac{\dfrac{\vdots}{[\![\Delta]\!], b : ![[\![\tau_1]\!]]^{h',k'} \vdash_1 [\![M_2]\!]^b}\,\text{IND.HYP.}}{[\![\Delta]\!], b : ![[\![\tau_1]\!]]^{h'',k''} \vdash_1 [\![M_2]\!]^b}\,\text{[T-LIFT]}}{}$$

where $h'' = \max\{h+1, h'\}$. Note that $h < h'' = \max\{h+1, h'\} < \max\{h+2, h'+1\} \leq m$. Using these relations we can now build the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{u : ![[\![\tau]\!]]^{m,n}, y : [\![\tau_1]\!] \vdash y, u : ([\![\tau_1]\!] \times ![[\![\tau]\!]]^{m',n'})}\,\text{[T-PAIR]}}{u : ![[\![\tau]\!]]^{m,n}, x : !^{\omega}[[\![\tau_1]\!] \times ![[\![\tau]\!]]^{m',n'}], y : [\![\tau_1]\!] \vdash_1 x!\langle y, u\rangle}\,\text{[T-OUT*]}}{u : ![[\![\tau]\!]]^{m,n}, b : ?[[\![\tau_1]\!]]^{h'',k'}, x : [\![\sigma]\!] \vdash_1 b?(y).x!\langle y, u\rangle}\,\text{[T-IN]}}{u : ![[\![\tau]\!]]^{m,n}, a : ?[[\![\sigma]\!]]^{h,k}, b : ?[[\![\tau_1]\!]]^{h'',k'} \vdash_1 a?(x).b?(y).x!\langle y, u\rangle}\,\text{[T-IN]}}{[\![\Delta]\!], u : ![[\![\tau]\!]]^{m,n}, a : \#[[\![\sigma]\!]]^{h,k}, b : \#[[\![\tau_1]\!]]^{h'',k'} \vdash_1 [\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y, u\rangle}\,\text{[T-PAR]}}{[\![\Delta]\!], u : ![[\![\tau]\!]]^{m,n}, a : \#[[\![\sigma]\!]]^{h,k} \vdash_1 (\nu b)([\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y, u\rangle)}\,\text{[T-NEW]}}{[\![\Delta]\!], u : ![[\![\tau]\!]]^{m,n} \vdash_1 (\nu a)(\nu b)([\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y, u\rangle)}\,\text{[T-NEW]}$$

where in the application of [T-OUT*] we use the fact that $m' \leq m$ by definition of m. □

To see why Theorem C.1 implies termination of any closed well-typed term M, suppose $\emptyset \vdash M : \tau \;\&\; m, n$. Then we can derive

$$\dfrac{\dfrac{\dfrac{\vdots}{a : \mathsf{!}[[\![\tau]\!]]^{m,n} \vdash_1 [\![M]\!]^a} \qquad \dfrac{}{x : [\![\tau]\!] \vdash_1 \mathbf{0}}}{a : \mathsf{?}[[\![\tau]\!]]^{m,n} \vdash_1 a?(x)}}{a : \#[[\![\tau]\!]]^{m,n} \vdash_1 [\![M]\!]^a \mid a?(x)}}{\emptyset \vdash_1 (\nu a)([\![M]\!]^a \mid a?(x))}$$

Now consider any reduction $(\nu a)([\![M]\!]^a | a?(x)) \to^* (\nu \tilde{a})(P | a?(x))$ and observe that the reduct is well typed by Lemma III.3. Then from Theorem III.4(2) we deduce that there exists Q such that $(\nu \tilde{a})(P \mid a?(x)) \to^* (\nu \tilde{b})(Q \mid a!\langle \mathsf{v} \rangle \mid a?(x))$ where v is the value of M.

APPENDIX D
EXTENDED EXAMPLE: MULTIPARTY SESSIONS

In this section we show that a whole class of protocols whose lock freedom is guaranteed *by design* can be implemented by processes that are well-typed according to our type system for lock freedom. The challenging aspect of this exercise is that in general the processes will be recursive and interleaving possibly blocking actions pertaining several different linear channels configured in a cyclic network topology. We proceed as follows. First of all, we define a language for specifying protocols whose lock freedom is granted. We do so by means of global types [1], [9], [10], [13] describing sequences of *interaction events* between a fixed number of *participants*. Then, we extract from global types a characteristic implementation of the participants involved in the protocol, using the process language we have defined in Section II. Finally, we show how to construct a typing environment for the participants so that they are well typed according to the type discipline of Section III.

Extensions. For technical convenience, we work with a slightly more general syntax of types:

$$t ::= \cdots \mid \mu\alpha_k\{\alpha_i := t_i\}_{i=1..n} \mid \$^{n,m}\, t$$

Basically we generalize recursions so that the type $\mu\alpha_k\{\alpha_i := t_i\}_{i\in I}$ simultaneously binds *all* the type variables α_i for $i \in I$ in *each* t_i for $i \in I$. The elected type variable α_k with $k \in I$ stands for the whole term. It is known that this generalized recursion operator has the same expressive power as the one that binds exactly one variable [6], but it is more convenient to work with in this section. The notion of type contractiveness can be easily adjusted to this generalization. Equality between types works as before by considering the infinite unfolding of recursions, in particular, $\mu\alpha_k\{\alpha_i := t_i\}_{i\in I} = t_k\{\mu\alpha_i\{\alpha_i := t_i\}_{i\in I}/\alpha_i\}_{i\in I}$ where $t\{s_i/\alpha_i\}_{i\in I}$ denotes the simultaneous substitution of every free occurrence of α_i in t with s_i. We also consider $\$^{n,m}$ as a type constructor rather than a function, and we extend equality between types with the laws of (7). Because of contractiveness, every type can be rewritten into an equivalent one without $\$^{n,m}$ constructors. This constructor is necessary because we must be able to denote the shifting of *possibly open* types, whereas the shifting function (7) is defined for closed types only.

Global types. We assume a finite, totally ordered set $\mathscr{R}$ of *participant tags* $\mathtt{p}, \mathtt{q}, \ldots, \mathtt{A}, \mathtt{B}, \ldots$ that we use to uniquely identify the participants of a protocol. We also need a set of *labels* $\ell, \ldots$ for identifying events occurring in protocols. *Events* $\varepsilon, \ldots$ and *global types* $G, \ldots$ are defined below:

$$\varepsilon ::= \mathtt{p} \to \mathtt{q}@\ell \qquad G ::= \alpha \mid \varepsilon.G \mid \varepsilon.\{G, G\} \mid \mu\alpha.G$$

An event $\mathtt{p} \to \mathtt{q}@\ell$ represents the communication of a message from participant $\mathtt{p}$ (the sender) to participant $\mathtt{q}$ (the receiver). We omit the type of the message, because it is irrelevant for our purposes, and we will think of events merely as synchronizations. We decorate events with a label ℓ that uniquely identifies them, although we will see that the same label may occur several times in a given global type. The global type $\varepsilon.G$ describes a protocol in which ε *may* occur before all the other events in G (the assurance that it will *necessarily* occur before them depends on other properties of the global type that are not captured by their syntactic structure). The global type $\varepsilon.\{G_1, G_2\}$ describes a protocol in which the sender in ε communicates a binary decision to the receiver in ε. Depending on the outcome of the decision, the protocol evolves as either G_1 or G_2. Global types α and $\mu\alpha.G$ are used for building recursive protocols. We do not impose any contractiveness conditions on global types, which we treat as purely syntactic objects. In fact, we abbreviate $\mu\alpha.\alpha$ as $\mathtt{end}$ to denote the terminated protocol in which no event occurs. We write $\mathsf{fv}(G)$ for the set of free type variables in G, defined as expected.

Some global types are unreasonable. For example,

$$G_a \stackrel{\text{def}}{=} \mathtt{A} \to \mathtt{B}.\{\mathtt{C} \to \mathtt{D}.\mathtt{end}, \mathtt{D} \to \mathtt{C}.\mathtt{end}\}$$

specifies that either $\mathtt{C}$ sends a message to $\mathtt{D}$ or $\mathtt{D}$ sends a message to $\mathtt{C}$ depending on the decision taken by $\mathtt{A}$. The problem is that neither $\mathtt{C}$ nor $\mathtt{D}$ are aware of such decision, for $\mathtt{A}$ communicates it to $\mathtt{B}$, so there is no way $\mathtt{C}$ and $\mathtt{D}$ can behave differently in the two branches of the choice. We therefore need to identify a class of global types that describe sensible protocols, and that

we call *realizable*. An *edge* $\mathtt{p} \between \mathtt{q}$ is a set $\{\mathtt{p}, \mathtt{q}\}$ with $\mathtt{p} \neq \mathtt{q}$ representing a communication channel between two participants. We let $\mathsf{edge}(\mathtt{p} \to \mathtt{q}@\ell) = \mathtt{p} \between \mathtt{q}$. A *sort* is a set of edges and tells us which pairs of participants are supposed to communicate. Since we will analyze possibly open global types, we define the sort of a global type relative to a sort environment that maps type variables to sorts:

Definition D.1. A *sort environment* Σ is a finite map from type variables to sorts. We say that Σ is a sort environment for G if $\mathsf{fv}(G) \subseteq \mathsf{dom}(\Sigma)$. The *sort* of G relative to a sort environment Σ for G is $\Sigma(G) \stackrel{\text{def}}{=} \{\mathsf{edge}(\varepsilon) \mid \varepsilon \text{ occurs in } G\} \cup \bigcup_{\alpha \in \mathsf{fv}(G)} \Sigma(\alpha)$.

We write $\emptyset$ for the empty sort environment and $\mathsf{upd}(\Sigma, \alpha, G)$ for the updated sort environment $\Sigma, \alpha : \Sigma(\mu\alpha.G)$. We can now define the *projection* $\downarrow(\Sigma, G, \mathtt{p} \between \mathtt{q})$ of a global type G with respect to a sort environment Σ and an edge $\mathtt{p} \between \mathtt{q}$. Intuitively, $\downarrow(\Sigma, G, \mathtt{p} \between \mathtt{q})$ extracts from G the sub-protocol that concerns only participants $\mathtt{p}$ and $\mathtt{q}$. Formally:

$$
\begin{array}{rcll}
\downarrow(\Sigma, \alpha, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \alpha & \\
\downarrow(\Sigma, \varepsilon.G, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \varepsilon.\downarrow(\Sigma, G, \mathtt{p} \between \mathtt{q}) & \text{if } \mathtt{p} \between \mathtt{q} = \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.G, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \downarrow(\Sigma, G, \mathtt{p} \between \mathtt{q}) & \text{if } \mathtt{p} \between \mathtt{q} \neq \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.\{G_1, G_2\}, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \varepsilon.\left\{\begin{array}{l}\downarrow(\Sigma, G_1, \mathtt{p} \between \mathtt{q}) \\ \downarrow(\Sigma, G_2, \mathtt{p} \between \mathtt{q})\end{array}\right\} & \text{if } \mathtt{p} \between \mathtt{q} = \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.\{G_1, G_2\}, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \downarrow(\Sigma, G_i, \mathtt{p} \between \mathtt{q}) & \text{if } \mathtt{p} \between \mathtt{q} \neq \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \mu\alpha.G, \mathtt{p} \between \mathtt{q}) & \stackrel{\text{def}}{=} & \mu\alpha.\downarrow(\mathsf{upd}(\Sigma, \alpha, G), G, \mathtt{p} \between \mathtt{q}) &
\end{array}
$$

Note that projection is a *partial* function. It may be ill defined if the projections of two branches of a choice with respect to the same edge $\mathtt{p} \between \mathtt{q}$ differ (fifth equation). For example, $\downarrow(\emptyset, G_a, \mathtt{C} \between \mathtt{D})$ is ill defined because it can be either $\mathtt{C} \to \mathtt{D}.\mathtt{end}$ or $\mathtt{D} \to \mathtt{C}.\mathtt{end}$. Note the role played by labels in determining the existence of a projection. For example,

$$
G_b \stackrel{\text{def}}{=} \mathtt{A} \to \mathtt{B}@\ell_1.\left\{\begin{array}{l}\mathtt{B} \to \mathtt{A}@\ell_2.\mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{end} \\ \mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{B} \to \mathtt{A}@\ell_4.\mathtt{end}\end{array}\right\}
$$

is projectable with respect to $\mathtt{C} \between \mathtt{D}$ because the two occurrences of $\mathtt{C} \to \mathtt{D}@\ell_3$ in the two branches have the same label. They must in fact represent the same event, since $\mathtt{C}$ and $\mathtt{D}$ are unaware of the choice taken by $\mathtt{A}$. Conversely, $\mathtt{B} \to \mathtt{A}@\ell_2$ and $\mathtt{B} \to \mathtt{A}@\ell_4$ may have different labels because both $\mathtt{A}$ and $\mathtt{B}$ are aware of the choice taken by $\mathtt{A}$.

Projectability alone is not a sufficient condition for realizability. We must also take into account the ordering of events induced by the structure of global types. For instance

$$
G_c \stackrel{\text{def}}{=} \mathtt{A} \to \mathtt{B}@\ell_1.\left\{\begin{array}{l}\mathtt{E} \to \mathtt{F}@\ell_2.\mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{end} \\ \mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{E} \to \mathtt{F}@\ell_2.\mathtt{end}\end{array}\right\}
$$

is projectable but not realizable: the two occurrences of $\mathtt{C} \to \mathtt{D}$ and $\mathtt{E} \to \mathtt{F}$ must be tagged with the same label in order for $\downarrow(\emptyset, G_c, \mathtt{C} \between \mathtt{D})$ and $\downarrow(\emptyset, G_c, \mathtt{E} \between \mathtt{D})$ to be well defined, and yet G_c specifies that the order of these events depends on the decision taken by $\mathtt{A}$, which none of $\mathtt{C}$, $\mathtt{D}$, $\mathtt{E}$, and $\mathtt{F}$ is aware of. To capture these inconsistencies we reason on the relation $\prec_G$ induced by the structure of G such that $\ell \prec_G \ell'$ holds if and only if the event with label ℓ immediately precedes the event with label ℓ' in G. We denote by $\prec_G^+$ the transitive closure of $\prec_G$ and we require $\prec_G^+$ to be irreflexive for G to be realizable.

Definition D.2 (realizable global type)**.** We say that G is *realizable* if **(1)** $\downarrow(\emptyset, G, \mathtt{p} \between \mathtt{q})$ is well defined for every $\mathtt{p} \between \mathtt{q}$ and **(2)** the relation $\prec_G^+$ is irreflexive.

According to this definition, G_b is realizable but G_c is not, because $\ell_2 \prec_{G_c}^+ \ell_2$. Note that the relation $\prec_G$ is solely determined by the syntactic structure of G regardless of recursions and type variables occurring in G. No non-trivial recursive global type would be realizable had we considered, for example, $\mu\alpha.\mathtt{A} \to \mathtt{B}@\ell.\alpha$ equivalent to its own unfolding.

Characteristic participants. We now show how to build a set of processes that implement any realizable global type. Every participant $\mathtt{p}$ in a global type G will be modeled as a process $\mathbf{P}(\emptyset, G, \mathtt{p})$ (which will possibly fork into more sub-processes) that uses a tuple $\mathbf{x}_{[\mathtt{p}]} = x_{\mathtt{pq}_1}, \ldots, x_{\mathtt{pq}_n}$ of variables, where $\{\mathtt{q}_1, \ldots, \mathtt{q}_n\} = \mathscr{R} \setminus \{\mathtt{p}\}$ and the $\mathtt{q}_i$'s appear according to the total order on $\mathscr{R}$ (the total order serves only so that $\mathbf{x}_{[\mathtt{p}]}$ is a uniquely determined tuple). In particular, $x_{\mathtt{pq}}$ is the channel that connects $\mathtt{p}$ to $\mathtt{q}$. $\mathbf{P}(\Sigma, G, \mathtt{p})$ is defined inductively on G by the equations in the top half of Table 4. Despite the daunting look of the definition, the idea is simple: every event $\mathtt{A} \to \mathtt{B}$ to which $\mathtt{p}$ participates determines either an input or an output action on the channel $x_{\mathtt{pq}}$. When $\mathtt{p}$ is the receiver, a message is read from $x_{\mathtt{pq}}$ and is given name $x_{\mathtt{pq}}$, because the message is actually the continuation channel on which the conversation between $\mathtt{p}$ and $\mathtt{q}$ continues. When $\mathtt{p}$ is the sender, it sends to $\mathtt{q}$ the continuation for the subsequent communications on the edge $\mathtt{p} \between \mathtt{q}$. This is done through the auxiliary function $\mathbf{C}(\Sigma, \mathtt{p}, \mathtt{q}, id, G)$ where id is the identity function: the continuation is an actual channel a if $\mathtt{p} \between \mathtt{q}$ occurs in $\Sigma(G)$, that is if $\mathtt{p}$ and $\mathtt{q}$ will communicate again in the future, or the number 0 if no other interaction between $\mathtt{p}$ and $\mathtt{q}$ is expected. When the event implies a choice $\varepsilon.\{G_1, G_2\}$, we make the (arbitrary) decision that the sender always chooses the left branch G_1. We use injection to encode the decision in the sender and `case` to decode the decision in the receiver and we use the same auxiliary function $\mathbf{C}$ for sending

TABLE 4: Global type projections.

$$
\begin{array}{rll}
\mathbf{P}(\Sigma, \alpha, \mathtt{p}) & \stackrel{\text{def}}{=} & c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.G, \mathtt{A}) & \stackrel{\text{def}}{=} & \mathbf{C}(\Sigma, \mathtt{A}, \mathtt{B}, id, G) \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.G, \mathtt{B}) & \stackrel{\text{def}}{=} & x_{\mathtt{BA}}?(x_{\mathtt{BA}}).\mathbf{P}(\Sigma, G, \mathtt{B}) \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.G, \mathtt{p}) & \stackrel{\text{def}}{=} & \mathbf{P}(\Sigma, G, \mathtt{p}) & \mathtt{p} \notin \mathtt{A} \wr \mathtt{B} \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.\{G_1, G_2\}, \mathtt{A}) & \stackrel{\text{def}}{=} & \mathbf{C}(\Sigma, \mathtt{A}, \mathtt{B}, \mathtt{inl}, G_1) \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.\{G_1, G_2\}, \mathtt{B}) & \stackrel{\text{def}}{=} & x_{\mathtt{BA}}?(y).\mathtt{case}\ y\ \{\mathtt{inl}\ x_{\mathtt{BA}} \Rightarrow \mathbf{P}(\Sigma, G_1, \mathtt{B}), \mathtt{inr}\ x_{\mathtt{BA}} \Rightarrow \mathbf{P}(\Sigma, G_2, \mathtt{B})\} \\
\mathbf{P}(\Sigma, \mathtt{A} \to \mathtt{B}.\{G_1, G_2\}, \mathtt{p}) & \stackrel{\text{def}}{=} & \mathbf{P}(\Sigma, G_i, \mathtt{p}) & \mathtt{p} \notin \mathtt{A} \wr \mathtt{B} \\
\mathbf{P}(\Sigma, \mu\alpha.G, \mathtt{p}) & \stackrel{\text{def}}{=} & (\nu c_\alpha)(c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle \mid *c_\alpha?(\mathbf{x}_{[\mathtt{p}]}).\mathbf{P}(\mathsf{upd}(\Sigma, \alpha, G), G, \mathtt{p})) \\
\mathbf{C}(\Sigma, \mathtt{p}, \mathtt{q}, f, G) & \stackrel{\text{def}}{=} & (\nu a)(x_{\mathtt{pq}}!\langle f\ a \rangle \mid \mathtt{let}\ x_{\mathtt{pq}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G, \mathtt{p})) & \mathtt{p} \wr \mathtt{q} \in \Sigma(G) \\
\mathbf{C}(\Sigma, \mathtt{p}, \mathtt{q}, f, G) & \stackrel{\text{def}}{=} & x_{\mathtt{pq}}!\langle f\ 0 \rangle \mid \mathtt{let}\ x_{\mathtt{pq}} = 0\ \mathtt{in}\ \mathbf{P}(\Sigma, G, \mathtt{p}) & \mathtt{p} \wr \mathtt{q} \notin \Sigma(G)
\end{array}
$$

$$
\begin{array}{rll}
\mathbf{T}(\Sigma, \alpha, \mathtt{p}, \mathtt{q}, n) & \stackrel{\text{def}}{=} & \begin{cases} \$^{0,1}\ \alpha_{\mathtt{pq}} & \mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha) \\ \mathtt{int} & \text{otherwise} \end{cases} \\
\mathbf{T}(\Sigma, \mathtt{A} \to \mathtt{B}@\ell.G, \mathtt{p}, \mathtt{q}, n) & \stackrel{\text{def}}{=} & \begin{cases} \$^{\ell-n,0}\ \mathtt{!}[\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, \ell)] & \mathtt{p}, \mathtt{q} = \mathtt{A}, \mathtt{B} \\ \$^{\ell-n,0}\ \mathtt{?}[\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, \ell)] & \mathtt{p}, \mathtt{q} = \mathtt{B}, \mathtt{A} \\ \$^{\ell-n,0}\ \mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, \ell) & \text{otherwise} \end{cases} \\
\mathbf{T}(\Sigma, \mathtt{A} \to \mathtt{B}@\ell.\{G_1, G_2\}, \mathtt{p}, \mathtt{q}, n) & \stackrel{\text{def}}{=} & \begin{cases} \$^{\ell-n,0}\ \mathtt{!}[\mathbf{T}(\Sigma, G_1, \mathtt{q}, \mathtt{p}, \ell) \oplus \mathbf{T}(\Sigma, G_2, \mathtt{q}, \mathtt{p}, \ell)] & \mathtt{p}, \mathtt{q} = \mathtt{A}, \mathtt{B} \\ \$^{\ell-n,0}\ \mathtt{?}[\mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, \ell) \oplus \mathbf{T}(\Sigma, G_2, \mathtt{p}, \mathtt{q}, \ell)] & \mathtt{p}, \mathtt{q} = \mathtt{B}, \mathtt{A} \\ \$^{\ell-n,0}\ \mathbf{T}(\Sigma, G_i, \mathtt{p}, \mathtt{q}, \ell) & \text{otherwise} \end{cases} \\
\mathbf{T}(\Sigma, \mu\alpha.G, \mathtt{p}, \mathtt{q}, n) & \stackrel{\text{def}}{=} & \$^{0,1}\ \mu\alpha_{\mathtt{pq}} \left\{ \begin{array}{l} \alpha_{\mathtt{pq}} = \mathbf{T}(\mathsf{upd}(\Sigma, \alpha, G), G, \mathtt{p}, \mathtt{q}, n) \\ \alpha_{\mathtt{qp}} = \mathbf{T}(\mathsf{upd}(\Sigma, \alpha, G), G, \mathtt{q}, \mathtt{p}, n) \end{array} \right\}
\end{array}
$$

the decision, to which we pass the injector `inl` to wrap the continuation. Recursions $\mu\alpha.G$ are implemented by associating the type variable α with an unlimited channel c_α: the replicated input on c_α corresponds to defining a recursive process; the output on c_α corresponds to invoking the recursive process. The whole tuple $\mathbf{x}_{[\mathtt{p}]}$ is transmitted at each invocation.

For instance, given $G_d \stackrel{\text{def}}{=} \mu\alpha.\mathtt{A} \to \mathtt{B}.\mathtt{B} \to \mathtt{C}.\mathtt{C} \to \mathtt{A}.\alpha$ and assuming $\mathscr{R} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, we have

$$
\begin{array}{l}
\mathbf{P}(\emptyset, G_d, \mathtt{A}) = \\
\quad (\nu c_\alpha)(c_\alpha!\langle x_{\mathtt{AB}}, x_{\mathtt{AC}} \rangle \mid \\
\qquad *c_\alpha?(x_{\mathtt{AB}}, x_{\mathtt{AC}}).(\nu a)(x_{\mathtt{AB}}!\langle a \rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in} \\
\qquad\qquad\qquad\qquad x_{\mathtt{AC}}?(x_{\mathtt{AC}}).c_\alpha!\langle x_{\mathtt{AB}}, x_{\mathtt{AC}} \rangle))
\end{array}
$$

Typing participants. We now arrive at the main point of this section, namely the definition of a suitable type environment $\Gamma_\mathtt{p} \stackrel{\text{def}}{=} \{x_{\mathtt{pq}} : t_{\mathtt{pq}}\}_{\mathtt{q} \in \mathscr{R} \setminus \{\mathtt{p}\}}$ such that the process $\mathbf{P}(\emptyset, G, \mathtt{p})$ is well typed in $\Gamma_\mathtt{p}$. Determining the input/output behavior of $\mathbf{P}(\emptyset, G, \mathtt{p})$ with respect to a channel $x_{\mathtt{pq}}$ is easy: this information is written in the global type. But for $\mathbf{P}(\emptyset, G, \mathtt{p})$ to be well typed, we must also make sure that blocking actions respect the priority of channels (see [T-IN]) and that delegations are allowed by the rank of channels (see [T-OUT] and [T-OUT*]). We determine priorities using the event order $\prec_G$ and ranks looking at the structure of $\mathbf{P}(\emptyset, G, \mathtt{p})$. We know that every realizable global type G has an irreflexive event order $\prec_G^+$, meaning that the events in G form a DAG. Therefore, there exists a topological ordering ord from labels in G to positive natural numbers such that $\ell \prec_G \ell'$ implies $0 < \mathsf{ord}(\ell) < \mathsf{ord}(\ell')$. In a sense, $\mathsf{ord}(\ell)$ is the abstract time at which the event labeled ℓ occurs. The idea is to use $\mathsf{ord}(\ell)$ as the priority for the channel types when projecting the action labeled by ℓ. In what follows, we simplify the notation writing just ℓ in place of $\mathsf{ord}(\ell)$.

Now, the bottom half of Table 4 defines $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)$ as the type of $x_{\mathtt{pq}}$ in the global type G relative to an arbitrary offset n that is smaller than any label occurring in G. Given how $\mathbf{P}(\Sigma, G, \mathtt{p})$ uses the channel $x_{\mathtt{pq}}$, the definition of $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)$ is hopefully self-explanatory except for a few twists that we comment on here: The type $\mathbf{T}(\Sigma, \alpha, \mathtt{p}, \mathtt{q}, n)$ is the type variable $\alpha_{\mathtt{pq}}$ only if the edge $\mathtt{p} \wr \mathtt{q}$ is used in the recursion marked by the type variable α; otherwise, it is `int` to denote a dummy natural number. Priorities are determined as the difference between the label ℓ of the topmost action in G and the current offset n; after the action, the offset becomes ℓ. We have $\mathbf{T}(\Sigma, \mathtt{A} \to \mathtt{B}@\ell.G, \mathtt{A}, \mathtt{B}, n) = \$^{\ell-n,0}\ \mathtt{!}[\mathbf{T}(\Sigma, G, \mathtt{B}, \mathtt{A}, \ell)]$ where $\mathtt{A}$ and $\mathtt{B}$ are *swapped* within $\mathtt{!}[\cdot]$. The swapping is motivated by the fact that $\mathtt{A}$ sends to $\mathtt{B}$ a channel whose type determines how the channel is used by $\mathtt{B}$. The same phenomenon occurs in determining the type of a choice. Then, because of the definition of $\mathbf{T}(\Sigma, \alpha, \mathtt{p}, \mathtt{q}, n)$, it may happen that the type $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)$ contains both the $\alpha_{\mathtt{pq}}$ and the $\alpha_{\mathtt{qp}}$ type variables. For this reason, $\mathbf{T}(\Sigma, \mu\alpha.G, \mathtt{p}, \mathtt{q}, n)$ binds both $\alpha_{\mathtt{pq}}$ and $\alpha_{\mathtt{qp}}$, taking advantage of the generalized recursion for channel types. We apply $\$^{0,1}$ in the two places where a channel is delegated, in accordance with the definition of $\mathbf{P}(\Sigma, G, \mathtt{p})$. Finally, note that $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)$ is well defined whenever G is realizable.

For example, considering again G_d above, we have

$$\mathbf{T}(\emptyset, G_d, \mathtt{A}, \mathtt{B}, 0) = \$^{0,1}\, \mu\alpha_{\mathtt{AB}}.\left\{\begin{array}{l}\alpha_{\mathtt{AB}} := \mathtt{!}[\$^{\ell_3-\ell_1,1}\, \alpha_{\mathtt{BA}}]^{\ell_1,0} \\ \alpha_{\mathtt{BA}} := \mathtt{?}[\$^{\ell_3-\ell_1,1}\, \alpha_{\mathtt{BA}}]^{\ell_1,0}\end{array}\right\}$$

assuming that ℓ_1, ℓ_2, and ℓ_3 label the three events in G_d.

The main result of this section states that the $\mathbf{P}(\emptyset, G, \mathtt{p})$'s are well typed and so is their parallel composition:

Theorem D.3. *Let G be realizable. Then for every $\mathtt{p} \in \mathscr{R}$ we have $\{x_{\mathtt{pq}} : \mathbf{T}(\emptyset, G, \mathtt{p}, \mathtt{q}, 0) \mid \mathtt{q} \in \mathscr{R} \setminus \{\mathtt{p}\}\} \vdash_1 \mathbf{P}(\emptyset, G, \mathtt{p})$.*

Note that a global type with two participants is always realizable. This corresponds to a binary session.

Appendix E
Proofs for Appendix D

Definition E.1. We write $n < G$ if $n < \mathsf{ord}(\ell)$ for every ℓ occurring in G.

Lemma E.2. *Let $n < G$ and $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = t$. Then:*

1) *$\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$ implies $t = \mathtt{int}$;*
2) *$\mathtt{p} \wr \mathtt{q} \notin G$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$ and $\alpha \in \mathsf{fv}(G)$ implies $t = \$^{h,k}\, \alpha_{\mathtt{pq}}$;*
3) *$\mathtt{p} \wr \mathtt{q} \in G$ implies $t = p[s]^{h,k}$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \overline{p}[s]^{h,k}$ for some $h > 0$.*

Proof: By induction on G.

- ($G = \alpha$) We have:
  1) ($\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$) Then $t = \mathtt{int}$ by definition of $\mathbf{T}$.
  2) ($\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$) Then $t = \$^{0,1}\, \alpha_{\mathtt{pq}}$ by definition of $\mathbf{T}$ and we conclude by taking $h = 0$ and $k = 1$.
  3) ($\mathtt{p} \wr \mathtt{q} \in G$) This case is impossible.
- ($G = \mathtt{A} \to \mathtt{B}@\ell.G'$) We have:
  1) ($\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$) Then $\mathtt{p} \wr \mathtt{q} \neq \mathtt{A} \wr \mathtt{B}$ and $\mathtt{p} \wr \mathtt{q} \notin \Sigma(G')$ and $t = \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = \$^{\ell-n,0}\, \mathtt{int} = \mathtt{int}$ by definition of $\mathbf{T}$, by induction hypothesis, and by equality on types.
  2) ($\mathtt{p} \wr \mathtt{q} \notin G$ and $\alpha \in \mathsf{fv}(G)$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(G)$) Then $\mathtt{p} \wr \mathtt{q} \neq \mathtt{A} \wr \mathtt{B}$ and $\mathtt{p} \wr \mathtt{q} \notin G'$ and $\alpha \in \mathsf{fv}(G')$. We have $t = \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = \$^{\ell-n,0}\, \$^{m,k}\, \alpha_{\mathtt{pq}} = \$^{\ell-n+m,k}\, \alpha_{\mathtt{pq}}$ by definition of $\mathbf{T}$ and by induction hypothesis. We conclude by taking $h = \ell - n + m$.
  3) ($\mathtt{p} \wr \mathtt{q} \in G$) Then either $\mathtt{p} \wr \mathtt{q} = \mathtt{A} \wr \mathtt{B}$ or $\mathtt{p} \wr \mathtt{q} \in G'$. If $\mathtt{p}, \mathtt{q} = \mathtt{A}, \mathtt{B}$, then $t = \mathtt{!}[s]^{h,k}$ where $h = \ell - n$ and $k = 0$ and $s = \mathbf{T}(\Sigma, G', \mathtt{q}, \mathtt{p}, \ell)$ and we conclude by observing that $h > 0$ from the hypothesis $n < G$. If $\mathtt{p}, \mathtt{q} = \mathtt{B}, \mathtt{A}$ we conclude similarly. If $\mathtt{p} \wr \mathtt{q} \neq \mathtt{A} \wr \mathtt{B}$, then by induction hypothesis we have $\mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = p[s]^{m,k}$ for some $m > 0$ and we conclude $t = p[s]^{h,k}$ by taking $h = \ell - n + m$.
- ($G = \mathtt{A} \to \mathtt{B}@\ell.\{G_1, G_2\}$) This case is similar to the previous one, the only interesting sub-case is when $\mathtt{p} \wr \mathtt{q} \notin G$ and $\alpha \in \mathsf{fv}(G)$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(G)$. By definition of global type projection we have $t = \mathbf{T}(\Sigma, G_i, \mathtt{p}, \mathtt{q}, \ell)$ for every $i = 1, 2$, therefore we deduce $\alpha \in \mathsf{fv}(G_1) \cap \mathsf{fv}(G_2)$. We conclude by induction hypothesis as in the previous case.
- ($G = G_1 \parallel G_2$) Assume, without loss of generality, that $\mathtt{p} \wr \mathtt{q} \notin \Sigma(G_2)$ and that $t = \mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, n)$. We have:
  1) ($\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$) We conclude $\mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, n) = \mathtt{int}$ by induction hypothesis;
  2) ($\mathtt{p} \wr \mathtt{q} \notin G$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$ and $\alpha \in \mathsf{fv}(G)$) Then $\mathtt{p} \wr \mathtt{q} \notin G_1$ and $\alpha \in \mathsf{fv}(G_1)$. By induction hypothesis we conclude $\mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, n) = \$^{h,k}\, \alpha_{\mathtt{pq}}$.
  3) ($\mathtt{p} \wr \mathtt{q} \in G$) Then $\mathtt{p} \wr \mathtt{q} \in G_1$ and we conclude by induction hypothesis.
- ($G = \mu\alpha.G'$) Let

$$\begin{array}{rcl}\Sigma' & \stackrel{\text{def}}{=} & \mathsf{upd}(\Sigma, \alpha, G') \\ t_{\mathtt{pq}} & \stackrel{\text{def}}{=} & \mu\alpha_{\mathtt{pq}}\{\alpha_{\mathtt{pq}} := \mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n), \alpha_{\mathtt{qp}} := \mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n)\} \\ t_{\mathtt{qp}} & \stackrel{\text{def}}{=} & \mu\alpha_{\mathtt{qp}}\{\alpha_{\mathtt{pq}} := \mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n), \alpha_{\mathtt{qp}} := \mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n)\} \\ \sigma & \stackrel{\text{def}}{=} & \{t_{\mathtt{pq}}, t_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\end{array}$$

  and observe that Σ' is a sort environment for G' and $t = \$^{0,1}\, t_{\mathtt{pq}} = \$^{0,1}\, \mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n)\sigma$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \$^{0,1}\, t_{\mathtt{qp}} = \$^{0,1}\, \mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n)\sigma$. We distinguish three sub-cases:
  1) ($\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$) Then $\mathtt{p} \wr \mathtt{q} \notin \Sigma'(G')$ and we conclude $t = \mathtt{int}$.
  2) ($\mathtt{p} \wr \mathtt{q} \notin G$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(\beta)$ and $\beta \in \mathsf{fv}(G)$) Then $\mathtt{p} \wr \mathtt{q} \notin G'$ and $\beta \neq \alpha$ and $\beta \in \mathsf{fv}(G')$. By induction hypothesis we deduce $\mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n) = \$^{h,m}\, \beta_{\mathtt{pq}}$. We conclude by taking $k = m + 1$ and observing that $t = \$^{h,k}\, \beta_{\mathtt{pq}}$.
  3) ($\mathtt{p} \wr \mathtt{q} \in G$) Then $\mathtt{p} \wr \mathtt{q} \in G'$ and by induction hypothesis we deduce $\mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n) = p[s]^{h,m}$ and $\mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n) = \overline{p}[s]^{h,m}$ for some $h > 0$. We conclude by taking $k = m + 1$ and observing that $t = p[s\sigma]^{h,k}$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \overline{p}[s\sigma]^{h,k}$.

□

Definition E.3 (Σ-substitution)**.** A Σ-substitution is a finite map σ such that:

1) $\mathsf{dom}(\sigma) = \{\alpha_{\mathtt{pq}} \mid \alpha \in \mathsf{dom}(\Sigma) \wedge \mathtt{p} \wr \mathtt{q} \subseteq \mathscr{R}\}$;
2) $\sigma(\alpha_{\mathtt{pq}}) = \sigma(\alpha_{\mathtt{qp}}) = \mathtt{int}$ for every $\mathtt{p} \wr \mathtt{q} \notin \Sigma(\alpha)$;
3) $\sigma(\alpha_{\mathtt{pq}}) = p[t]^{m,n}$ and $\sigma(\alpha_{\mathtt{qp}}) = \overline{p}[t]^{m,n}$ and $m > 0$ for every $\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$.

Lemma E.4. *If* $\Sigma' = \Sigma, \alpha : \Sigma(\mu\alpha.G)$*, then* $\Sigma(\mu\alpha.G) = \Sigma'(G)$.

Proof: We have

$$\begin{array}{rcll}
\Sigma(\mu\alpha.G) & = & \Sigma(\mu\alpha.G) \underbrace{\cup\, \Sigma(\mu\alpha.G)}_{\alpha \in \mathsf{fv}(G)} & \text{property of set union} \\
& = & \mathsf{sort}(\mu\alpha.G) \cup \bigcup_{\beta \in \mathsf{fv}(\mu\alpha.G)} \Sigma(\beta) \underbrace{\cup\, \Sigma(\mu\alpha.G)}_{\alpha \in \mathsf{fv}(G)} & \text{definition of } \Sigma(\mu\alpha.G) \\
& = & \mathsf{sort}(\mu\alpha.G) \cup \bigcup_{\beta \in \mathsf{fv}(G)} \Sigma'(\beta) & \mathsf{fv}(\mu\alpha.G) = \mathsf{fv}(G) \setminus \{\alpha\} \\
& = & \mathsf{sort}(G) \cup \bigcup_{\beta \in \mathsf{fv}(G)} \Sigma'(\beta) & \mathsf{sort}(\mu\alpha.G) = \mathsf{sort}(G) \\
& = & \Sigma'(G) & \text{definition of } \Sigma'(G)
\end{array}$$

□

Lemma E.5. *Let*

- σ *be a* Σ*-substitution;*
- $n < G$*;*
- $\Sigma' = \Sigma, \alpha : \Sigma(\mu\alpha.G)$*;*
- $s_{\mathtt{pq}} = \mu\alpha_{\mathtt{pq}}.\{\alpha_{\mathtt{pq}} := \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n), \alpha_{\mathtt{qp}} := \mathbf{T}(\Sigma', G, \mathtt{q}, \mathtt{p}, n)\}$*;*
- $\sigma' = \sigma, \{s_{\mathtt{pq}}\sigma/\alpha_{\mathtt{pq}}\}_{\mathtt{p} \wr \mathtt{q} \subseteq \mathscr{R}}$.

Then σ' *is a* Σ'*-substitution.*

Proof: We must prove the three conditions of Definition E.3. The first condition is satisfied because $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{\alpha_{\mathtt{pq}} \mid \mathtt{p} \wr \mathtt{q} \subseteq \mathscr{R}\}$, so let us consider conditions 2 and 3. For every type variable $\beta \neq \alpha$ these conditions are trivially satisfied from the hypothesis that σ is a Σ-substitution, therefore we focus on α. Observe that, by Lemma E.4, we have $\Sigma'(\alpha) = \Sigma(\mu\alpha.G) = \Sigma'(G)$.

Regarding condition 2 of Definition E.3, assume $\mathtt{p} \wr \mathtt{q} \notin \Sigma'(\alpha) = \Sigma'(G)$. We conclude

$$\begin{array}{rcll}
\sigma'(\alpha_{\mathtt{pq}}) & = & s_{\mathtt{pq}}\sigma & \text{by definition of } \sigma' \\
& = & \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by unfolding of } s_{\mathtt{pq}} \\
& = & \mathtt{int}\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by Lemma E.2} \\
& = & \mathtt{int} & \text{by definition of type substitution}
\end{array}$$

Regarding condition 3 of Definition E.3, assume $\mathtt{p} \wr \mathtt{q} \in \Sigma'(\alpha) = \Sigma'(G)$. We distinguish two sub-cases:

- ($\mathtt{p} \wr \mathtt{q} \notin G$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma'(\beta)$ and $\beta \in \mathsf{fv}(G)$) From $\mathtt{p} \wr \mathtt{q} \notin G$ we deduce $\beta \neq \alpha$, so

$$\begin{array}{rcll}
\sigma'(\alpha_{\mathtt{pq}}) & = & \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by definition of } \sigma' \\
& = & \$^{h,k}\, \beta_{\mathtt{pq}}\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by Lemma E.2(2)} \\
& = & \$^{h,k}\, \sigma(\beta_{\mathtt{pq}}) & \text{because } \beta \neq \alpha \\
& = & \$^{h,k}\, p[s]^{h',k'} & \text{because } \sigma \text{ is a } \Sigma\text{-substitution} \\
& = & p[s]^{h+h',k+k'} & \text{equality on types}
\end{array}$$

  and similarly we obtain $\sigma'(\alpha_{\mathtt{pq}}) = \overline{p}[s]^{h+h',k+k'}$.
- ($\mathtt{p} \wr \mathtt{q} \in G$) We have

$$\begin{array}{rcll}
\sigma'(\alpha_{\mathtt{pq}}) & = & \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by definition of } \sigma' \\
& = & p[t]^{h,k} & \text{by Lemma E.2(3)}
\end{array}$$

  and similarly we obtain $\sigma'(\alpha_{\mathtt{qp}}) = \overline{p}[t]^{h,k}$.

□

Lemma E.6. *Let* $n < G$ *and* σ *be a* Σ*-substitution. Then:*

- $\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$ *implies* $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)\sigma = \mathtt{int}$*;*
- $\mathtt{p} \wr \mathtt{q} \in \Sigma(G)$ *implies* $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n)\sigma = p[t]^{h,k}$ *and* $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n)\sigma = \overline{p}[t]^{h,k}$ *and* $h > 0$.

Proof: If $\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$, then by Lemma E.2 we deduce $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = \mathtt{int}$ and we conclude immediately. If $\mathtt{p} \wr \mathtt{q} \in \Sigma(G)$, then we distinguish two sub-cases:

- ($\mathtt{p} \wr \mathtt{q} \in G$) By Lemma E.2 we have $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = p[s]^{h,k}$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \overline{p}[s]^{h,k}$ and $h > 0$. We conclude by taking $t = s\sigma$.
- ($\mathtt{p} \wr \mathtt{q} \notin G$ and $\alpha \in \mathsf{fv}(G)$ and $\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$) By Lemma E.2 we have $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = \$^{h',k'}\, \alpha_{\mathtt{pq}}$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \$^{h',k'}\, \alpha_{\mathtt{qp}}$. By definition of Σ-substitution we deduce $\sigma(\alpha_{\mathtt{pq}}) = p[s]^{h'',k''}$ and $\sigma(\alpha_{\mathtt{qp}}) = \overline{p}[s]^{h'',k''}$ and $h'' > 0$. We conclude by taking $h = h' + h''$ and $k = k' + k''$.

□

We introduce the following abbreviations for referring to the environments used in the typing derivations of the following lemma:

$$\begin{array}{rcl} \Theta_{[\Sigma,\mathtt{p}]} & \stackrel{\text{def}}{=} & \{c_\alpha : !^\omega[\alpha_{[\mathtt{p}]}] \mid \alpha \in \mathsf{dom}(\Sigma)\} \\ \Gamma_{[\Sigma,G,\mathtt{p},n]} & \stackrel{\text{def}}{=} & \{x_{\mathtt{pq}} : \mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) \mid \mathtt{q} \in \mathscr{R} \setminus \{\mathtt{p}\}\} = \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G, [\mathtt{p}], n) \end{array}$$

Lemma E.7. *Let σ be a Σ-substitution and $n < G$. Then $\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 \mathbf{P}(\Sigma, G, \mathtt{p})$.*

Proof: By induction on G. We omit the cases for choices, since they are analogous to the ones for interactions.

$\boxed{G = \alpha}$ Let $s_{\mathtt{pq}} \stackrel{\text{def}}{=} \sigma(\alpha_{\mathtt{pq}})$. We have:

$$\begin{array}{rcl} \mathbf{P}(\Sigma, G, \mathtt{p}) & = & c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle \\ \Theta_{[\Sigma,\mathtt{p}]}\sigma & \ni & c_\alpha : !^\omega[\alpha_{[\mathtt{p}]}]\sigma \quad = \quad c_\alpha : !^\omega[s_{[\mathtt{p}]}] \\ \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma & = & \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G, [\mathtt{p}], n)\sigma \quad = \quad \mathbf{x}_{[\mathtt{p}]} : \$^{0,1}\, s_{[\mathtt{p}]} \end{array}$$

We derive:

$$\frac{\Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash \mathbf{x}_{[\mathtt{p}]} : \$^{0,1}\, s_{[\mathtt{p}]}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle}\ [\text{T-OUT*}]$$

where we use the fact that $0 \leq |(|s_{\mathtt{pq}})$ which follows from the hypothesis that σ is a Σ-substitution.

$\boxed{G = \mu\alpha.G'}$ Let

$$\begin{array}{rcl} \Sigma' & \stackrel{\text{def}}{=} & \mathsf{upd}(\Sigma, \alpha, G') = \Sigma, \alpha : \Sigma(G) \\ s_{[\mathtt{p}]} & \stackrel{\text{def}}{=} & s_{\mathtt{pq}_1} \times \cdots \times s_{\mathtt{pq}_n} \\ s_{\mathtt{pq}} & \stackrel{\text{def}}{=} & \mu\alpha_{\mathtt{pq}}\{\alpha_{\mathtt{pq}} := \mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n), \alpha_{\mathtt{qp}} := \mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n)\} \\ \sigma' & \stackrel{\text{def}}{=} & \sigma \circ \{s_{[\mathtt{p}]}/\alpha_{[\mathtt{p}]}\} \end{array}$$

and observe that σ' is a Σ'-substitution by Lemma E.5. We have:

$$\begin{array}{rcl} \mathbf{P}(\Sigma, G, \mathtt{p}) & = & (\nu c_\alpha)(c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle \mid *c_\alpha?(\mathbf{x}_{[\mathtt{p}]}).\mathbf{P}(\Sigma', G', \mathtt{p})) \\ \Theta_{[\Sigma',\mathtt{p}]}\sigma' & = & (\Theta_{[\Sigma,\mathtt{p}]}, c_\alpha : !^\omega[\alpha_{[\mathtt{p}]}])\sigma' \quad = \quad \Theta_{[\Sigma,\mathtt{p}]}\sigma, c_\alpha : !^\omega[s_{[\mathtt{p}]}]\sigma \\ \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma & = & \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G, [\mathtt{p}], n)\sigma \quad = \quad \mathbf{x}_{[\mathtt{p}]} : \$^{0,1}\, s_{[\mathtt{p}]}\sigma \\ \Gamma_{[\Sigma',G',\mathtt{p},n]}\sigma' & = & \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma', G', [\mathtt{p}], n)\sigma' \quad = \quad \mathbf{x}_{[\mathtt{p}]} : s_{[\mathtt{p}]}\sigma \end{array}$$

We derive:

$$\dfrac{\dfrac{\Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash \mathbf{x}_{[\mathtt{p}]} : \$^{0,1}\, s_{[\mathtt{p}]}\sigma}{c_\alpha : !^\omega[s_{[\mathtt{p}]}]\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle}\ [\text{T-OUT*}] \quad \dfrac{\dfrac{\vdots}{\Theta_{[\Sigma',\mathtt{p}]}\sigma', \Gamma_{[\Sigma',G',\mathtt{p},n]}\sigma' \vdash_1 \mathbf{P}(\Sigma', G', \mathtt{p})}\ \text{IND.HYP.}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, c_\alpha : \#^\omega[s_{[\mathtt{p}]}]\sigma \vdash_1 *c_\alpha?(\mathbf{x}_{[\mathtt{p}]}).\mathbf{P}(\Sigma', G', \mathtt{p})}\ [\text{T-IN*}]}{\dfrac{\Theta_{[\Sigma,\mathtt{p}]}\sigma, c_\alpha : \#^\omega[s_{[\mathtt{p}]}]\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 c_\alpha!\langle \mathbf{x}_{[\mathtt{p}]} \rangle \mid *c_\alpha?(\mathbf{x}_{[\mathtt{p}]}).\mathbf{P}(\Sigma', G', \mathtt{p})}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 \mathbf{P}(\Sigma, G, \mathtt{p})}\ [\text{T-NEW}]}\ [\text{T-PAR}]$$

$\boxed{G = \mathtt{A} \to \mathtt{B}@\ell.G' \text{ and } \mathtt{p} = \mathtt{A}}$ We consider the case $\mathtt{A} \wr \mathtt{B} \in \Sigma(G')$, the other being analogous. Then, from Lemma E.6, we know that

$$\begin{array}{rcl} \mathbf{T}(\Sigma, G', \mathtt{A}, \mathtt{B}, \ell)\sigma & = & p[s]^{h,k} \\ \mathbf{T}(\Sigma, G', \mathtt{B}, \mathtt{A}, \ell)\sigma & = & \overline{p}[s]^{h,k} \end{array}$$

for some $h, k \in \mathbb{N}$, $p \in \{?, !\}$ such that $h > 0$, and s. We have:

$$\begin{array}{rcl} \mathbf{P}(\Sigma, G, \mathtt{p}) & = & (\nu a)(x_{\mathtt{AB}}!\langle a \rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p})) \\ \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma & = & \{x_{\mathtt{pq}} : \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell)\sigma\}_{\mathtt{q} \neq \mathtt{B}}, x_{\mathtt{AB}} : !\,[\overline{p}[s]^{h,k}]^{\ell-n,0} \\ \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma & = & \{x_{\mathtt{pq}} : \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell)\sigma\}_{\mathtt{q} \neq \mathtt{B}}, x_{\mathtt{AB}} : p[s]^{h,k} \end{array}$$

We derive $\mathscr{A}$:

$$\frac{a : \overline{p}[s]^{\ell-n+h,k+1} \vdash a : \overline{p}[s]^{\ell-n+h,k+1}}{x_{\mathtt{AB}} : \mathtt{!}\,[\overline{p}[s]^{h,k}]^{\ell-n,0}, a : \overline{p}[s]^{\ell-n+h,k+1} \vdash_1 x_{\mathtt{AB}}!\langle a\rangle}\ [\text{T-OUT}]$$

as well as $\mathscr{B}$:

$$\frac{\dfrac{\dfrac{\vdots}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma \vdash_1 \mathbf{P}(\Sigma, G', \mathtt{p})}\ \text{IND.HYP.}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \{x_{\mathtt{pq}} : \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, \ell)\sigma\}_{\mathtt{q}\neq\mathtt{B}}, x_{\mathtt{AB}} : p[s]^{\ell-n+h,k} \vdash_1 \mathbf{P}(\Sigma, G', \mathtt{p})}\ [\text{T-LIFT}]}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \{x_{\mathtt{pq}} : \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, \sigma)\}_{\mathtt{q}\neq\mathtt{B}}, a : p[s]^{\ell-n+h,k} \vdash_1 \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p})}\ [\text{T-LET}]$$

From these we obtain:

$$\frac{\dfrac{\mathscr{A} \qquad \mathscr{B}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma, a : \#[s]^{\ell-n+h,2k+1} \vdash_1 x_{\mathtt{AB}}!\langle a\rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p})}\ [\text{T-PAR}]}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 (\nu a)(x_{\mathtt{AB}}!\langle a\rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p}))}\ [\text{T-NEW}]$$

$\boxed{G = \mathtt{A} \to \mathtt{B}@\ell.G' \text{ and } \mathtt{p} = \mathtt{B}}$ We have:

$$\begin{array}{lcl} \mathbf{P}(\Sigma, G, \mathtt{p}) & = & x_{\mathtt{BA}}?(x_{\mathtt{BA}}).\mathbf{P}(\Sigma, G', \mathtt{p}) \\ \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma & = & \{x_{\mathtt{pq}} : \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell)\sigma\}_{\mathtt{q}\neq\mathtt{A}}, x_{\mathtt{BA}} : ?[\mathbf{T}(\Sigma, G', \mathtt{B}, \mathtt{A}, \ell)\sigma]^{\ell-n,0} \\ \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma & = & \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G', [\mathtt{p}], \ell)\sigma \end{array}$$

We derive:

$$\frac{\dfrac{\dfrac{\vdots}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma \vdash_1 \mathbf{P}(\Sigma, G', \mathtt{p})}\ \text{IND.HYP.}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \{x_{\mathtt{pq}} : \mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell)\sigma\}_{\mathtt{q}\neq\mathtt{A}}, x_{\mathtt{BA}} : ?[\mathbf{T}(\Sigma, G', \mathtt{B}, \mathtt{A}, \ell)\sigma]^{0,0} \vdash_1 x_{\mathtt{BA}}?(x_{\mathtt{BA}}).\mathbf{P}(\Sigma, G', \mathtt{p})}\ [\text{T-IN}]}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 \mathbf{P}(\Sigma, G, \mathtt{p})}\ [\text{T-LIFT}]$$

where the application of [T-IN] uses Lemma E.6 for deducing that the subject ranks of all the linear types in the environment are strictly positive.

$\boxed{G = \mathtt{A} \to \mathtt{B}@\ell.G' \text{ and } \mathtt{p} \notin \mathtt{A} \wr \mathtt{B}}$ We have:

$$\begin{array}{lcl} \mathbf{P}(\Sigma, G, \mathtt{p}) & = & \mathbf{P}(\Sigma, G', \mathtt{p}) \\ \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma & = & \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G, [\mathtt{p}], n)\sigma = \mathbf{x}_{[\mathtt{p}]} : \$^{\ell-n,0}\, \mathbf{T}(\Sigma, G', [\mathtt{p}], \ell)\sigma = \$^{\ell-n,0}\, \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma \end{array}$$

We derive:

$$\frac{\dfrac{\vdots}{\Theta_{[\Sigma',\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma \vdash_1 \mathbf{P}(\Sigma, G', \mathtt{p})}\ \text{IND.HYP.}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma \vdash_1 \mathbf{P}(\Sigma, G, \mathtt{p})}\ [\text{T-LIFT}]$$

□

Theorem E.8 (Theorem D.3)**.** *Let G be realizable. Then $\{x_{\mathtt{pq}} : \mathbf{T}(\emptyset, G, \mathtt{p}, \mathtt{q}, 0) \mid \mathtt{q} \in \mathscr{R} \setminus \{\mathtt{p}\}\} \vdash_1 \mathbf{P}(\emptyset, G, \mathtt{p})$ for every $\mathtt{p} \in \mathscr{R}$.*

Proof: Consequence of Lemma E.7 by taking $\sigma = \emptyset$. □