Yet Another Type System for Lock-Free Processes

Luca Padovani

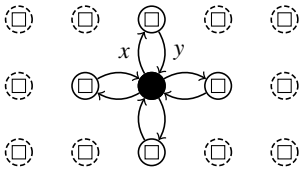
Yet Another Type System for Lock-Free Processes

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

Abstract. A network of processes is *lock free* if every message produced in it is eventually consumed and if every process waiting for a message eventually receives one. We study a type system guaranteeing that well-typed process networks are lock free. Despite its minimality, our type system subsumes existing type-based approaches for lock freedom. In particular, we show that interactions whose lock freedom is guaranteed *by design*, because they are described by a global specification, can be realized as a well-typed network of processes.

1 Introduction

Communication has become a prominent aspect of all modern system architectures, which are characterized by a fragmentation into interacting computational entities. The ever growing complexity of such systems calls for reliable (*i.e.* formal) methods verifying that they enjoy various safety and liveness properties. In this work, we are concerned with *lock freedom* [11], namely the property that every message that is produced is eventually consumed, and every message expected to be consumed is eventually produced. In its simplicity, lock freedom entails a number of other desirable properties including *deadlock freedom* and absence of *orphan messages*.



To make the discussion more concrete and to set up one of the bench tests for our solution, consider the network depicted aside, representing a grid of processes that elaborate some compound data structure (say a matrix M). Such configurations arise frequently in the parallel implementation of iterative algorithms, such as Jacobi finite differences. Each process is dedicated to computing one element $M_{i,j}$ of the matrix. To determine the value of $M_{i,j}$ at the $(n+1)$-th iteration, the process must know the value of its neighbour elements at iteration n. To this aim, the process at (i,j) sends $M_{i,j}(n)$ to its neighbours, receives $M_{i-1,j}(n), M_{i,j-1}(n), M_{i,j+1}(n), M_{i+1,j}(n)$ from them, computes $M_{i,j}(n+1)$ and then iterates. We can model the behavior of a process with respect to one of its neighbours as a term

$$Node_A(x,y) \stackrel{\text{def}}{=} (\nu a)(x!\langle a\rangle \mid y?(z).Node_A\langle a,z\rangle) \tag{1.1}$$

parametric in two channels, x for sending messages to and y for receiving messages from the neighbour. The process creates a new channel a (a *continuation*) which sends in $x!\langle a\rangle$ along with the payload $M_{i,j}(n)$ (omitted). So, a is the channel on which $Node_A$ will send $M_{i,j}(n+1)$ at the next iteration. The process then waits for a message from the y channel, which carries another continuation z along with the neighbour's payload (omitted). At that point, the process starts anew with the two new continuations. It is important to use fresh channels at each iteration to make sure that messages are sent and

received in the desired order (we are assuming that communication is asynchronous and that processes run independently).

A necessary requirement for the correctness of the system is that the composition

$$L_1 \stackrel{\text{def}}{=} Node_A\langle e, f\rangle \mid Node_A\langle f, e\rangle$$

representing a link between a process and one of its neighbours must be lock free. Note that we swap the two distinct channels e and f in the two invocations of $Node_A$, so that no confusion arises between messages exchanged by the two linked processes. Note also that $Node_A$ is not the only possible modeling of a process. An alternative one is

$$Node_B(x, y) \stackrel{\text{def}}{=} y?(z).(\nu a)(x!\langle a\rangle \mid Node_B\langle a, z\rangle) \tag{1.2}$$

in which the process first waits for a message from its neighbour, and only then sends its own information. Therefore, it is also relevant to determine whether the compositions

$$L_2 \stackrel{\text{def}}{=} Node_A\langle e, f\rangle \mid Node_B\langle f, e\rangle \qquad L_3 \stackrel{\text{def}}{=} Node_B\langle e, f\rangle \mid Node_B\langle f, e\rangle$$

are lock free. In L_1 each process produces its own message *before* waiting for the neighbour's; in L_2 one process waits for the neighbour's message before producing its own, but the neighbour does not; in L_3 each process waits for the neighbour's message before producing its own. From these observations we can argue that both L_1 and L_2 are lock free, while L_3 is not. The main contribution of this article is the definition of a type system that turns these informal arguments into verified facts.

Current type-based approaches for guaranteeing lock freedom of communicating processes can be roughly classified as *bottom-up*, *top-down*, and *hybrid*. The basic idea of *bottom-up approaches* [11,13,14] is to enrich channel types with information that specifies the (partial) order in which different channels can used. This information comes in different forms (as *time tags* [11], as *priorities* [13], or as *events* [14]) although the purpose is the same, namely to detect circular dependencies between channels like in L_3, where the input on e blocks the output on f and the input on f blocks the output on e. In general, these approaches have problems dealing with recursive processes, either because they lack (or admit limited forms of) recursive types, or because the information they attach to channel types has an absolute meaning, which easily leads to false circular dependencies due to the fact that the *same* time tag/priority/event occurs several times in the unfolding of a recursive type. For example, none of the L_i's is well typed in any of these type systems: even in L_1, which is the most "asynchronous" of these compositions, the type systems spot a circularity between the inputs on e and f. *Top-down approaches* [9,1,8,7] use *global types* to describe from a vantage point of view the topology of the communication network as a whole, along with the number and role of the interacting processes. For example, the global type satisfying the equation

$$G = \mathtt{A} \to \mathtt{B}.\mathtt{B} \to \mathtt{C}.\mathtt{C} \to \mathtt{A}.G$$

describes an interaction between three processes `A`, `B` and `C` that pass messages around in a ring configuration. If the global type satisfies some realizability conditions, it can be projected onto the local behaviors of the single participants and their parallel composition is guaranteed to be lock free. Top-down approaches enjoy better expressiveness

regarding recursive behaviors; for example, none of the known bottom-up approaches is able to declare that the process implementing the session described by G is well typed. On the down side, global types lack compositionality and they require advance planning of, and impose constraints to, the network topology (consider that it takes the most sophisticated global type language available to date [7] to faithfully describe the interactions occurring in a system as simple as L_1). For these reasons, and despite their "global" denomination, global types are successful for describing confined interactions such as those occurring within multiparty sessions but find limited applicability on larger scales. The same reasons explain the interest towards *hybrid approaches* [1,3,4] where global types are complemented by information on the order in which different sessions interleave. The outcome is barely the sum of the first two approaches, which is disappointing considering that hybrid approaches inherit the disadvantages of the other approaches and require complex, multi-level type systems.

For completeness we also mention other works [2,15] not fitting the above categories where the syntax of (well-typed) processes prevents the modeling of cyclic network topologies. In these cases a plain session type discipline without specific features ensures lock freedom. Note however that all the L_i's and G shown earlier describe cyclic network topologies, and therefore cannot be expressed or typed by the type disciplines described in these articles.

The type system we put forward, albeit falling in the bottom-up classification, encompasses the expressiveness of current bottom-up approaches, fills the gap with and actually goes beyond top-down approaches, and achieves all this with a simple and uniform machinery. To substantiate our claims:

- We work with the type system for the linear π-calculus [12], to which we add a single construct $\mathtt{M}t$ where $\mathtt{M}$ contains (literally) one bit of information and t is a type. Not only the extension is minimal, but we give up any advanced type structure for describing complex interactions, which must be encoded by means of explicit continuation passing [6]. Our type system is impervious to such additional burden.
- We give typing derivations for $Node_A$ and $Node_B$ which, written with explicit continuations as shown in (1.1) and (1.2), are representative of the kind of processes that current bottom-up approaches are *not* able to handle.
- We prove that the encoding of well-typed terms of the simply-typed λ-calculus yields well-typed (*i.e.*, lock-free) processes. This is a proof of weak termination for the simply-typed λ-calculus which also shows the ability of our type system to handle interleaved, terminating communications in dynamic network topologies.
- We prove that every multiparty session described by a realizable global type can be encoded as a well-typed (*i.e.*, lock-free) process that communicates through linear channels. This shows the general ability of our type system to handle interleaved, possibly non-terminating communications in static networks topologies.

Section 2 defines syntax and semantics of the π-calculus and gives the precise definition of lock freedom. In Section 3 we take an informal tour of the type system, to prepare the reader for its formal development in Section 4. Sections 5 and 6 show the encodings of the simply-typed λ-calculus and of multiparty sessions. Section 7 concludes and hints at future developments. *Proofs and additional technical material can be found in the Appendix, which is not part of the formal submission.*

2 Language

We use h, k, …, m, n, … to denote natural numbers; we let x, y, … range over a countable set of *variables* and a, b, … range over a countable set of *channels*; *names* u, v, … are either channels or variables. *Expressions* and *processes* are defined below:

$$\begin{array}{rcl} e & ::= & n \mid u \mid e,e \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \cdots \\ P & ::= & \mathbf{0} \mid u?(x).P \mid *u?(x).P \mid u!\langle e\rangle \mid (P \mid Q) \mid (\nu a)P \mid \begin{array}{l}\texttt{let}\ x,y = e \\ \texttt{in}\ P\end{array} \mid \begin{array}{l}\texttt{case}\ e\ \texttt{of} \\ \{\ \texttt{inl}\ x \Rightarrow P, \\ \quad \texttt{inr}\ y \Rightarrow Q\ \}\end{array} \end{array}$$

Expressions e, … are either natural numbers, names, pairs of expressions, or expressions injected using one of the two constructors `inl` and `inr`. Other operators and data types can be accommodated, assuming that the evaluation of expressions always terminates. *Values* v, w, … are expressions without variables.

Processes P, Q, … are terms of the π-calculus enriched with pattern matching for pairs and injected values. In particular, $\mathbf{0}$ performs no action. An input $u?(x).P$ waits for a message from channel u and then continues as P where the message has been substituted to x. A replicated input $*u?(x).P$ is similar, but is *persistent*: each time a message is received from u, a new copy of P is spawned and $*u?(x).P$ remains available for further inputs. An output $u!\langle e\rangle$ sends the value of e on u. Outputs have no continuation, namely communication is asynchronous. Processes $P \mid Q$ and $(\nu a)P$ respectively denote the parallel composition of P and Q and the restriction of a in P, as usual. A process $\texttt{let}\ x,y = e\ \texttt{in}\ P$ 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 $\texttt{case}\ e\ \texttt{of}\{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}}$ evaluates e, which must result into a value $(i\ \mathsf{v})$ where $i \in \{\texttt{inl},\texttt{inr}\}$, and continues as P_i where x_i has been replaced by v.

Binders are the same as in the π-calculus, with the addition of $\texttt{let}\ x,y = e\ \texttt{in}\ P$, which binds both x and y in P, and $\texttt{case}\ e\ \texttt{of}\{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}}$, which binds x_i in P_i. The notions of *free* and *bound names* of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, follow consequently.

The operational semantics is defined as usual by a combination of *structural congruence* $\equiv$ and a *reduction relation* $\to$. We omit the definition of structural congruence, which is basically 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 an *ad hoc* way to guarantee input receptiveness. Reduction is defined by the rules below

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

$$\text{[R-LET]}\quad \texttt{let}\ x,y = \mathsf{v},\mathsf{w}\ \texttt{in}\ P \to P\{\mathsf{v},\mathsf{w}/x,y\} \qquad \text{[R-CASE]}\quad \frac{k \in \{\texttt{inl},\texttt{inr}\}}{\texttt{case}\ (k\ \mathsf{v})\ \texttt{of}\{i\ x_i \Rightarrow P_i\}_{i=\texttt{inl},\texttt{inr}} \to P_k\{\mathsf{v}/x_k\}}$$

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

To formulate the lock-freedom property, we define a few predicates that describe the immediate input/output capabilities of a process P with respect to some channel a:

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

In words, $\mathsf{input}(a,P)$ 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 $*\mathsf{input}(a,P)$ and $\mathsf{output}(a,P)$ are similar, but they regard replicated inputs and outputs. Whenever one of $\mathsf{input}(a,P)$ or $\mathsf{output}(a,P)$ holds, the lock-freedom property is "in danger", in the sense that there is some pending input/output operation that is required to succeed (eventually). On the contrary, we do not consider $*\mathsf{input}(a,P)$ as endangering lock freedom because we do not require that a persistent input process should unblock infinitely often. This discussion explains the $\mathsf{free}(a,P)$ 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 available. Then, lock freedom can be expressed as the property that every process P such that $\mathsf{wait}(a,P)$ holds can be reduced to a state in which $\mathsf{free}(a,P)$ holds. Formally:

Definition 2.1 (lock-free process). *We say that P is* lock free *if for every Q such that $P \to^* (\nu\tilde{a})Q$ and $\mathsf{wait}(a,Q)$ there exists R such that $Q \to^* R$ and $\mathsf{free}(a,R)$.*

Example 2.1. We encode the process $Node_B$ from Section 1 using replication for process definition, output for process invocation, and pair construction and decomposition for passing multiple parameters. More precisely:

$$Node_B \stackrel{\text{def}}{=} *c_B?(x').\texttt{let}\ x,y = x'\ \texttt{in}\ y?(z).(\nu a)(x!\langle a\rangle \mid c_B!\langle a,z\rangle)$$

where channel c_B can be used for invoking $Node_B$. Using this encoding, the configuration L_3 is represented and reduces thus:

$$\begin{aligned}
L_3 \stackrel{\text{def}}{=}\ & Node_B \mid c_B!\langle e,f\rangle \mid c_B!\langle f,e\rangle \\
\to^*\ & Node_B \mid f?(z).(\nu a)(e!\langle a\rangle \mid c_B!\langle a,z\rangle) \mid e?(z).(\nu b)(f!\langle a\rangle \mid c_B!\langle b,z\rangle) \not\to
\end{aligned}$$

and the final configuration satisfies the input predicate but not output for both e and f. We deduce that L_3 is *not* lock free. A similar configuration is reached also by $Node_B \mid c_B!\langle e,e\rangle$, where the input on e guards the very output that is meant to unblock it. ■

In the following we will use polyadic inputs $u?(x_1,\ldots,x_n)$ as an abbreviation for monadic input of (possibly nested) pairs followed by (possibly nested) pair decostructions using `let`'s, as in Example 2.1. We will also occasionally write $\texttt{let}\ x = e\ \texttt{in}\ P$ in place of $\texttt{let}\ x,y = e,0\ \texttt{in}\ P$ for some fresh y.

3 Types for lock freedom (informal)

We start from a layer of familiar types for channels and messages, those for the linear π-calculus [12,6]. This means classifying channels as either *linear* – that *must* be used exactly once for input and once for output, like e and f in Example 2.1 – or *unlimited* – that *can* be used any number of times, like c_B in Example 2.1. The types $?[t]$ and $![t]$ denote linear channels for respectively receiving/sending one message of type t, while $?^\omega[t]$ and $!^\omega[t]$ denote unlimited channels. We will make sure that a channel of type $?^\omega[t]$ is used for a replicated input, to guarantee *input receptiveness*. That is, using the terminology of [12], we treat channels with type $?^\omega[t]$ as *replicated channels*. A channel $!^\omega[t]$ *can* (but need not) be used for output.

With this basis we are able to exclude some simple locked processes right away, for example those where a linear channel is used just once, either for one input or for one output. But this is clearly not enough: all the processes in Section 1 respect linearity, and yet some of them eventually lock. A paradigmatic example of locked process is

$$a?(x).b!\langle x\rangle \mid b?(y).a!\langle y\rangle \tag{3.1}$$

where the subprocess on the left hand side of $\mid$ forwards one message from channel a to channel b, and the subprocess on the right hand side of $\mid$ forwards one message from channel b to channel a. We have already met a configuration like this in Example 2.1, while discussing the reduction of L_3. The process (3.1) is well typed in a conventional linear type system, for example by typing the left subprocess under the type environment $a : ?[\mathtt{nat}], b : ![\mathtt{nat}]$ and the right subprocess under the type environment $a : ![\mathtt{nat}], b : ?[\mathtt{nat}]$ (note that each channel is given opposite polarities $?$ and $!$ in the two environments). Having assigned independent types to the two channels a and b, there is nothing in these type environments that hints at the circular dependency between the input on a that blocks the output on b and the input on b that blocks the output on a. To keep track of these dependencies, we decorate channel types with a construct $\mathtt{S}^m$ where m is the *rank* of the channel, a number measuring the urgency with which the channel must be used ($\mathtt{S}^m$ denotes m consecutive applications $\mathtt{S}\cdots\mathtt{S}$ of the $\mathtt{S}$ type constructor). We obtain two tentative type environments for the processes in (3.1)

$$a : \mathtt{S}^m\,?[\mathtt{nat}], b : \mathtt{S}^n\,![\mathtt{nat}] \qquad \text{and} \qquad b : \mathtt{S}^n\,?[\mathtt{nat}], a : \mathtt{S}^m\,![\mathtt{nat}]$$

which are no longer independent, because they mention the same ranks m and n. Then, we verify that processes perform input operations on channels in an order that respects their rank: *inputs on channels with lower rank must occur before other operations on channels with higher rank*. With this constraint (3.1) becomes ill typed because it should simultaneously satisfy the contradictory constraints $m < n$ and $n < m$.

The same principle extends to bound variables. For example, the process

$$a?(x).b?(y).y!\langle x\rangle \mid b!\langle a\rangle \tag{3.2}$$

is locked although there is no immediate circular dependency between inputs on a and inputs on b. There is an input on a that blocks an input on b, from which we deduce that the rank of a, say m, should be lower than that of b, say n. There is also an input on b that

blocks an output on y, which is the message received from b itself. Therefore, n should be lower than the rank of y. From $b!\langle a\rangle$ we also learn that the rank of a coincides with that of y. Overall, we end up once again with a conjunction of contradicting constraints $m < n$ and $n < m$. From (3.2) we extrapolate another general rule: *the rank of a message must be greater than that of the channel on which it travels*. Thinking again at ranks as of measures of urgency, this makes sense considering that a message can be used only *after* the channel on which it travels has been used.

By now the attentive reader will be aware of a conflict. On the one hand, the need to work with recursive processes such as (1.1) in Section 1. On the other hand, the escalation of ranks from channels to messages received from them. Consider the process

$$*c?(x).x?(y).c!\langle y\rangle \tag{3.3}$$

which reproduces, in a simplified form, the recursive structure of $Node_A$. Suppose that the type of c indicates that x is a channel with rank m; the process receives a message y from x, from which we deduce that the rank of y is higher than m; the process starts a new iteration sending y on c. The trouble here is that x and y must have different ranks (hence different types) and yet they are used in the same position (x is received from c and y is sent on c). This problem is the main show-stopper for all the bottom-up approaches mentioned in Section 1. It turns out that unlimited channels like c in (3.3) enjoy an interesting (and crucial) form of polymorphism to the extent that *it is safe to send y on c even if the rank of y is lifted compared to the one of x*. Intuitively, this is because it is not the *absolute* value of the rank of y that matters, but only the fact that it is higher than that of x.

There is another problem though, this time concerning the concrete representation of the type of x and y. The recursive structure of (3.3) requires x and y to have a recursive type. In particular, from the above discussion one could come up with the assignment $x : t$ where t is the type that satisfies the equation

$$t = \mathtt{S}^m\,?[\mathtt{S}^n\,?[t]]$$

and $\mathtt{S}^n\,?[t]$ is the type of y. However, no matter how we choose $m < n$ (we want y to have higher rank than x) we have only shifted the problem because at the next iteration the message received from y will have again rank m, and $n < m$ does not hold. We solve this conflict by *giving a relative, rather than absolute, interpretation to ranks written in types*. In particular, we postulate that the actual type of y is $\mathtt{S}^{m+n}\,?[t]$ and not $\mathtt{S}^n\,?[t]$ because y comes from x which has rank m; the channel received from y will have rank $2m+n$, the next one rank $2m+2n$, and so on and so forth. In fact, we will see shortly (Example 4.1) that it suffices to choose $t = ?[\mathtt{S}\,t]$ to find a typing derivation for (3.3).

The idea of using ranks for restricting the order in which channels should be used (for input) is insufficient to enforce lock freedom. For example, the process

$$c!\langle a\rangle \mid *c?(x).c!\langle x\rangle \mid a!\langle 1984\rangle \to c!\langle a\rangle \mid *c?(x).c!\langle x\rangle \mid a!\langle 1984\rangle \to \cdots \tag{3.4}$$

is not lock free despite the fact that it reduces forever. The problem is that there is a pending output on channel a, but no corresponding input action because the "other half" of channel a is just passed around as message in the infinite recursion that involves the

unlimited channel c. Observe that the three sub-processes in (3.4) can be respectively typed under the three type environments (ranks omitted)

$$a : \mathtt{?}[\mathtt{nat}], c : \mathtt{!}^{\omega}[\mathtt{?}[\mathtt{nat}]] \qquad c : \mathtt{?}^{\omega}[\mathtt{?}[\mathtt{nat}]] \qquad a : \mathtt{!}[\mathtt{nat}]$$

and that, in particular, the linear channel a occurs indeed linearly, once with input polarity in the first sub-process and once with output polarity in the third sub-process. This example is an instance of a degenerate phenomenon whereby some channel – a in (3.4) – is the *object* of infinite communications occurring on other channels – c in (3.4) – but it is never the *subject* of such communications. We can now reveal that the $\mathtt{S}$ in the examples above stands for "subject rank" of a channel, and that we introduce another constructor $\mathtt{O}$ standing for "object rank" of a channel. The object rank of a channel gives an upper bound to the number of times it can be sent in a message. Every time this happens, the object rank of the channel decreases. Because all channels originate with finite ranks, every channel must eventually be used as the subject of some communication. For example, we will see that the channel y in $Node_A$ can be given the type $t = \mathtt{?}[\mathtt{S}\,\mathtt{O}\,t]$ because the channel z received from it is used only once as an object – in the invocation $Node_A\langle a, z\rangle$ – before it is used as subject.

4 Type system

Syntax. We introduce some more notation. *Polarities* $p, q, \ldots$ are non-empty subsets of $\{\mathtt{?}, \mathtt{!}\}$. For readability we will often abbreviate $\{\mathtt{?}\}$ with $\mathtt{?}$, $\{\mathtt{!}\}$ with $\mathtt{!}$, and $\{\mathtt{?}, \mathtt{!}\}$ with $\mathtt{\#}$. *Multiplicities* $\iota, \ldots$ are either 1 or ω. *Rank modifiers* $\mathtt{M}, \ldots$ are either $\mathtt{S}$ or $\mathtt{O}$. We also need a countable set of *type variables* $\alpha, \ldots$. *Types* $t, s, \ldots$ are defined by the grammar below:

$$t \quad ::= \quad \mathtt{nat} \;\mid\; \alpha \;\mid\; t \times s \;\mid\; t \oplus s \;\mid\; p^{\iota}[t] \;\mid\; \mathtt{M}\,t \;\mid\; \mu\alpha_k\{\alpha_i := t_i\}_{i=1..n}$$

The types $\mathtt{nat}$, $t \times s$, and $t \oplus s$ respectively denote natural numbers, pairs inhabited by values $(\mathtt{v}, \mathtt{w})$ where $\mathtt{v}$ has type t and $\mathtt{w}$ has type s, and the disjoint sum of t and s inhabited by values $(\mathtt{inl}\ \mathtt{v})$ when $\mathtt{v}$ has type t or $(\mathtt{inr}\ \mathtt{w})$ when $\mathtt{w}$ has type s. The type $p^{\iota}[t]$ 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: $\mathtt{?}$ 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. We will usually write $p[t]$ in place of $p^{1}[t]$. We use type variables and μ's for building recursive types. In particular, 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 than the one that binds exactly one variable [5] and in the following we will often write $\mu\alpha.t$ in place of $\mu\alpha\{\alpha := t\}$. We adopt this generalized operator because it simplifies many technicalities in Section 6. Notions of free and bound type variables are as expected. Modifiers lift the subject and object ranks of a type. In particular, $\mathtt{S}\,t$ adds 1 to the subject rank of t and $\mathtt{O}\,t$ adds 1 to the object rank of t. We will see how to compute subject and object rank of a type shortly. We write $\mathtt{M}^n\,t$ for n (possibly 0) consecutive applications of the modifier $\mathtt{M}$ to t.

Contractiveness. We forbid non-contractive types where recursion variables are not guarded by a channel type. In particular, $\mu\alpha.\alpha$, $\mu\alpha.\mathtt{M}\alpha$, $\mu\alpha.(\alpha\times\alpha)$, and $\mu\alpha.(\alpha\oplus\alpha)$ are illegal while $\mu\alpha.p^{\mathsf{l}}[\alpha]$ is allowed. The formal definition of contractiveness can be found in Appendix A. 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 [5]. 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.

Ranks. We now define the $\mathtt{M}$-rank of a type, where ranks are elements of $\mathbb{N}\cup\{\bot,\top\}$ ordered in the obvious way ($\bot < n < \top$ for every $n\in\mathbb{N}$). We extend the $+$ operation on natural numbers to ranks so that $\bot+n=\bot$ and $\top+n=\top$; we leave $\bot+\top$ undefined. The *$\mathtt{M}$-rank* of a closed type t, denoted $\mathtt{M}\text{-}\mathsf{rank}(t)$, is defined as:

$$\mathtt{M}\text{-}\mathsf{rank}(t) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } t = (\mathsf{?}\cup p)^{\omega}[s] \\ 0 & \text{if } t = p^{\mathsf{l}}[s] \\ 1+\mathtt{M}\text{-}\mathsf{rank}(s) & \text{if } t = \mathtt{M}s \\ \mathtt{M}\text{-}\mathsf{rank}(s) & \text{if } t = \mathtt{M}_0\,s \text{ and } \mathtt{M}\neq\mathtt{M}_0 \\ \min\{\mathtt{M}\text{-}\mathsf{rank}(t_1),\mathtt{M}\text{-}\mathsf{rank}(t_2)\} & \text{if } t = t_1\times t_2 \text{ or } t = t_1\oplus t_2 \\ \top & \text{otherwise} \end{cases} \tag{4.1}$$

Contractiveness ensures that $\mathtt{M}\text{-}\mathsf{rank}(t)$ is well defined. Intuitively, every type that has a top-level unlimited channel type with input polarity has $\mathtt{M}$-rank $\bot$ (first equation). In the other cases, $\mathtt{M}\text{-}\mathsf{rank}(t)$ is the smallest number of $\mathtt{M}$'s occurring along any path from the root of t to any of its topmost channel types (second to fifth equations), or $\top$ if all the top-level channel types are unlimited and only have output polarity (last equation). The rationale of this definition is clear when thinking of the $\mathtt{S}$-rank as an inverse priority: an unlimited channel with input polarity must be used with maximum priority ($\bot$) because we want to ensure input receptiveness, so its use cannot be postponed by any means; numbers and unlimited channels with only output polarity can be used with the least priority ($\top$) because their use does not affect lock freedom in any way; linear channels must be used with the relative priority determined by the number of $\mathtt{S}$ constructors. Note the difference between the ranks $\bot$ and 0: the former one cannot be lifted because $\bot$ absorbs any natural number ($1+\bot=\bot$), while the latter one can be lifted ($1+0=1$).

We say that a (name with) type t is *unlimited* if $\mathtt{S}\text{-}\mathsf{rank}(t)=\top$; we say that it is *linear* if $\mathtt{S}\text{-}\mathsf{rank}(t)\in\mathbb{N}$; we say that it is *relevant* if $\mathtt{S}\text{-}\mathsf{rank}(t)=\bot$.

Type equivalence. It is handy to move modifiers around without altering the rank of a type. To this aim we define an *equivalence relation* $\simeq$ as the least congruence such that:

$$\mathtt{M}_1\mathtt{M}_2 t \simeq \mathtt{M}_2\mathtt{M}_1 t \qquad \mathtt{M}(t\times s)\simeq \mathtt{M}t\times\mathtt{M}s \qquad \mathtt{M}(t\oplus s)\simeq \mathtt{M}t\oplus\mathtt{M}s \qquad \frac{\mathtt{M}\text{-}\mathsf{rank}(t)\in\{\bot,\top\}}{t\simeq\mathtt{M}t}$$

With $\simeq$ we can permute modifiers and distribute them over products and sum types. We can also materialize and erase modifiers *ad libitum* in front of unlimited and relevant types. This is useful for rewriting products $t\times s$ and sums $t\oplus s$ where only one of t and

s is linear. For instance, $\mathtt{nat} \times (\mathtt{M}\,p[t]) \simeq (\mathtt{M}\mathtt{nat}) \times (\mathtt{M}\,p[t]) \simeq \mathtt{M}\,(\mathtt{nat} \times p[t])$. In general we will silently use $\simeq$ everywhere a type occurs for rewriting it in a normal form:

Proposition 4.1 (type normal form). *For every t there exist m, n, and s such that $t \simeq \mathtt{S}^m\,\mathtt{0}^n\,s$ and s is a constructor other than a modifier or a μ.*

Type environments. We check that processes are well typed in *type environments*, which are finite maps from names to types written $u_1 : t_1, \ldots, u_n : t_n$. We let $\Gamma, \ldots$ range over type environments, we write $\mathsf{dom}(\Gamma)$ for the *domain* of Γ, namely the set of names for which there is a binding in Γ, and we write 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 [12] we define a partial composition operator $+$ between types, thus:

$$\begin{array}{rll} t + t = t & & \text{if } t \text{ is unlimited} \\ p^\omega[t] + q^\omega[t] = (p \cup q)^\omega[t] & & \\ \mathtt{S}^n\,\mathtt{0}^h\,p[s] + \mathtt{S}^n\,\mathtt{0}^k\,q[s] = \mathtt{S}^n\,\mathtt{0}^{h+k}\,(p \cup q)[s] & & \text{if } p \cap q = \emptyset \end{array} \tag{4.2}$$

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 $\mathtt{S}$-rank and disjoint polarities, and the composition has the union of their polarities. Note that the $\mathtt{0}$-ranks of the two types are added together in the composition, meaning that there is some loss of information as to which component had which $\mathtt{0}$-rank. This is however irrelevant for the soundness of the type system.

We extend $+$ to a partial composition operator between type environments:

$$\Gamma_1 + \Gamma_2 \stackrel{\text{def}}{=} \begin{cases} \Gamma_1, \Gamma_2 & \text{if } \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset \\ (\Gamma_1' + \Gamma_2'), u : t + s & \text{if } \Gamma_1 = \Gamma_1', u : t \text{ and } \Gamma_2 = \Gamma_2', u : s \end{cases} \tag{4.3}$$

Note that $\Gamma_1 + \Gamma_2$ is undefined if there is $u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ such that $\Gamma_1(u) + \Gamma_2(u)$ is undefined and that $\mathsf{dom}(\Gamma_1 + \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Here is some more notation regarding type environments. We write $\mathsf{un}(\Gamma)$ if all the types in the range of Γ are unlimited. We let $\mathtt{S}\text{-}\mathsf{rank}(\Gamma)$ denote the least $\mathtt{S}$-rank of the types in the range of Γ, that is $\mathtt{S}\text{-}\mathsf{rank}(\Gamma) = \min\{\mathtt{S}\text{-}\mathsf{rank}(\Gamma(u)) \mid u \in \mathsf{dom}(\Gamma)\}$. We let $\mathtt{S}\,\Gamma$ denote the environment that is the same as Γ, except that all types in its range have been lifted by the $\mathtt{S}$ constructor, that is $(\mathtt{S}\,\Gamma)(u) = \mathtt{S}\,\Gamma(u)$ for every $u \in \mathsf{dom}(\Gamma)$.

Type rules. The type rules for expressions and processes are presented in Table 1. The former ones derive judgments of the form $\Gamma \vdash e : t$, denoting that e is well typed and has type t in Γ; the latter ones derive judgments of the form $\Gamma \vdash P$, denoting that P is well typed in Γ. The type rules for expressions are unremarkable. Just observe that [T-CONST] and [T-NAME] require the unused part of the type environment to be unlimited. In other words, linear and relevant names *must* be used. Also note that [T-PAIR] splits the type environment according to (4.3).

Rule [T-IDLE] states that the idle process is well typed in any unlimited environment.

Table 1. Type rules for expressions and processes.

Expressions

$$[\text{T-CONST}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash n : \mathtt{nat}} \qquad [\text{T-NAME}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma, u : t \vdash u : t} \qquad [\text{T-PAIR}]\ \frac{\Gamma \vdash e : t \qquad \Gamma' \vdash e' : s}{\Gamma + \Gamma' \vdash e, e' : t \times s}$$

$$[\text{T-INL}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{inl}\ e : t \oplus s} \qquad [\text{T-INR}]\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inr}\ e : t \oplus s}$$

Processes

$$[\text{T-IDLE}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad [\text{T-IN}]\ \frac{\Gamma, x : \mathtt{S}^n\, t \vdash P \qquad n < \mathtt{S}\text{-rank}(\Gamma)}{\Gamma + u : \mathtt{S}^n\, \mathtt{0}^m\, ?[t] \vdash u?(x).P}$$

$$[\text{T-IN}^*]\ \frac{\Gamma, x : t \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma + u : ?^\omega[t] \vdash *u?(x).P} \qquad [\text{T-PAR}]\ \frac{\Gamma \vdash P \qquad \Gamma' \vdash Q}{\Gamma + \Gamma' \vdash P \mid Q}$$

$$[\text{T-OUT}]\ \frac{\Gamma \vdash e : \mathtt{S}^n\, \mathtt{0}\, t \qquad 0 < \mathtt{S}\text{-rank}(t)}{\Gamma + u : \mathtt{S}^n\, \mathtt{0}^m\, !^\iota[t] \vdash u!\langle e \rangle} \qquad [\text{T-OUT}^*]\ \frac{\Gamma \vdash e : \mathtt{S}^n\, \mathtt{0}\, t \qquad 0 \leq \mathtt{S}\text{-rank}(t)}{\Gamma + u : !^\omega[t] \vdash u!\langle e \rangle}$$

$$[\text{T-NEW}]\ \frac{\Gamma, a : \mathtt{S}^n\, \mathtt{0}^m\, \#^\iota[t] \vdash P}{\Gamma \vdash (\nu a) P}$$

$$[\text{T-LET}]\ \frac{\Gamma \vdash e : t \times s \qquad \Gamma', x : t, y : s \vdash P}{\Gamma + \Gamma' \vdash \mathtt{let}\ x, y = e\ \mathtt{in}\ P} \qquad [\text{T-CASE}]\ \frac{\Gamma \vdash e : t \oplus s \qquad \Gamma', x : t \vdash P \qquad \Gamma', y : s \vdash Q}{\Gamma + \Gamma' \vdash \mathtt{case}\ e\ \mathtt{of}\{\mathtt{inl}\ x \Rightarrow P, \mathtt{inr}\ y \Rightarrow Q\}}$$

Rule [T-IN] is used for typing (linear) input processes $u?(x).P$, where u must be a linear channel with input polarity. 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 lifted by the subject rank of u. The object rank of u is irrelevant, because u is used for an input operation, not as the content of a message. The condition $n < \mathtt{S}\text{-rank}(\Gamma)$ ensures that the input on u does not block operations on other channels with lower or equal rank (see the discussion of (3.1)). In particular, Γ cannot contain u (whose rank would be n) nor any relevant channel (whose rank would be $\bot$).

Rule [T-OUT] is used for typing output processes. Note that the type of the message e must be t (as specified in the type of the channel u) with subject rank lifted by n (which is the rank of u) and object rank lifted by 1. The lifting of the subject rank derives from the fact that the subject rank of t is relative to the subject rank of the channel. The lifting of the object rank actually means that, by sending e as a message, one unit from every finite object rank in the type of e is consumed. This prevents the degenerate phenomenon of infinite delegation that we have modeled in (3.4). The condition $0 < \mathtt{S}\text{-rank}(t)$ makes sure that the subject rank of e is greater than the subject rank of u (we have seen why this is important discussing (3.2)).

Rule [T-IN*] is used for typing replicated input processes $*u?(x).P$. This rules differs from [T-IN] in several important ways. First of all, u must be an unlimited channel with input polarity. Subject and object ranks do not matter, they are both $\bot$ anyway according to (4.1). Second, the residual environment Γ must be unlimited, because there is no guarantee on the number of messages that will be sent on u, and hence on the number of times the continuation P will be spawned (see [R-COMM*]). Third, it may be the case that $u \in \mathsf{dom}(\Gamma)$, because $?^\omega[t] + !^\omega[t] = \#^\omega[t]$ according to (4.2) and $!^\omega[t]$ is unlimited.

This means that replicated input processes may invoke themselves. We have used this feature extensively in all the examples seen so far.

Rule [T-OUT*] is used for outputs on unlimted channels. There are two crucial differences with respect to [T-OUT]. First, the condition $0 \leq \mathsf{S}\text{-rank}(t)$, where t is the type of x, only implies that no relevant names can be communicated. Second, the type of the message need not match exactly the type t in the channel, but its subject rank can be lifted by an arbitrary amount n. This can be tolerated because we know, from [T-IN*], that the receiver process has no free linear channels.

Rules [T-PAR], [T-LET], and [T-CASE] are standard, and [T-NEW] is used for typing the creation of new (linear and unlimted) channels. In both cases the subject and object ranks of the channel can be lifted by arbitrary finite amounts. Note that every new channel comes with the full set $\#$ of polarities.

Properties. Much of the expressiveness of the type system comes from the relative interpretation of ranks, as we have seen in [T-IN] and [T-OUT], and by the polymorphism on subject ranks enabled by [T-OUT*]. The following Lemma, which plays a key role in the soundness proof (Theorem 4.1) and in all the encodings (Sections 5 and 6), formalizes the fact that subject ranks are relative by stating that the uniform lifting of the subject ranks in the type environment preserves typing.

Lemma 4.1 (lifting). *If $\Gamma \vdash P$, then $\mathsf{S}\,\Gamma \vdash P$.*

Note that the converse of the Lemma does not hold. For example, the derivation for $u : \mathsf{S}\,!\,[\mathtt{nat}] \vdash (\nu a)(a!\langle 1984\rangle \mid a?(x).u!\langle x\rangle)$ relies on the fact that u has S-rank 1, for otherwise it would not be possible to prefix $u!\langle x\rangle$ with an input operation on a. It is however possible to lift this derivation by giving u any strictly positive rank.

The type system enjoys subject reduction and is sound (details in Appendix A):

Theorem 4.1 (soundness). *If $\emptyset \vdash P$, then P is lock free.*

Example 4.1. We are now ready to show that both $Node_A$ and $Node_B$ are well typed, and so are (the encodings of) the compositions L_1 and L_2 of Section 1. In fact we are going to do a little more, by guessing the *most general type* (with respect to subject ranks) of the channels c_A and c_B that represent $Node_A$ and $Node_B$. We take

$$c_A, c_B : \#^{\omega}[t \times s] \quad \text{where} \quad t = \mathsf{S}^n\,!\,[\mathsf{S}^m\,\mathsf{0}\,s] \quad \text{and} \quad s = \mu\alpha.\mathsf{S}^h\,?\,[\mathsf{S}^m\,\mathsf{0}\,\alpha] = \mathsf{S}^h\,?\,[\mathsf{S}^m\,\mathsf{0}\,s]$$

where the fact that the same type $\mathsf{S}^m\,\mathsf{0}\,s$ occurs within both t and s is an inevitable consequence of the structure of $Node_A$ (and $Node_B$), so the only variability is given by n and h. For $Node_A$ we obtain the derivation below if and only if $0 < m$:

$$\dfrac{\dfrac{\dfrac{\dfrac{a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s \vdash a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s}{x : t, a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s \vdash x!\langle a\rangle} \quad \dfrac{\dfrac{a : \mathsf{S}^{h+m}\,\mathsf{0}\,t, z : \mathsf{S}^{h+m}\,\mathsf{0}\,s \vdash a, z : \mathsf{S}^{h+m}\,\mathsf{0}\,(t \times s)}{c_A : !^{\omega}[t \times s], a : \mathsf{S}^{h+m}\,\mathsf{0}\,t, z : \mathsf{S}^{h+m}\,\mathsf{0}\,s \vdash c_A!\langle a, z\rangle} \quad 0 < m}{c_A : !^{\omega}[t \times s], y : s, a : \mathsf{S}^{h+m}\,\mathsf{0}\,t \vdash y?(z).c_A!\langle a, z\rangle}}{c_A : !^{\omega}[t \times s], x : t, y : s, a : \mathsf{S}^{h+m+n}\,\mathsf{0}^3\,\#[\mathsf{S}^m\,\mathsf{0}\,s] \vdash x!\langle a\rangle \mid y?(z).c_A!\langle a, z\rangle}}{c_A : !^{\omega}[t \times s], x : t, y : s \vdash (\nu a)(x!\langle a\rangle \mid y?(z).c_A!\langle a, z\rangle)}}{c_A : \#^{\omega}[t \times s] \vdash *c_A?(x, y).(\nu a)(x!\langle a\rangle \mid y?(z).c_A!\langle a, z\rangle)}$$

For $Node_B$ we obtain the derivation below if and only if $h < n$ and $0 < m$:

$$\dfrac{\dfrac{\dfrac{a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s \vdash a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s}{x : t, a : \mathsf{S}^{m+n}\,\mathsf{0}^2\,s \vdash x!\langle a\rangle} \quad \dfrac{z : \mathsf{S}^{h+m}\,\mathsf{0}\,s, a : \mathsf{S}^{h+m}\,\mathsf{0}\,t \vdash a, z : \mathsf{S}^{h+m}\,\mathsf{0}\,(t \times s)}{c_B : !^{\omega}[t \times s], z : \mathsf{S}^{h+m}\,\mathsf{0}\,s, a : \mathsf{S}^{h+m}\,\mathsf{0}\,t \vdash c_B!\langle a, z\rangle}}{c_B : !^{\omega}[t \times s], x : t, z : \mathsf{S}^{h+m}\,\mathsf{0}\,s, a : \mathsf{S}^{h+m+n}\,\mathsf{0}^3\,\#[\mathsf{S}^m\,\mathsf{0}\,s] \vdash x!\langle a\rangle \mid c_B!\langle a, z\rangle}}{\dfrac{c_B : !^{\omega}[t \times s], x : t, z : \mathsf{S}^{h+m}\,\mathsf{0}\,s \vdash (\nu a)(x!\langle a\rangle \mid c_B!\langle a, z\rangle) \qquad h < n \quad 0 < m}{\dfrac{c_B : !^{\omega}[t \times s], x : t, y : s \vdash y?(z).(\nu a)(x!\langle a\rangle \mid c_B!\langle a, z\rangle)}{c_B : \#^{\omega}[t \times s] \vdash *c_B?(x, y).y?(z).(\nu a)(x!\langle a\rangle \mid c_B!\langle a, z\rangle)}}}$$

Since we compose two nodes invoking $Node_A$ and $Node_B$ with swapped pairs of channels, we deduce that $Node_A\langle e, f\rangle \mid Node_A\langle f, e\rangle$ and $Node_A\langle e, f\rangle \mid Node_B\langle f, e\rangle$ and $Node_A\langle e, e\rangle$ are all well typed. Since the constraints found above are the most general ones (concerning subject ranks), we also deduce that there are no derivations for $Node_B\langle e, f\rangle \mid Node_B\langle f, e\rangle$ and $Node_B\langle e, e\rangle$, suggesting that these are not lock free. ■

5 Encoding the simply-typed λ-calculus

In this section we show that the standard encoding of the simply-typed λ-calculus into the π-calculus using the parallel call-by-value evaluation strategy produces well-typed processes according to our type discipline. Terms $M, N, \ldots$ of the λ-calculus and their encoding are defined below:

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

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 $\mathtt{nat} \to \mathtt{nat}$ becomes a channel of type $!^{\omega}[\mathtt{nat} \times ![\mathtt{nat}]]$, where $![\mathtt{nat}]$ 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

$$\tau ::= \mathtt{nat} \mid \tau \xrightarrow{m,n} \sigma$$

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 $x!\langle y,u\rangle$.

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

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

Types are encoded taking into account the effects recorded over arrows, and the encoding of terms preserves typing:

$$[\![\mathtt{nat}]\!] \stackrel{\text{def}}{=} \mathtt{nat} \qquad [\![\tau \xrightarrow{m,n} \sigma]\!] \stackrel{\text{def}}{=} !^{\omega}[[\![\tau]\!] \times \mathtt{S}^m\mathtt{0}^n\,![[\![\sigma]\!]]]$$

Theorem 5.1. *$\emptyset \vdash M : \tau \;\&\; m,n$ implies $a : \mathtt{S}^m\mathtt{0}^n\,![[\![\tau]\!]] \vdash [\![M]\!]^a$.*

6 Encoding multiparty sessions

In this section we show how to encode a multiparty session using solely linear channels. The challenge is to obtain a well-typed (and consequently lock-free) encoding, considering that each participant of the session will be modeled by a recursive process that interleaves possibly blocking actions pertaining several different linear channels. We assume a finite, totally ordered set $\mathscr{R}$ of *participant tags* $\mathtt{p}, \mathtt{q}, \ldots, \mathtt{A}, \mathtt{B}, \ldots$ and a set of *labels* $\ell, \ldots$ for identifying events in multiparty sessions. *Events* $\varepsilon, \ldots$ and *global types* $G, \ldots$ are defined below:

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

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. The global type $G_1 \parallel G_2$ describes a protocol where G_1 and G_2 take place in some unspecified order. Note that the same participant may occur in both G_1 and 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 define `end` as $\mu\alpha.\alpha$ for denoting the terminated protocol in which no synchronization occurs. We write $\mathsf{fv}(G)$ for the set of free type variables in G, which is defined as expected.

Not every global type describes a sensible prototol. 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 `C` sends a message to `D` or `D` sends a message to `C` depending on the decision taken by `A`. The problem is that neither `C` nor `D` have been explicitly informed about such decision, so there is no way they 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* $\mathtt{p} \wr \mathtt{q}$ is a set $\{\mathtt{p}, \mathtt{q}\}$ with $\mathtt{p} \neq \mathtt{q}$ representing a communication channel between two participants. We let $\mathsf{edge}(\mathtt{p} \to \mathtt{q}@\ell) = \mathtt{p} \wr \mathtt{q}$. 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 6.1. *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 with respect 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* $\downarrow(\Sigma, G, \mathtt{p} \wr \mathtt{q})$ of a global type G with respect to a sort environment Σ and an edge $\mathtt{p} \wr \mathtt{q}$. Intuitively, $\downarrow(\Sigma, G, \mathtt{p} \wr \mathtt{q})$ extracts from G the sub-protocol that concerns only participants $\mathtt{p}$ and $\mathtt{q}$. Formally:

$$\begin{array}{rcll}
\downarrow(\Sigma, \alpha, \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \alpha & \\
\downarrow(\Sigma, \varepsilon.G, \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \varepsilon.\downarrow(\Sigma, G, \mathtt{p} \wr \mathtt{q}) & \text{if } \mathtt{p} \wr \mathtt{q} = \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.G, \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \downarrow(\Sigma, G, \mathtt{p} \wr \mathtt{q}) & \text{if } \mathtt{p} \wr \mathtt{q} \neq \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.[G_1 + G_2], \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \varepsilon.[\downarrow(\Sigma, G_1, \mathtt{p} \wr \mathtt{q}) + \downarrow(\Sigma, G_2, \mathtt{p} \wr \mathtt{q})] & \text{if } \mathtt{p} \wr \mathtt{q} = \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, \varepsilon.[G_1 + G_2], \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \downarrow(\Sigma, G_i, \mathtt{p} \wr \mathtt{q}) & \text{if } \mathtt{p} \wr \mathtt{q} \neq \mathsf{edge}(\varepsilon) \\
\downarrow(\Sigma, G_1 \parallel G_2, \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \downarrow(\Sigma, G_i, \mathtt{p} \wr \mathtt{q}) & \text{if } \mathtt{p} \wr \mathtt{q} \notin \Sigma(G_{3-i}) \\
\downarrow(\Sigma, \mu\alpha.G, \mathtt{p} \wr \mathtt{q}) & \stackrel{\text{def}}{=} & \mu\alpha.\downarrow(\mathsf{upd}(\Sigma, \alpha, G), G, \mathtt{p} \wr \mathtt{q}) &
\end{array}$$

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 $\mathtt{p} \wr \mathtt{q}$ differ (fifth equation), or undefined if the same edge occurs in two independent sub-protocols (sixth equation), in which case the two sub-protocols are not actually independent. For example, $\downarrow(\emptyset, G_a, \mathtt{C} \wr$

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,

$$G_b \stackrel{\text{def}}{=} \mathtt{A} \to \mathtt{B}@\ell_1.[\mathtt{B} \to \mathtt{A}@\ell_2.\mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{end} + \mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{B} \to \mathtt{A}@\ell_4.\mathtt{end}]$$

is projectable with respect to $\mathtt{C} \wr \mathtt{D}$ 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 C and D are unaware of the choice taken by A. Conversely, $\mathtt{B} \to \mathtt{A}@\ell_2$ and $\mathtt{B} \to \mathtt{A}@\ell_4$ may have different labels because both A and B are aware of the choice taken by 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

$$G_c \stackrel{\text{def}}{=} \mathtt{A} \to \mathtt{B}@\ell_1.[\mathtt{E} \to \mathtt{F}@\ell_2.\mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{end} + \mathtt{C} \to \mathtt{D}@\ell_3.\mathtt{E} \to \mathtt{F}@\ell_2.\mathtt{end}]$$

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 $\downarrow(\emptyset, G_c, \mathtt{C} \wr \mathtt{D})$ and $\downarrow(\emptyset, G_c, \mathtt{E} \wr \mathtt{D})$ to be well defined, and yet G_c specifies that the order of these events depends on the decision taken by A, which none of C, D, E, and 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 6.2 (realizable global type). *We say that G is* realizable *if* **(1)** $\downarrow(\emptyset, G, \mathtt{p} \wr \mathtt{q})$ *is well defined for every* $\mathtt{p} \wr \mathtt{q}$ *and* **(2)** *the relation* $\prec_G^+$ *is irreflexive.*

According to Definition 6.2, 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 if we considered, for example, $\mu\alpha.\mathtt{A} \to \mathtt{B}@\ell.\alpha$ equivalent to its own unfolding.

We now show how to build a set of processes that implement any realizable global type. Every participant p in a global type G will be implemented by 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 p to q. $\mathbf{P}(\Sigma, G, \mathtt{p})$ is defined inductively on G by the equations in Table 2. Despite the daunting look of the definition, the idea is simple: every event $\mathtt{A} \to \mathtt{B}$ to which p participates determines either an input or an output action on the channel $x_{\mathtt{pq}}$. When 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 p and q continues. When p is the sender, it sends to q the continuation for the subsequent communications on the edge $\mathtt{p} \wr \mathtt{q}$. 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 $\mathtt{p} \wr \mathtt{q}$ occurs in $\Sigma(G)$, that is if p and q will communicate again in the future, or the number 0 if no other interaction between p and 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

Table 2. Characteristic participant.

$$
\begin{array}{rcll}
\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{of}\{\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,G_1\parallel G_2,\mathtt{p}) & \stackrel{\text{def}}{=} & \mathbf{P}(\Sigma,G_1,\mathtt{p})\mid\mathbf{P}(\Sigma,G_2,\mathtt{p}) & \\
\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}
$$

in the sender and `case` to decode the decision in the receiver and we use the same auxiliary function $\mathbf{C}$ for sending the decision, to which we pass the injector `inl` to wrap the continuation. Forked global types are encoded as parallel processes: the fact that they are realizable guarantees that the same variable $x_{\mathtt{pq}}$ is used in at most one of the two processes, therefore respecting linearity. Finally, recursions $\mu\alpha.G$ are implemented following the same scheme that we have seen over and over in the previous sections: we associate 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 illustration, 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 $\mathscr{R}=\{\mathtt{A},\mathtt{B},\mathtt{C}\}$ we have

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

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 since this information is written in the global type. But in order for $\mathbf{P}(\emptyset,G,\mathtt{p})$ to be well typed, we must also make sure that blocking actions respect the subject rank of channels (see [T-IN]) and that delegations are allowed by the object rank of channels (see [T-OUT] and [T-OUT*]). We determine subject ranks using the event order $\prec_G$ and object 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 so the idea is that we can use just $\mathsf{ord}(\ell)$ as the (relative) subject rank for the channel types when projecting the action labeled by ℓ. In what follows, we simplify the notation writing just ℓ in place of $\mathsf{ord}(\ell)$.

Table 3 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 $\mathbf{P}(\Sigma,G,\mathtt{p})$

Table 3. Global type projection.

$$\mathbf{T}(\Sigma,\alpha,\mathtt{p},\mathtt{q},n) \stackrel{\text{def}}{=} \begin{cases} \mathtt{0}\,\alpha_{\mathtt{pq}} & \mathtt{p}\wr\mathtt{q}\in\Sigma(\alpha) \\ \mathtt{nat} & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\Sigma,\mathtt{A}\to\mathtt{B}@\ell.G,\mathtt{p},\mathtt{q},n) \stackrel{\text{def}}{=} \begin{cases} \mathtt{S}^{\ell-n}\,!\,[\mathbf{T}(\Sigma,G,\mathtt{q},\mathtt{p},\ell)] & \mathtt{p},\mathtt{q}=\mathtt{A},\mathtt{B} \\ \mathtt{S}^{\ell-n}\,?\,[\mathbf{T}(\Sigma,G,\mathtt{p},\mathtt{q},\ell)] & \mathtt{p},\mathtt{q}=\mathtt{B},\mathtt{A} \\ \mathtt{S}^{\ell-n}\,\mathbf{T}(\Sigma,G,\mathtt{p},\mathtt{q},\ell) & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\Sigma,\mathtt{A}\to\mathtt{B}@\ell.[G_1+G_2],\mathtt{p},\mathtt{q},n) \stackrel{\text{def}}{=} \begin{cases} \mathtt{S}^{\ell-n}\,!\,[\mathbf{T}(\Sigma,G_1,\mathtt{q},\mathtt{p},\ell)\oplus\mathbf{T}(\Sigma,G_2,\mathtt{q},\mathtt{p},\ell)] & \mathtt{p},\mathtt{q}=\mathtt{A},\mathtt{B} \\ \mathtt{S}^{\ell-n}\,?\,[\mathbf{T}(\Sigma,G_1,\mathtt{p},\mathtt{q},\ell)\oplus\mathbf{T}(\Sigma,G_2,\mathtt{p},\mathtt{q},\ell)] & \mathtt{p},\mathtt{q}=\mathtt{B},\mathtt{A} \\ \mathtt{S}^{\ell-n}\,\mathbf{T}(\Sigma,G_i,\mathtt{p},\mathtt{q},\ell) & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\Sigma,G_1\parallel G_2,\mathtt{p},\mathtt{q},n) \stackrel{\text{def}}{=} \mathbf{T}(\Sigma,G_i,\mathtt{p},\mathtt{q},n) \qquad \mathtt{p}\wr\mathtt{q}\notin\Sigma(G_{3-i})$$

$$\mathbf{T}(\Sigma,\mu\alpha.G,\mathtt{p},\mathtt{q},n) \stackrel{\text{def}}{=} \mathtt{0}\,\mu\alpha_{\mathtt{pq}}\left\{\begin{array}{l}\alpha_{\mathtt{pq}}=\mathbf{T}(\mathtt{upd}(\Sigma,\alpha,G),G,\mathtt{p},\mathtt{q},n)\\ \alpha_{\mathtt{qp}}=\mathbf{T}(\mathtt{upd}(\Sigma,\alpha,G),G,\mathtt{q},\mathtt{p},n)\end{array}\right\}$$

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 $\mathtt{nat}$ to denote a dummy natural number. Subject ranks 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 $\mathbf{T}(\Sigma,\mathtt{A}\to\mathtt{B}@\ell.G,\mathtt{A},\mathtt{B},n) = \mathtt{S}^{\ell-n}\,!\,[\mathbf{T}(\Sigma,G,\mathtt{B},\mathtt{A},\ell)]$ where $\mathtt{A}$ and $\mathtt{B}$ are *swapped* within $!\,[\cdot]$. 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. The $\mathtt{0}$ constructor is used in the two strategic 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.

As an example, considering again the global type G_d above, we have

$$\mathbf{T}(\emptyset,G_d,\mathtt{A},\mathtt{B},0) = \mathtt{0}\,\mu\alpha_{\mathtt{AB}}\{\alpha_{\mathtt{AB}} := \mathtt{S}^{\ell_1}\,!\,[\mathtt{S}^{\ell_3-\ell_1}\,\mathtt{0}\,\alpha_{\mathtt{BA}}], \alpha_{\mathtt{BA}} := \mathtt{S}^{\ell_1}\,?\,[\mathtt{S}^{\ell_3-\ell_1}\,\mathtt{0}\,\alpha_{\mathtt{BA}}]\}$$

where we assume 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:

Theorem 6.1. *Let G be realizable. Then $\{x_{\mathtt{pq}} : \mathbf{T}(\emptyset,G,\mathtt{p},\mathtt{q},0) \mid \mathtt{q}\in\mathscr{R}\setminus\{\mathtt{p}\}\} \vdash \mathbf{P}(\emptyset,G,\mathtt{p})$ for every $\mathtt{p}\in\mathscr{R}$.*

It is easy to verify that the parallel composition of the $\mathbf{P}(\emptyset,G,\mathtt{p})$ is also well typed. Note that every global type with two participants and without $\parallel$ is realizable. This corresponds to a traditional binary session.

7 Concluding remarks

Existing type-based approaches for lock freedom impose severe restrictions on processes, especially recursive ones. Interestingly, *all of them* make use of rich behav-

ioral types (usages [11], session types [13], global types [9,1,8,7,3,4], conversation types [14]), and when they are able to handle recursive processes it is because they take advantage of such richness, almost suggesting that complex types are necessary to ensure a property as strong as lock freedom. We have shown that this is not the case: our modest extension of the linear type system for the π-calculus [12] ensures lock freedom for an unprecedented variety of process networks. Like other bottom-up approaches for lock freedom [11,13,14], our type systems associates information (subject ranks) with channels to constrain the order of input/output operations on them. Unlike other approaches, however, subject ranks have a relative interpretation (Lemma 4.1). This feature accounts for much of the expressiveness of our technique and enables a form of polymorphism that plays a crucial role in all the examples and encodings that have been discussed.

Given the abundance and diversity of type systems ensuring lock freedom, a detailed comparison with each of them seems unfeasible and perhaps not so interesting. We have chosen to assess the expressiveness of our type system by addressing two wide classes of networks whose processes interleave operations on different channels (indeed, it is such interleaving that determines the presence or absence of locks). Networks in the first class (Section 5) are characterized by a varying number of terminating processes, while networks in the second class (Section 6) are characterized by a fixed number of non-terminating processes. Our type system is also able to address combinations of these. For example, the following variation of $Node_A$ in Section 1

$$c_F!\langle c,c\rangle \mid *c_F?(x,y).(\nu a)(y!\langle a\rangle \mid x?(z).(\nu b)(c_F!\langle z,b\rangle \mid c_F!\langle b,a\rangle))$$

is an extreme case of non-terminating, dynamic system that develops like a *fairy ring*: each output on c_F spawns two processes connected by a new channel b. As a consequence, the initial output $c_F!\langle c,c\rangle$ bootstraps the growth of a ring of processes, each sending a message to its neighbour, and the ring doubles its diameter each time a whole round of communications is completed. The interested reader may check that this process is well typed.

Some processes are well typed according to other type systems, but not according to our own. For example, Kobayashi's usages [11] can handle non-linear channels in a more flexible way so that his modeling of the lock-free dining philosophers is not captured by our type system. There are also multiparty sessions which are not realizable with point-to-point communication. For instance, the global type

$$\mathtt{A} \to \mathtt{B}.[\mathtt{A} \to \mathtt{C}.[\mathtt{B} \to \mathtt{C} + \mathtt{C} \to \mathtt{B}] + \mathtt{A} \to \mathtt{C}.[\mathtt{B} \to \mathtt{C} + \mathtt{C} \to \mathtt{B}]]$$

is realizable in [1,4,3] but not according to our definition. The point is that B and C are both notified of A's decisions, hence they know whether it should be B to send a message to C or viceversa. In a native multiparty session the channel connecting A, B and C is one, so its type reflects this shared knowledge. But when communication is point-to-point, the types of the channels used by a single participant are independent and this shared knowledge is lost. Anyway, these examples are not pinpointing an intrinsic limitation of our approach (based on the ranking of channels), but rather are a consequence of the simple type discipline that we have chosen as foundation for our study. In fact, we do not see major obstacles in adding the $\mathtt{M}t$ construct to richer type languages.

There are some open questions regarding our type system that require further investigation. For example, is there a precise characterization for the class of well-typed processes? Is there a type reconstruction algorithm? The second question has practical relevance: as shown by Example 4.1, it is unrealistic to assume that the programmer is able to estimate subject and object ranks in any non-trivial process. Remarkably, type reconstruction algorithms are known for the linear π-calculus [10] and for Kobayashi's type system [11], both of which share several aspects with ours. In addition, all we had to do in Example 4.1 was to start from types where subject and object ranks are considered as variables, to derive a set of constraints between ranks (these constraints are always linear) and to apply "by hand" a linear constraint solver. Encouraged by these facts, we are now studying the problem of type reconstruction for our type discipline.

References

1. L. Bettini, M. Coppo, L. D'Antoni, M. D. 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. 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.
4. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. in Comp. Sci.*, to appear.
5. B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.
6. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.
7. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12*, LNCS 7211, pages 194–213. Springer, 2012.
8. P.-M. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Log. Meth. in Comp. Sci.*, 8(4), 2012.
9. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
10. A. Igarashi and N. Kobayashi. Type reconstruction for linear -calculus with i/o subtyping. *Inf. and Comp.*, 161(1):1–44, 2000.
11. N. Kobayashi. A type system for lock-free processes. *Inf. and Comp.*, 177(2):122–159, 2002.
12. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
13. L. Padovani. From lock freedom to progress using session types. In *PLACES'13*, to appear.
14. H. T. Vieira and V. T. Vasconcelos. Typing progress in communication-centred systems. In *COORDINATION'13*, LNCS 7890, pages 236–250. Springer, 2013.
15. P. Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286. ACM, 2012.

A Supplement to Section 4

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

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

$$\frac{A \vdash \mathsf{con}(t_i)^{\ (i=1,2)}}{A \vdash \mathsf{con}(t_1 \times t_2)} \qquad \frac{A \vdash \mathsf{con}(t_i)^{\ (i=1,2)}}{A \vdash \mathsf{con}(t_1 \oplus t_2)} \qquad \frac{A \cup \{\alpha_i \mid i \in I\} \vdash \mathsf{con}(t_i)^{\ (i \in I)}}{A \vdash \mathsf{con}(\mu\alpha\{\alpha_i := t_i\}_{i \in I})}$$

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 other than priority modifiers, at the cost of a more complicated definition of $\mathtt{M}$-rank (see (4.1)). In this article we do not investigate the issue further and we always assume to work with contractive types.

Basic properties. Below are two auxiliary results regarding the type system, namely the lifting 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 A.1 (lifting). *The following properties hold:*

1. *If* $\Gamma \vdash e : t$, *then* $\mathtt{S}\,\Gamma \vdash e : \mathtt{S}t$.
2. *If* $\Gamma \vdash P$, *then* $\mathtt{S}\,\Gamma \vdash P$.

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

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

We will make extensive use of Lemma A.1 in Sections B and C, to the point that it is convenient to extend the type system presented in Table 1 with the (admissible) rule

$$[\text{T-LIFT}]\ \frac{\Gamma \vdash P}{\mathtt{S}^m\,\Gamma \vdash P}$$

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. Let $\to$ be the least relation defined by the rule

$$\Gamma, a : \mathtt{S}^n\,\mathtt{O}^m\,\#[t] \to \Gamma \qquad \Gamma, a : \mathtt{O}t \to \Gamma, a : t$$

and let $\to^*$ be its reflexive, transitive closure. The first rule erases 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 second rule erases one $\mathtt{O}$ constructor from the type of a and is used each time a is sent in a message.

Lemma A.3. *Let* $\Gamma \vdash \mathsf{v} : \mathtt{O}t$. *Then there exists* Γ' *such that* $\Gamma \to^* \Gamma'$ *and* $\Gamma' \vdash \mathsf{v} : t$.

Proof. Easy induction on the derivation of $\Gamma \vdash \mathsf{v} : \mathtt{O}t$. □

The following result states that the type of an expression is the same as the type 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 A.4. *Let* $\Gamma \vdash e : t$ *and* $e \downarrow \mathsf{v}$. *Then* $\Gamma \vdash \mathsf{v} : t$.

The following result states that well typed processes reduce to well typed processes.

Lemma A.5. *Let* $\Gamma \vdash P$ *and* $P \to P'$. *Then* $\Gamma' \vdash P'$ *for some* Γ' *such that* $\Gamma \to^* \Gamma'$.

Proof. By induction on the derivation of $P \to P'$ and by cases on the last rule applied.

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

[R-COMM*] Then $P = a!\langle e\rangle \mid *a?(x).Q \to Q\{\mathsf{v}/x\} \mid *a?(x).Q = P'$ where $e \downarrow \mathsf{v}$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash a!\langle e\rangle$ and $\Gamma_2 \vdash *a?(x).Q$. From [T-OUT*] we deduce $\Gamma_1 = \Gamma_1' + a : !^{\omega}[t]$ and $\Gamma_1' \vdash e : \mathtt{S}^n\mathtt{O}t$. From [T-IN*] we deduce $\Gamma_2 = \Gamma_2' + a : ?^{\omega}[t]$ and $\Gamma_2', x : t \vdash Q$ and $\mathsf{un}(\Gamma_2')$. By Lemma A.4 we obtain $\Gamma_1' \vdash \mathsf{v} : \mathtt{S}^n\mathtt{O}t$. By Lemma A.3 we deduce that there exists Γ_1'' such that $\Gamma_1'' \vdash \mathsf{v} : \mathtt{S}^n t$ and $\Gamma_1' \to^* \Gamma_1''$. By Lemma A.1(2) we deduce $\mathtt{S}^n\Gamma_2', x : \mathtt{S}^n t \vdash Q$. From $\mathsf{un}(\Gamma_2')$ we deduce $\Gamma_2' \simeq \mathtt{S}^n\Gamma_2'$, therefore $\Gamma_1' + \mathtt{S}^n\Gamma_2'$ is defined. By Lemma A.2(2) we deduce $\Gamma_1'' + \Gamma_2' \vdash Q\{\mathsf{v}/x\}$. From [T-IN*] and [T-PAR] we obtain $\Gamma_1'' + \Gamma_2 \vdash Q\{\mathsf{v}/x\} \mid *a?(x).Q$. We conclude by taking $\Gamma' = \Gamma_1'' + \Gamma_2$ and observing that $\Gamma \to^* \Gamma'$.

[R-LET] Then $P = \mathtt{let}\, x, y = e\, \mathtt{in}\, Q$ and $e \downarrow \mathsf{v}, \mathsf{w}$ and $P' = Q\{\mathsf{v}, \mathsf{w}/x, y\}$. From [T-MATCH] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash e : t \times s$ and $\Gamma_2, x : t, y : s \vdash Q$. From Lemma A.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 A.2 we deduce $\Gamma \vdash Q\{\mathsf{v}, \mathsf{w}/x, y\}$ and we conclude by taking $\Gamma' = \Gamma$.

[R-CASE] Then $P = \mathtt{case}\ e\ \mathtt{of}\{\mathtt{inl}\ x_1 \Rightarrow P_1, \mathtt{inr}\ x_2 \Rightarrow P_2\}$ and either $e \downarrow \mathtt{inl}\ \mathsf{v}_1$ or $e \downarrow \mathtt{inr}\ \mathsf{v}_2$ 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 $\Gamma_2, x_i : t_i \vdash P_i$. From Lemma A.4 and [T-SUM] we deduce $\Gamma_1 \vdash \mathsf{v}_i : t_i$. From Lemma A.2 we deduce $\Gamma \vdash P_i\{\mathsf{v}_i/x_i\}$ and we conclude by taking $\Gamma' = \Gamma$. □

Soundness. In order to prove that the type system is sound, namely that well-typed processes are lock free, 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

Table 4. Depth of a channel.

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

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, where Γ is *balanced* if for every $a \in \mathsf{dom}(\Gamma)$ we have that $\Gamma(a) = \mathtt{0}^m\,\mathtt{S}^n\,\#^t[t]$.

Definition A.1. *Let* $\Gamma \vdash P$ *where* Γ *is balanced and* $a : t \in \Gamma$. *Then* $\mathsf{measure}(a,\Gamma,P) \stackrel{\text{def}}{=} (\mathtt{S}\text{-}\mathsf{rank}(t), \mathtt{0}\text{-}\mathsf{rank}(t), [\![P]\!]_a)$ *where* $[\![P]\!]_a$ *is inductively defined in Table 4.*

Given $\Gamma \vdash P$ where Γ is balanced and $a \in \mathsf{dom}(\Gamma)$, the measure of a is a triple consisting of the subject rank $\mathtt{S}\text{-}\mathsf{rank}(\Gamma(a))$ of a, the object rank $\mathtt{0}\text{-}\mathsf{rank}(\Gamma(a))$ 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 $<$.

Lemma A.6. *Let* **(1)** $\Gamma \vdash P$ *and* **(2)** Γ *balanced and* **(3)** *either* $\mathsf{input}(a,P)$ *or* $\mathsf{output}(a,P)$. *Then there exists* Q *such that* $P \to^* Q$ *and* $\mathsf{free}(a,Q)$.

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

$$\begin{array}{ll} n < n + \mathtt{S}\text{-}\mathsf{rank}(t) & \text{from } 0 < \mathtt{S}\text{-}\mathsf{rank}(t) \\ = \mathtt{S}\text{-}\mathsf{rank}(\mathtt{S}^n\,\mathtt{0}\,t) & \text{by definition of } \mathtt{S}\text{-}\mathsf{rank}(\cdot) \\ \leq n & \text{from } a \in \mathsf{fn}(e) \end{array}$$

  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 A.5 we deduce that $\Gamma' \vdash P'$ for some Γ' such that $\mathtt{0}\text{-}\mathsf{rank}(\Gamma'(a)) < \mathtt{0}\text{-}\mathsf{rank}(\Gamma(a))$, 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 $\mathtt{S}\text{-}\mathsf{rank}(\Gamma(b)) < \mathtt{S}\text{-}\mathsf{rank}(\Gamma(a))$. 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 A.5 there exists Γ' such that $\Gamma' \vdash P'$ and $\mathtt{0}\text{-}\mathsf{rank}(\Gamma'(a)) < \mathtt{0}\text{-}\mathsf{rank}(\Gamma(a))$. 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 $\mathtt{S}\text{-}\mathsf{rank}(\Gamma(b)) < \mathtt{S}\text{-}\mathsf{rank}(\Gamma(a))$. By induction hypothesis we derive $P \to^* \mathscr{C}'[b!\langle e\rangle \mid b?(x).R] \to \mathscr{C}'[R\{\mathsf{v}/x\}] = P'$ for some v such that $e \downarrow \mathsf{v}$. By Lemma A.5 we deduce $\Gamma' \vdash P'$ for some Γ' such that $\Gamma \to^* \Gamma'$ 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. □

B Supplement to Section 5

The following Lemma generalizes Theorem 5.1.

Lemma B.1. *If $\Delta \vdash M : \tau$ & m, n and $u \notin \mathsf{dom}(\Delta)$, then $[\![\Delta]\!], u : \mathtt{S}^m\,\mathtt{0}^n\,![[\![\tau]\!]] \vdash [\![M]\!]^u$.*

Proof. By induction on M and by cases on its shape.

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

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

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 $[\![\tau]\!] = \,!^{\omega}[[\![\tau_1]\!] \times \mathsf{S}^h\,\mathsf{0}^k\,!\,[[\![\tau_2]\!]]]$. We conclude with the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{[\![\Delta]\!], x : [\![\tau_1]\!], y : \mathsf{S}^h\,\mathsf{0}^k\,!\,[[\![\tau_2]\!]] \vdash [\![N]\!]^y}\,\text{IND.HYP.}}{[\![\Delta]\!], f : ?^{\omega}[[\![\tau_1]\!] \times \mathsf{S}^h\,\mathsf{0}^k\,!\,[[\![\tau_2]\!]]] \vdash *f?(x,y).[\![N]\!]^y}\,[\text{T-IN}^*] \quad \dfrac{\dfrac{}{f : [\![\tau]\!] \vdash f : [\![\tau]\!]}\,[\text{T-NAME}]}{f : [\![\tau]\!], u : \,!\,[[\![\tau]\!]] \vdash u!\langle f\rangle}\,[\text{T-OUT}]}{[\![\Delta]\!], f : \#^{\omega}[[\![\tau_1]\!] \times \mathsf{S}^h\,\mathsf{0}^k\,!\,[[\![\tau_2]\!]]], u : \,!\,[[\![\tau]\!]] \vdash *f?(x,y).[\![N]\!]^y \mid u!\langle f\rangle}\,[\text{T-PAR}]}{[\![\Delta]\!], u : \,!\,[[\![\tau]\!]] \vdash (\nu f)(*f?(x,y).[\![N]\!]^y \mid u!\langle f\rangle)}\,[\text{T-NEW}^*]$$

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

$\boxed{M = M_1 M_2}$ 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 $\mathscr{A}$

$$\dfrac{\dfrac{\vdots}{[\![\Delta]\!], a : \mathsf{S}^h\,\mathsf{0}^k\,!\,[[\![\sigma]\!]] \vdash [\![M_1]\!]^a}\,\text{IND.HYP.}}{}$$

as well as $\mathscr{B}$

$$\dfrac{\dfrac{\dfrac{\vdots}{[\![\Delta]\!], b : \mathsf{S}^{h'}\,\mathsf{0}^{k'}\,!\,[[\![\tau_1]\!]] \vdash [\![M_2]\!]^b}\,\text{IND.HYP.}}{[\![\Delta]\!], b : \mathsf{S}^{h''}\,\mathsf{0}^{k'}\,!\,[[\![\tau_1]\!]] \vdash [\![M_2]\!]^b}\,[\text{T-LIFT}]}{}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], y : [\![\tau_1]\!] \vdash y, u : \mathsf{S}^{m-m'}\,\mathsf{0}\,([\![\tau_1]\!] \times \mathsf{S}^{m'}\,\mathsf{0}^{n'}\,!\,[[\![\tau]\!]])}\,[\text{T-PAIR}]}{u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], x : \,!^{\omega}[[\![\tau_1]\!] \times \mathsf{S}^{m'}\,\mathsf{0}^{n'}\,!\,[[\![\tau]\!]]], y : [\![\tau_1]\!] \vdash x!\langle y,u\rangle}\,[\text{T-OUT}^*]}{u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], b : \mathsf{S}^{h''}\,\mathsf{0}^{k'}\,?[[\![\tau_1]\!]], x : [\![\sigma]\!] \vdash b?(y).x!\langle y,u\rangle}\,[\text{T-IN}]}{u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], a : \mathsf{S}^h\,\mathsf{0}^k\,?[[\![\sigma]\!]], b : \mathsf{S}^{h''}\,\mathsf{0}^{k'}\,?[[\![\tau_1]\!]] \vdash a?(x).b?(y).x!\langle y,u\rangle}\,[\text{T-IN}]}{[\![\Delta]\!], u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], a : \mathsf{S}^h\,\mathsf{0}^k\,\#[[\![\sigma]\!]], b : \mathsf{S}^{h''}\,\mathsf{0}^{k'}\,\#[[\![\tau_1]\!]] \vdash [\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y,u\rangle}\,[\text{T-PAR}]}{[\![\Delta]\!], u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]], a : \mathsf{S}^h\,\mathsf{0}^k\,\#[[\![\sigma]\!]] \vdash (\nu b)([\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y,u\rangle)}\,[\text{T-NEW}]}{[\![\Delta]\!], u : \mathsf{S}^m\,\mathsf{0}^n\,!\,[[\![\tau]\!]] \vdash (\nu a)(\nu b)([\![M_1]\!]^a \mid [\![M_2]\!]^b \mid a?(x).b?(y).x!\langle y,u\rangle)}\,[\text{T-NEW}]$$

($\mathscr{A}$ and $\mathscr{B}$ as above)

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

C Supplement to Section 6

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

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

Proof. By induction on G.

- ($G = \alpha$) We have:
  1. ($\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$) Then $t = \mathtt{nat}$ by definition of $\mathbf{T}$.
  2. ($\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$) Then $t = \mathtt{0}\,\alpha_{\mathtt{pq}}$ 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 $t = \mathtt{S}^{\ell-n}\,\mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = \mathtt{S}^{\ell-n}\,\mathtt{nat} = \mathtt{nat}$ 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 $t = \mathtt{S}^{\ell-n}\,\mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = \mathtt{S}^{\ell-n}\,\mathtt{S}^m\,\mathtt{0}^k\,\alpha_{\mathtt{pq}} = \mathtt{S}^{\ell-n+m}\,\mathtt{0}^k\,\alpha_{\mathtt{pq}}$ 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 $t = \mathtt{S}^h\,\mathtt{0}^k\,!\,[s]$ where $h = \ell - n$ and $h = 0$ 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 $\mathbf{T}(\Sigma, G', \mathtt{p}, \mathtt{q}, \ell) = \mathtt{S}^m\,\mathtt{0}^k\,p[s]$ for some $m > 0$ and we conclude $t = \mathtt{S}^h\,\mathtt{0}^k\,p[s]$ 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 $i = 1, 2\mathrm{m}$ 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 $\mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, n) = \mathtt{nat}$ 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 $\mathbf{T}(\Sigma, G_1, \mathtt{p}, \mathtt{q}, n) = \mathtt{S}^h\,\mathtt{0}^k\,\alpha_{\mathtt{pq}}$.
  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

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

and observe that Σ' is a sort environment for G' and $t = \mathtt{0}\, t_{\mathtt{pq}} = \mathtt{0}\, \mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n)\sigma$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \mathtt{0}\, t_{\mathtt{qp}} = \mathtt{0}\, \mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n)\sigma$ where $\sigma = \{t_{\mathtt{pq}}, t_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}$. 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 $t = \mathtt{nat}$.
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 $\mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n) = \mathtt{S}^h\, \mathtt{0}^m\, \beta_{\mathtt{pq}}$. We conclude by taking $k = m+1$ and observing that $t = \mathtt{S}^h\, \mathtt{0}^k\, \beta_{\mathtt{pq}}$.
3. $(\mathtt{p} \wr \mathtt{q} \in G)$ Then $\mathtt{p} \wr \mathtt{q} \in G'$ and by induction hypothesis we deduce $\mathbf{T}(\Sigma', G', \mathtt{p}, \mathtt{q}, n) = \mathtt{S}^h\, \mathtt{0}^m\, p[s]$ and $\mathbf{T}(\Sigma', G', \mathtt{q}, \mathtt{p}, n) = \mathtt{S}^h\, \mathtt{0}^m\, \overline{p}[s]$ for some $h > 0$. We conclude by taking $k = m+1$ and observing that $t = \mathtt{S}^h\, \mathtt{0}^k\, p[s\sigma]$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \mathtt{S}^h\, \mathtt{0}^k\, \overline{p}[s\sigma]$.

□

Definition C.2 (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. $\sigma(\alpha_{\mathtt{pq}}) = \sigma(\alpha_{\mathtt{qp}}) = \mathtt{nat}$ *for every* $\mathtt{p} \wr \mathtt{q} \notin \Sigma(\alpha)$;
3. $\sigma(\alpha_{\mathtt{pq}}) = \mathtt{S}^m\, \mathtt{0}^n\, p[t]$ *and* $\sigma(\alpha_{\mathtt{qp}}) = \mathtt{S}^m\, \mathtt{0}^n\, \overline{p}[t]$ *and* $m > 0$ *for every* $\mathtt{p} \wr \mathtt{q} \in \Sigma(\alpha)$.

Lemma C.2. *If $\Sigma' = \Sigma, \alpha : \Sigma(\mu\alpha.G)$, then $\Sigma(\mu\alpha.G) = \Sigma'(G)$.*

Proof. We have

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

□

Lemma C.3. *Let*

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

Then σ' is a Σ'-substitution.

Proof. We must prove the three conditions of Definition C.2. 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 C.2, we have $\Sigma'(\alpha) = \Sigma(\mu\alpha.G) = \Sigma'(G)$.

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

$$\begin{array}{ll}
\sigma'(\alpha_{\mathtt{pq}}) = s_{\mathtt{pq}}\sigma & \text{by definition of } \sigma' \\
\quad = \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by unfolding of } s_{\mathtt{pq}} \\
\quad = \mathtt{nat}\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by Lemma C.1} \\
\quad = \mathtt{nat} & \text{by definition of type substitution}
\end{array}$$

Regarding condition 3 of Definition C.2, 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

$$\begin{array}{ll}
\sigma'(\alpha_{\mathtt{pq}}) = \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by definition of } \sigma' \\
\quad = \mathtt{S}^h\,\mathtt{0}^k\,\beta_{\mathtt{pq}}\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by Lemma C.1(2)} \\
\quad = \mathtt{S}^h\,\mathtt{0}^k\,\sigma(\beta_{\mathtt{pq}}) & \text{because } \beta \neq \alpha \\
\quad = \mathtt{S}^h\,\mathtt{0}^k\,\mathtt{S}^{h'}\,\mathtt{0}^{k'}\,p[s] & \text{because } \sigma \text{ is a } \Sigma\text{-substitution} \\
\quad = \mathtt{S}^{h+h'}\,\mathtt{0}^{k+k'}\,p[s] & \text{equality on types}
\end{array}$$

and similarly we obtain $\sigma'(\alpha_{\mathtt{pq}}) = \mathtt{S}^{h+h'}\,\mathtt{0}^{k+k'}\,\overline{p}[s]$.

– ($\mathtt{p} \wr \mathtt{q} \in G$) We have

$$\begin{array}{ll}
\sigma'(\alpha_{\mathtt{pq}}) = \mathbf{T}(\Sigma', G, \mathtt{p}, \mathtt{q}, n)\{s_{\mathtt{pq}}, s_{\mathtt{qp}}/\alpha_{\mathtt{pq}}, \alpha_{\mathtt{qp}}\}\sigma & \text{by definition of } \sigma' \\
\quad = \mathtt{S}^h\,\mathtt{0}^k\,p[t] & \text{by Lemma C.1(3)}
\end{array}$$

and similarly we obtain $\sigma'(\alpha_{\mathtt{qp}}) = \mathtt{S}^h\,\mathtt{0}^k\,\overline{p}[t]$. □

Lemma C.4. *Let $n < G$ and σ be a Σ-substitution. Then:*

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

Proof. If $\mathtt{p} \wr \mathtt{q} \notin \Sigma(G)$, then by Lemma C.1 we deduce $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = \mathtt{nat}$ 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 C.1 we have $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = \mathtt{S}^h\,\mathtt{0}^k\,p[s]$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \mathtt{S}^h\,\mathtt{0}^k\,\overline{p}[s]$ 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 C.1 we have $\mathbf{T}(\Sigma, G, \mathtt{p}, \mathtt{q}, n) = \mathtt{S}^{h'}\,\mathtt{0}^{k'}\,\alpha_{\mathtt{pq}}$ and $\mathbf{T}(\Sigma, G, \mathtt{q}, \mathtt{p}, n) = \mathtt{S}^{h'}\,\mathtt{0}^{k'}\,\alpha_{\mathtt{qp}}$. By definition of Σ-substitution we deduce $\sigma(\alpha_{\mathtt{pq}}) = \mathtt{S}^{h''}\,\mathtt{0}^{k''}\,p[s]$ and $\sigma(\alpha_{\mathtt{qp}}) = \mathtt{S}^{h''}\,\mathtt{0}^{k''}\,\overline{p}[s]$ 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:

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

Lemma C.5. *Let σ be a Σ-substitution and $n < G$. Then $\Theta_{[\Sigma,\mathrm{p}]}\sigma, \Gamma_{[\Sigma,G,\mathrm{p},n]}\sigma \vdash \mathbf{P}(\Sigma,G,\mathrm{p})$.*

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

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

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

We derive:

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

where we use the fact that $0 \leq \mathsf{S}\text{-}\mathsf{rank}(s_{\mathrm{pq}})$ which follows from the hypothesis that σ is a Σ-substitution.

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

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

and observe that σ' is a Σ'-substitution by Lemma C.3. We have:

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

We derive:

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

$\boxed{G = \mathtt{A} \to \mathtt{B}@\ell.G' \text{ and } \mathtt{p} = \mathtt{A}}$ We consider the case $\mathtt{A} \wr \mathtt{B} \in \Sigma(G')$, the other being analogous. Then, from Lemma C.4, we know that

$$\begin{aligned}\mathbf{T}(\Sigma, G', \mathtt{A}, \mathtt{B}, \ell)\sigma &= \mathtt{S}^h\,\mathtt{0}^k\,p[s]\\ \mathbf{T}(\Sigma, G', \mathtt{B}, \mathtt{A}, \ell)\sigma &= \mathtt{S}^h\,\mathtt{0}^k\,\overline{p}[s]\end{aligned}$$

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

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

We derive $\mathscr{A}$:

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

as well as $\mathscr{B}$:

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

From these we obtain:

$$\frac{\dfrac{\mathscr{A} \qquad \mathscr{B}}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma, G, \mathtt{p}, n]}\sigma, a : \mathtt{S}^{\ell-n+h}\,\mathtt{0}^{2k+1}\,\#[s] \vdash x_{\mathtt{AB}}!\langle a\rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p})}\ [\text{T-PAR}]}{\Theta_{[\Sigma,\mathtt{p}]}\sigma, \Gamma_{[\Sigma, G, \mathtt{p}, n]}\sigma \vdash (\nu a)(x_{\mathtt{AB}}!\langle a\rangle \mid \mathtt{let}\ x_{\mathtt{AB}} = a\ \mathtt{in}\ \mathbf{P}(\Sigma, G', \mathtt{p}))}\ [\text{T-NEW}]$$

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

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

We derive:

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

where the application of [T-IN] uses Lemma C.4 for deducing that the subject priorities 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:

$$\mathbf{P}(\Sigma, G, \mathtt{p}) = \mathbf{P}(\Sigma, G', \mathtt{p})$$
$$\Gamma_{[\Sigma,G,\mathtt{p},n]}\sigma = \mathbf{x}_{[\mathtt{p}]} : \mathbf{T}(\Sigma, G, [\mathtt{p}], n)\sigma = \mathbf{x}_{[\mathtt{p}]} : \mathtt{S}^{\ell-n}\,\mathbf{T}(\Sigma, G', [\mathtt{p}], \ell)\sigma = \mathtt{S}^{\ell-n}\,\Gamma_{[\Sigma,G',\mathtt{p},\ell]}\sigma$$

We derive:

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

□

Theorem C.1 (Theorem 6.1). *Let G be realizable. Then $\{x_{\mathtt{pq}} : \mathbf{T}(\emptyset, G, \mathtt{p}, \mathtt{q}, 0) \mid \mathtt{q} \in \mathscr{R} \setminus \{\mathtt{p}\}\} \vdash \mathbf{P}(\emptyset, G, \mathtt{p})$ for every $\mathtt{p} \in \mathscr{R}$.*

Proof. Consequence of Lemma C.5 by taking $\sigma = \emptyset$. □