# A Bunch of Sessions: A Propositions-as-Sessions Interpretation of Bunched Implications in Channel-Based Concurrency

EMANUELE D'OSUALDO, MPI-SWS, Germany
DAN FRUMIN, BAS VAN DEN HEUVEL, and JORGE A. PÉREZ, Bernoulli Institute, University of Groningen, The Netherlands

The emergence of *propositions-as-sessions*, a Curry-Howard correspondence between propositions of Linear Logic and session types for concurrent processes, has settled the logical foundations of message-passing concurrency. Central to this approach is the *resource consumption* paradigm heralded by Linear Logic.

In this paper, we investigate a new point in the design space of session type systems for message-passing concurrent programs. We identify O'Hearn and Pym's *Logic of Bunched Implications* (BI) as a fruitful basis for an interpretation of the logic as a concurrent programming language. This leads to a treatment of *non-linear resources* that is radically different from existing approaches based on Linear Logic. We introduce a new $\pi$-calculus with sessions, called $\pi$BI; its most salient feature is a construct called *spawn*, which expresses new forms of sharing that are induced by structural principles in BI. We illustrate the expressiveness of $\pi$BI and lay out its fundamental theory: type preservation, deadlock-freedom, and weak normalization results for well-typed processes; an operationally sound and complete typed encoding of an affine $\lambda$-calculus; and a non-interference result for access of resources.

## 1 INTRODUCTION

In this paper, we investigate a new point in the design space of session type systems for message-passing concurrent programs. We identify the *Logic of Bunched Implications* (BI) of O'Hearn and Pym [1999] as a fruitful basis for an interpretation of the logic as a concurrent programming language, in the style of *propositions-as-sessions* [Caires and Pfenning 2010; Wadler 2012]. This leads to a treatment of *non-linear resources* that is radically different from existing approaches based on Girard's Linear Logic (LL). We propose $\pi$BI, the first concurrent interpretation of BI, and we study the behavioral properties enforced by typing, laying the meta-theoretical foundations needed, and clarifying its relation to the other type-theoretic interpretations of BI.

*Session types for message-passing concurrency.* Writing concurrent programs is notoriously hard, as bugs might be caused by subtle undesired interactions between processes. Statically enforcing the absence of bugs while allowing expressive concurrency patterns is important but difficult. In the context of message-passing concurrency, type systems based on *session types* provide an effective approach. Session type systems enforce a communication structure between processes and channels, with the intent of (statically) ruling out races (as in, e.g., two threads sending messages over the same channel at the same time) and other undesirable behaviors, like deadlocks. This communication structure is formulated at the type level. For example, the session type $T = \,!\mathsf{int}.?\mathsf{string}.!\mathsf{bool}.\mathsf{end}$ (written in the syntax of [Vasconcelos 2012]) describes a protocol that first outputs an integer ($!\mathsf{int}$), then inputs a string ($?\mathsf{string}$), and finally outputs a boolean ($!\mathsf{bool}$). In session-based concurrency, types are assigned to channel names; this way, e.g., the assignment $x : T$ dictates that the communications on channel $x$ must adhere to the protocol described by $T$.

The fundamental idea behind session type systems is that an assumption such as $x : T$ is like a *resource* that can be consumed and produced. For example, the act of sending an integer on the channel $x$ consumes $x : !\mathsf{int}.?\mathsf{string}.!\mathsf{bool}.\mathsf{end}$ and produces a new resource $x : ?\mathsf{string}.!\mathsf{bool}.\mathsf{end}$, representing the expected continuation of the protocol. Then, the coordinated use of channels requires a strict discipline on how resources can be consumed and produced: it is unwise to allow multiple processes to access the same resource $x : T$, otherwise simultaneous concurrent outputs by different processes on the same channel will render the protocol invalid. The type system is thus designed to enforce that some resources, like those associated with channels, are *linear*: they are consumed *exactly once*. By enforcing linearity of these resources, session type systems ensure that well-typed programs conform to the protocols encoded as types, and satisfy important correctness properties, such as deadlock-freedom.

*Propositions-as-sessions.* A central theme in this paper is how logical foundations can effectively inform the design of expressive type disciplines for programs. In the realm of functional programming languages, such logical foundations have long been understood via type systems obtained through strong Curry-Howard correspondences with known logical proof systems (e.g. the correspondence between the simply-typed $\lambda$-calculus and intuitionistic propositional logic). For concurrent languages, on the other hand, such correspondences have been more elusive. Indeed, although the original works on session types by Honda [1993]; Honda et al. [1998] feature an unmistakable influence of LL in their formulation, the central question of establishing firm logical foundations for session types remained open until relatively recently. The first breakthroughs were the logical correspondences based on the concurrent languages $\pi$DILL [Caires and Pfenning 2010] and CP [Wadler 2012] (based on Intuitionistic LL and Classical LL, respectively). These works define a bidirectional correspondence, in the style of Curry-Howard, which allows us to interpret propositions as session types (protocols), proofs as $\pi$-calculus processes, and cut elimination as process communication. These correspondences are often collectively referred to as *propositions-as-sessions*.

Intensely studied in the last decade, the line of work on propositions-as-sessions provides a principled justification to a linear typing discipline. These correspondences also clarify our understanding of the status of *non-linear* resources, which do not obey resource consumption considerations. Non-linear resources, such as mutable references, client/server channels, and shared databases, are commonplace in practical programs and systems. Disciplining non-linear resources is challenging, because there is a tension between flexibility and correctness: ideally, one would like to increase the range of (typable) programs that can be written, while ensuring that such programs treat non-linear resources consistently.

LL allows for a controlled treatment of non-linear resources through the modality $!A$. Within propositions-as-sessions, the idea is that a session of type $!A$ represents a server providing a session of type $A$ to its clients, and the server itself can be duplicated or dropped. Those particular features—being able to replicate or drop a session—are achieved through the usage of *structural rules* in the sequent calculus, specifically the rules of contraction and weakening, which are restricted to propositions of the form $!A$. A series of recent works have explored quite varied ways of going beyond this treatment of non-linear resources: they have put forward concepts such as *manifest sharing* [Balzer and Pfenning 2017], dedicated frameworks such as *client-server logic* [Qian et al. 2021], and specific constructs for non-deterministic, fail-prone channels [Caires and Pérez 2017].

*The Logic of Bunched Implications.* At their heart, the aforementioned works propose different ways of treating non-linear resources through modalities. Relaxing linearity through a modality allows a clean separation between the worlds of linear and non-linear resources. This approach relies on rules that act as "interfaces" between the two words, allowing conversions between linear and non-linear types only under controlled circumstances.

However, modalities are not the only way in which substructural logics can integrate non-linear resources. A very prominent alternative is provided by the Logic of Bunched Implications (BI) of O'Hearn and Pym [1999]. BI embeds the pure linear core of LL as multiplicative conjunction $*$ and implication $\mathrel{-\!\!*}$, but extends it by introducing additive conjunction $\wedge$ and implication $\rightarrow$, which are treated non-linearly. BI can thus be thought of as enabling the free combination of linear and non-linear resources in a single coherent logic.

The result is a logic which admits an interpretation of linearity that is enticingly different from LL. Conceptually, LL admits a "number of uses" interpretation, where types can specify *how many times a resource should be used*: exactly once for linear resources, any number of times for $!A$ resources. On the contrary, BI admits an "ownership" interpretation [Pym et al. 2004], which focuses on *who has access to which resources.*

The ownership interpretation has positioned BI as the logic of choice for program logics for reasoning about stateful and concurrent programs, under the umbrella of (Concurrent) Separation Logic (see, e.g., the surveys by O'Hearn [2019] and Brookes and O'Hearn [2016]). While separation logic has received significant attention, the same cannot be said about type-theoretic interpretations of BI as a type system for concurrency. To our knowledge, the only type-theoretic investigation into the (proof theory of) BI has been the $\alpha\lambda$-calculus [O'Hearn 2003]—a $\lambda$-calculus arising from the natural deduction presentation of BI—and its variations [Atkey 2004; Collinson et al. 2008].

*Our key idea.* Here we propose $\pi$BI: the first process calculus for the propositions-as-sessions and processes-as-proofs interpretation of BI, based on its sequent calculus formulation. The result is an expressive concurrent calculus with a new mechanism to handle non-linear resources, which satisfies important behavioral properties, derived from a tight correspondence with BI's proof theory. The central novelty of $\pi$BI is a process interpretation of the structural rules, which closely follows the proof theory of BI.

Consider the case of contraction/duplication. Given a session $x : A$, how can we duplicate it into sessions $x_1 : A$ and $x_2 : A$? The difficulty here is that after duplication, the two assumptions might be used differently and asynchronously. We conclude that the actual process implementing those sessions in the current evaluation context needs to be duplicated, such that two independent processes can provide the duplicated sessions. This "on demand non-local replication" of a process in the evaluation context is not something supported natively by the $\pi$-calculus. We propose a new process construct, a prefix dubbed *spawn*, which achieves this.

We illustrate the spawn prefix with a simple example. Let $P$ and $Q$ be two processes, with $P$ providing a service on the channel $x$, and $Q$ requiring two copies of the service. The spawn prefix $\boldsymbol{\rho}[x \mapsto x_1, x_2]$ denotes a request to the environment to duplicate the service on $x$ into copies on the new channels $x_1$ and $x_2$. Then, $\boldsymbol{\rho}[x \mapsto x_1, x_2].Q$ is a process that first performs the request and then behaves as $Q$. The composition of these processes is denoted $(\boldsymbol{\nu}x).(P \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q)$, where '$\mid$' and '$(\boldsymbol{\nu}x)$' stand for parallel composition and restriction on $x$, respectively.

In the reduction semantics of $\pi$BI, obtained from the proof theory of BI, the composed process reduces as follows:

$$(\boldsymbol{\nu}x).(P \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q) \longrightarrow (\boldsymbol{\nu}x_1).\big(P[x_1/x] \mid (\boldsymbol{\nu}x_2).(P[x_2/x] \mid Q)\big).$$

This way, the duplication request leads to the composition of two copies of $P$ (each with an appropriate substitution $[x_i/x]$) with the process $Q$ on channels $x_1$ and $x_2$, as desired.

The behavior of the spawn prefix is determined by the context in which it is executed and it communicates with the run-time system to achieve contraction or weakening. This mechanism reminds us of horizontal scaling in cloud computing, with the spawn prefix playing the role of middleware: it requests the runtime environment to scale up/down a particular resource. For example, a load balancer might determine that in a certain situation the execution environment has to provide an additional snapshot of a Docker container, and route part of the environment's requests to it.

As we will see, spawn reductions involve the propagation of the effects of duplicating processes (such as $P$ above); we give the full definition and illustrate it further in Section 2.

*Contributions.* As mentioned, the spawn prefix provides a direct interpretation of the structural rules in the design of the type system, adopting BI as the underlying logic. The resulting system is significantly expressive and yet different from systems derived from propositions-as-sessions, which is not so surprising: as logics, BI and LL are *incomparable*: there are provable formulas of LL that are not provable in BI, and vice versa. As such, an immediate question is whether $\pi$BI satisfies the expected meta-theoretical properties for session-typed processes: *type preservation* and *deadlock-freedom*. The key difficulty is that the semantics of the spawn prefix is fundamentally *non-local*—it depends on its execution context. As a **first contribution**, we show that type preservation and deadlock-freedom hold for $\pi$BI; moreover, we prove *weak normalization*, which further justifies the semantics of spawn prefixes.

In addition to these meta-theoretical properties, an essential ingredient in the propositions-as-sessions research program is defined by concurrent interpretations of (typed) functional calculi, in the spirit of Milner's seminal work on *functions-as-processes* [Milner 1992]. As already mentioned, the only prior type-theoretic interpretation of BI is the (sequential) calculus $\alpha\lambda$-calculus [O'Hearn 2003]. As a **second contribution**, we define a translation from $\alpha\lambda$-calculus into $\pi$BI, and prove that it correctly preserves and reflects the operational semantics of terms and processes, respectively.

While insightful and novel, the operational semantics of $\pi$BI and the translation of the $\alpha\lambda$-calculus do not offer us a direct insight in the meaning of and difference between the types in our system (as is the case in the $\alpha\lambda$-calculus). A natural question is: what is the difference between multiplicative conjunction $*$ and additive conjunction $\wedge$ in $\pi$BI? As an answer to this question, our **third contribution** is a *denotational semantics* for $\pi$BI, which interprets processes as functions and describes types in terms of "provenance tracking".

Intuitively, our denotational semantics considers that duplication through a spawn prefix generates typed processes with the same provenance. This notion of provenance then allows us to precisely distinguish between $*$ and $\wedge$: in a process with a session of type $A * B$ the sub-processes providing sessions $A$ and $B$ have a *different origin*, a property that may not necessarily hold for processes with sessions of type $A \wedge B$. This is possible because the provenance information can be reconstructed from a typing derivation, and it is made evident through the denotational semantics.

In addition to providing a semantic meaning to types, the denotational semantics is sound with regard to *observational equivalence*. Two processes are observationally equivalent if no other process can (operationally) distinguish between them. Establishing observational equivalence of programs directly is hard, because it involves reasoning about process behavior under arbitrary contexts. On the other hand, a denotational semantics provides a direct way of establishing equivalence: if two processes have the same denotation, then they are observationally equivalent. As an application of the denotational semantics, we frame the operational correspondence for the $\alpha\lambda$-calculus mentioned above in terms of observational equivalence.

*Outline.* The rest of the paper is organized as follows. Section 2 presents the syntax, semantics, and type system of $\pi$BI, and illustrates its expressivity. In Section 3 we establish key meta-theoretical properties of typable processes: type preservation, deadlock freedom, and weak normalization. We formally connect the $\alpha\lambda$-calculus to $\pi$BI by defining a translation and proving operational correspondence for it in Section 4. We define the denotational semantics for $\pi$BI processes, define observational equivalence, and formally relate the two in Section 5. We discuss further related work in Section 6 and conclude in Section 7. The omitted technical details can be found in the appendix.

## 2 THE $\pi$BI CALCULUS

In this section we formally introduce $\pi$BI, a $\pi$-calculus with constructs for session-based concurrency [Honda 1993; Honda et al. 1998] and our new spawn prefix. We first describe syntax and

| | | | | |
|---|---|---|---|---|
| $P, Q, R ::= \overline{x}[y].(P \mid Q)$ | output | $\mid$ | $x(y).P$ | input |
| $\mid\ \overline{x}\langle\rangle$ | close | $\mid$ | $x().P$ | wait |
| $\mid\ x \triangleleft \mathsf{inl}.P$ | left selection | $\mid$ | $x \triangleright \mathsf{case}(P, Q)$ | branch |
| $\mid\ x \triangleleft \mathsf{inr}.P$ | right selection | $\mid$ | $[x \leftarrow y]$ | forwarder |
| $\mid\ (\boldsymbol{\nu}x).(P \mid Q)$ | restriction + parallel | $\mid$ | $\boldsymbol{\rho}[\sigma].P$ | spawn |

Fig. 1. Syntax of $\pi$BI processes.

dynamics (reduction semantics), and then present its associated type system, based on the sequent calculus for BI. Following $\pi$DILL [Caires and Pfenning 2010; Caires et al. 2016], our type system for $\pi$BI admits a "provide/use" reading for typable processes, whereby a specific channel *provides* a session by *using* zero or more other sessions.

*Notation.* We assume an enumerable set of *names* (or *channels*), $a, b, c, \ldots, x, y, z \in \mathrm{Name}$ to denote channels. We make use of finite partial functions $f\colon A \xrightarrow{\mathsf{fin}} B$. We write $f(x) = \bot$ if $f$ is not defined on $x$. We define $\mathrm{dom}(f) = \{x \in A \mid f(x) \neq \bot\}$. We write $[a_1 \mapsto b_1;\ \ldots;\ a_n \mapsto b_n]$ to denote a map, and $\emptyset$ for the empty map. We will also use set comprehensions for finite functions, e.g. $[a \mapsto b \mid a \in \{1, 2\}, b = a^2]$. For a finite partial function $f$ and a set $X$, we write $f \setminus X$ for the function that coincides with $f$ except for being undefined on $X$.

### 2.1 Process Syntax

The syntax of $\pi$BI is given in Figure 1. The structure and conventions of process calculi based on Curry-Howard correspondences are typically based on an implicit expectation for how the components of a system are organized — an expectation that is ultimately verified by typing. The idea is that interaction is grouped into a *session*, the sequence of interactions along a single channel. As hinted at above, a process $P$ should *provide* a session at some specific channel $x \in \mathrm{fn}(P)$, and there is always a single *user* of the session exchanging messages with $P$ along $x$. To provide a session, a process can make use of sessions on other channels.

Most constructs are standard and reflect these expectations of sessions with provide/use roles:

- *Input/Output:* A process $x(y).P$ receives a channel $y$ from the session at $x$ and proceeds as $P$, continuing the session at $x$.
  A process $\overline{x}[y].\big(P \mid Q\big)$ sends a *fresh* channel $y$ over the session at $x$; the process $P$ provides the new session at $y$, while $Q$ continues the session at $x$.
- *Labelled choice (selection and branching):* The processes $x \triangleleft \mathsf{inl}.P$ and $x \triangleleft \mathsf{inr}.P$ select left/right labels over the session at $x$, respectively. The dual process $x \triangleright \mathsf{case}(P, Q)$ offers these left/right options, which trigger continuation $P$ or $Q$, respectively.
- *Explicit session closing:* The end of a session is expected to be explicitly closed by a final handshake between the dual prefixes $\overline{x}\langle\rangle$ and $x().P$ (empty output/input, respectively).
- *Structured parallel composition:* Parallel composition, in keeping with $\pi$DILL [Caires et al. 2016], is used jointly with restriction. In a process $(\boldsymbol{\nu}x).(P \mid Q)$ a new session is created at $x$, provided by $P$ with $Q$ as its only user. To improve readability, we sometimes annotate the parallel operator with the name of the associated restriction, and write $(\boldsymbol{\nu}x).(P \mid_x Q)$.
- *Forwarders:* A process $[x \leftarrow y]$ provides a session at $x$ as a copycat of the session at $y$.

The key novel construct of $\pi$BI is the *spawn prefix* $\boldsymbol{\rho}[\sigma].P$. It is parametrized by what we call a *spawn binding* $\sigma$. Spawn bindings, formally defined below, are a unification and generalisation of prefixes like $\boldsymbol{\rho}[x \mapsto x_1, x_2]$ (copy the session at $x$ to $x_1$ and $x_2$) but also $\boldsymbol{\rho}[x \mapsto \emptyset]$ (drop the session

CONG-ASSOC-L
$(\boldsymbol{\nu}x).\big(P \mid_x (\boldsymbol{\nu}y).(Q \mid_y R)\big) \equiv (\boldsymbol{\nu}y).\big(Q \mid_y (\boldsymbol{\nu}x).(P \mid_x R)\big)$ *(when $x \notin \mathrm{fn}(Q) \wedge y \notin \mathrm{fn}(P)$)*

CONGR-ASSOC-R
$(\boldsymbol{\nu}x).\big(P \mid_x (\boldsymbol{\nu}y).(Q \mid_y R)\big) \equiv (\boldsymbol{\nu}y).\big((\boldsymbol{\nu}x).(P \mid_x Q) \mid_y R\big)$ *(when $x \notin \mathrm{fn}(R) \wedge y \notin \mathrm{fn}(P)$)*

CONGR-SPAWN-SWAP
$\boldsymbol{\rho}[\sigma_1].\boldsymbol{\rho}[\sigma_2].Q \equiv \boldsymbol{\rho}[\sigma_2].\boldsymbol{\rho}[\sigma_1].Q$ *(when $\sigma_1 \# \sigma_2$)*

Fig. 2. Structural congruence.

at $x$). Indeed, in addition to allowing the simultaneous mapping of more than one name $x$, we allow names to be mapped to sets of names, encompassing the nullary and binary cases above.

*Definition 2.1 (Spawn binding).* A function $\sigma\colon \mathsf{Name} \xrightarrow{\text{fin}} \wp(\mathsf{Name})$ is a *spawn binding* if: (*1*) $\forall x, y \in \mathrm{dom}(\sigma)\colon \sigma(x) \cap \sigma(y) = \emptyset$, and (*2*) $\forall x \in \mathrm{dom}(\sigma)\colon \mathrm{dom}(\sigma) \cap \sigma(x) = \emptyset$.

We define the *restrictions* of $\sigma$ to be the set $\mathrm{restr}(\sigma) = \bigcup_{x \in \mathrm{dom}(\sigma)} \sigma(x)$. We omit redundant delimiters in spawn prefixes, e.g. we write $\boldsymbol{\rho}[x \mapsto x_1, x_2; y \mapsto y_1]$ for $\boldsymbol{\rho}[[x \mapsto \{x_1, x_2\}; y \mapsto \{y_1\}]]$.

Given two spawn bindings $\sigma_1$ and $\sigma_2$ we say they are *independent*, written $\sigma_1 \# \sigma_2$, if $\mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$, $\mathrm{dom}(\sigma_1) \cap \mathrm{restr}(\sigma_2) = \emptyset$, $\mathrm{restr}(\sigma_1) \cap \mathrm{restr}(\sigma_2) = \emptyset$, and $\mathrm{dom}(\sigma_2) \cap \mathrm{restr}(\sigma_1) = \emptyset$.

*Free and bound names.* Except for the new spawn construct, the notion of free and bound names is standard: the processes $\overline{x}[y].(P \mid Q)$, $x(y).P$, and $(\boldsymbol{\nu}y).(P \mid Q)$ all bind $y$. For the spawn prefix, the situation is a bit different. Given a set of names $X$, a spawn $\boldsymbol{\rho}[x \mapsto X].P$ signals to the context that $P$ will use $n = |X|$ times the session at $x$. The names in $X$ indicate the new names that $P$ will use instead of $x$. As such, these new names are bound in $P$ by the spawn prefix, whereas the original name $x$ is free in $P$. Formally, $\mathrm{fn}(\boldsymbol{\rho}[\sigma].P) = \big(\mathrm{fn}(P) \setminus \mathrm{restr}(\sigma)\big) \cup \mathrm{dom}(\sigma)$.

We implicitly identify processes up to $\alpha$-conversion and we adopt Barendregt's variable convention: all bound names are different, and bound names are different from free names.

*Structural congruence.* As usual, we define a congruence that identifies processes up to inconsequential syntactical differences. *Structural congruence*, denoted $\equiv$, is the smallest congruence satisfying the rules in Figure 2: the orders of parallel compositions and independent spawn prefixes do not matter (Rules CONGR-ASSOC-R and CONG-ASSOC-L and Rule CONGR-SPAWN-SWAP, resp.).

Our structural congruence is a bit more fine-grained than is usual for the $\pi$-calculus. This is guided by the desire to make typing consistent under structural congruence. Typing will enforce the expectations of process structure alluded to before, so our congruence needs to preserve them. For example, in a process $\overline{x}[y].(P \mid Q)$ we expect $P$ to provide the new session at $y$ and $Q$ to continue the session at $x$. Admitting commutativity of parallel would break this expectation. Similarly, in the composition of processes $(\boldsymbol{\nu}x).(P \mid Q)$ it is important that $P$ provides the session that governs $x$, and that $Q$ dually uses the session at $x$. This choice of structural congruence simplifies the technical development and makes the correspondence between logic and type theory sharper.

## 2.2 Reduction Semantics

The operational semantics of $\pi$BI is defined in terms of a reduction relation, denoted $\longrightarrow$, which combines the usual reductions of the $\pi$-calculus with reductions for spawn prefixes. As usual, we shall write $\longrightarrow^*$ to denote the reflexive, transitive closure of $\longrightarrow$, and $P \not\longrightarrow$ when $P$ cannot reduce.

$$\textsc{red-comm-r}\quad (\nu x).(x(y).Q \mid_x \overline{x}[y].(P_1 \mid P_2)) \longrightarrow (\nu x).((\nu y).(P_1 \mid_y Q) \mid_x P_2)$$

$$\textsc{red-unit-r}\quad (\nu x).(x().Q \mid_x \overline{x}\langle\rangle) \longrightarrow Q$$

$$\textsc{red-case}\quad \frac{\ell \in \{\mathsf{inl}, \mathsf{inr}\}}{(\nu x).(x \triangleleft \ell.P \mid x \triangleright \mathsf{case}(Q_{\mathsf{inl}}, Q_{\mathsf{inr}})) \longrightarrow (\nu x).(P \mid Q_\ell)}$$

$$\textsc{red-fwd-r}\quad (\nu x).(P \mid_x [y \leftarrow x]) \longrightarrow P[y/x]$$

$$\textsc{red-spawn}\quad \frac{\sigma(x) = \{x_1, \ldots, x_n\} \qquad \sigma' = ((\sigma \setminus \{x\}) \cup [z \mapsto \{z_1, \ldots, z_n\} \mid z \in \mathrm{fn}(P) \setminus \{x\}])}{(\nu x).(P \mid_x \rho[\sigma].Q) \longrightarrow \rho[\sigma'].(\nu x_1).(P^{(1)} \mid_{x_1} \ldots (\nu x_n).(P^{(n)} \mid_{x_n} Q) \ldots)}$$

$$\textsc{red-spawn-r}\quad \frac{x \notin \mathrm{dom}(\sigma)}{(\nu x).(P \mid_x \rho[\sigma].Q) \longrightarrow \rho[\sigma].(\nu x).(P \mid_x Q)}$$

$$\textsc{red-spawn-merge}\quad \rho[\sigma_1].\rho[\sigma_2].P \longrightarrow \rho[\sigma_1 \ltimes \sigma_2].P$$

$$\begin{aligned}\mathcal{K}[\,\cdot\,] ::= {} & [\,\cdot\,] \mid \rho[\sigma].\mathcal{K}[\,\cdot\,] \\ \mid {} & (\nu x).(P \mid \mathcal{K}[\,\cdot\,]) \\ \mid {} & (\nu x).(\mathcal{K}[\,\cdot\,] \mid P)\end{aligned}$$

$$\textsc{red-eval-ctxt}\quad \frac{P \longrightarrow Q}{\mathcal{K}[P] \longrightarrow \mathcal{K}[Q]}$$

$$\textsc{red-congr}\quad \frac{P' \equiv P \qquad P \longrightarrow Q \qquad Q \equiv Q'}{P' \longrightarrow Q'}$$

Fig. 3. Main reduction rules for $\pi$BI.

Figure 3 gives the main reduction rules. For every "right" rule presented (denoted with the suffix '-r'), we omit the corresponding "left" rule, where the order of the parallel components interacting is reversed. The omitted rules can be found in Appendix A.

The four rules in Figure 3 describe interactions along a channel. Rule red-comm-r describes the exchange of channel $y$ along $x$. The resulting process contains an explicit restriction for $y$ with $P_2$ out of scope, reflecting the expectation that $P_1$ is the provider of the new session at $y$. Rule red-unit-r describes the closing of a session at $x$. Rule red-case shows how a branch offered on $x$ can be selected by sending inl or inr. Finally, Rule red-fwd-r explains the elimination of a forwarder connected by restriction in terms of a substitution.

The next three rules of Figure 3 define the semantics of spawn. The crucial rule is Rule red-spawn, which we explain by example.

*Example 2.2.* Consider a process $P$ that provides a session on channel $x$. Another process $Q$ provides a session on $v$ by relying *twice* on the session provided by $P$, on channels $x_1$ and $x_2$. Simple concrete examples are $P \triangleq \overline{x}\langle\rangle$ and $Q \triangleq x_1().x_2().\overline{v}\langle\rangle$. Now consider the following process:

$$R \triangleq (\nu x).(z().P \mid_x \rho[x \mapsto x_1, x_2].Q)$$

In $R$, the process $P$ is blocked waiting for the session on a channel $z$ to close. By Rule red-spawn,

$$R \longrightarrow \rho[z \mapsto z_1, z_2].(\nu x_1).(z_1().P[x_1/x] \mid_{x_1} (\nu x_2).(z_2().P[x_2/x] \mid_{x_2} Q)).$$

The result is two copies of $P$, providing their sessions on $x_1$ and $x_2$ instead of on $x$. Since we are also copying the closing prefixes on $z$, an additional spawn is generated, but now on $z$: it signals to the environment that two copies of the process providing the session on $z$ should be created and that they should provide its session on $z_1$ and $z_2$.

In the example above, the channel $z$ is a free name of the process that is copied by the spawn reduction. Generally, a copied process may rely on arbitrarily many sessions on the free names of the process, and all the processes providing these sessions will have to be copied as well. To handle the general case, Rule red-spawn uses the following definition.

*Definition 2.3 (Indexed renaming).* Given a process $P$ with $\mathrm{fn}(P) = \{a, b, \dots, z\}$, we define $P^{(i)}$ to be the process $P$ where every free name is replaced by a fresh copy of the name indexed by $i$. Formally, assuming $a_i, b_i, \dots, z_i \notin \mathrm{fn}(P)$, $P^{(i)} \triangleq P[a_i/a,\ b_i/b,\ \dots,\ z_i/z]$.

Note that Rule RED-SPAWN uniformly handles the case where a session is not used at all.

*Example 2.4.* Consider again $P$ that provides a session on $x$. This time, the process $Q'$ provides a session on $v$ *without* relying on the session provided by $P$ (e.g., simply $Q' \triangleq \overline{v}\langle\rangle$). Now consider the following process, obtained by replacing the spawn prefix and $Q$ in $R$ from Example 2.2:

$$R' \triangleq (\boldsymbol{\nu} x).(z().P \mid_x \boldsymbol{\rho}[x \mapsto \emptyset].Q')$$

By Rule RED-SPAWN, $R' \longrightarrow \boldsymbol{\rho}[z \mapsto \emptyset].Q'$. In this case, $P$ is dropped. Since the empty input prefix on $z$ is also dropped, an additional spawn is generated to signal to the environment that the process providing the session on $z$ should be dropped as well.

Rules RED-SPAWN-R and RED-SPAWN-MERGE show how the spawn prefix interacts with independent process compositions and with other spawn prefixes, respectively. Rule RED-SPAWN-R is a form of scope extrusion: spawn prefixes can "bubble up" past restrictions that do not capture their bindings, possibly enabling interactions of the spawn with processes in the outer context. Rule RED-SPAWN-MERGE describes how two consecutive spawn prefixes can be combined into a single spawn, by merging the spawn bindings, denoted $\ltimes$, as follows.

*Definition 2.5 (Merge).* Let $\sigma[X] \triangleq \bigcup\{\sigma(x) \mid x \in X, x \in \mathrm{dom}(\sigma)\}$. The *merge* of two spawn bindings $\sigma_1, \sigma_2$, written $\sigma_1 \ltimes \sigma_2$, is defined as:

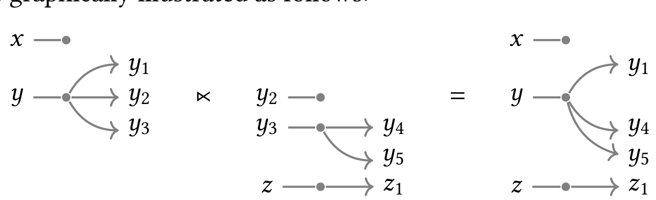
$$(\sigma_1 \ltimes \sigma_2)(x) \triangleq \begin{cases} \sigma_2[\sigma_1(x)] \cup (\sigma_1(x) \setminus \mathrm{dom}(\sigma_2)) & \text{if } x \in \mathrm{dom}(\sigma_1) \\ \sigma_2(x) & \text{if } x \notin \mathrm{dom}(\sigma_1) \wedge x \notin \mathrm{restr}(\sigma_1) \\ \bot & \text{otherwise} \end{cases}$$

Note that the merge of two independent spawn bindings is just disjoint union (as functions), and $\emptyset$ is the neutral element for $\ltimes$. Merge is associative: $(\sigma_1 \ltimes (\sigma_2 \ltimes \sigma_3)) = ((\sigma_1 \ltimes \sigma_2) \ltimes \sigma_3)$.

The idea behind the merge operation $\sigma_1 \ltimes \sigma_2$ is to "connect" the outputs of $\sigma_1$ to the inputs of $\sigma_2$, similarly to composition of relations. However, names that are irrelevant for $\sigma_1$ should still be subject to the mapping of $\sigma_2$, unless they are captured by the restrictions of $\sigma_1$. For example:

$$\begin{bmatrix} x \mapsto \emptyset \\ y \mapsto \{y_1, y_2, y_3\} \end{bmatrix} \ltimes \begin{bmatrix} y_2 \mapsto \emptyset \\ y_3 \mapsto \{y_4, y_5\} \\ z \mapsto z_1 \end{bmatrix} = \begin{bmatrix} x \mapsto \emptyset \\ y \mapsto \{y_1, y_4, y_5\} \\ z \mapsto z_1 \end{bmatrix}$$

This merge can be graphically illustrated as follows:



Note how $x$ and $z$ are both in the domain of the result, and how the mapping to $y_1$ is preserved by the merge, although it is not in the restrictions of the second binding.

The last two rules in Figure 3 are purely structural. Rule RED-EVAL-CTXT closes reduction under *evaluation contexts*, denoted $\mathcal{K}$, consisting of spawn prefixes and structured parallel compositions (cf. Figure 3). Rule RED-CONGR closes reduction under structural congruence.

## 2.3 Typing

The $\pi$BI type system is based on the BI sequent calculus, and follows the approach of $\pi$DILL: propositions are interpreted as session types, where the context governs the *use* of available channels and the conclusion governs the process' behavior on the *provided* channel. As such, the type system of $\pi$BI uses judgments of the form $\Delta \vdash P :: x : A$, where the process $P$ provides the session $A$ on channel $x$, while using the sessions provided by the typing context $\Delta$.

The top of Figure 4 gives types, bunches, and bunched contexts; we explain the session behavior associated with types when we discuss the typing rules below. Bunches $\Delta$ are binary trees with internal nodes labelled with either '**;**' or '**,**', and with leaves being either unit bunches ($\emptyset_{\mathsf{m}}$ or $\emptyset_{\mathsf{a}}$) or typing assignments ($x : A$). We write $\mathrm{fn}(\Delta)$ for the set of names occurring in the bunch $\Delta$, and write $x \in \Delta$ to denote $x \in \mathrm{fn}(\Delta)$. As is standard for BI, we consider bunches modulo the least congruence on bunches closed under commutative monoid laws for (**,**) with unit $\emptyset_{\mathsf{m}}$, and for (**;**) with unit $\emptyset_{\mathsf{a}}$, denoted $\equiv$. For example, $(\Delta_1 \mathbin{,} \emptyset_{\mathsf{m}}) \mathbin{;} \Delta_2 \equiv \Delta_2 \mathbin{;} \Delta_1$.

Bunched contexts $\Gamma(\,\cdot\,)$ are bunches with a hole $(\,\cdot\,)$. As usual, we write $\Gamma(\Delta)$ for a bunch obtained by replacing $(\,\cdot\,)$ with $\Delta$ in $\Gamma$. We write $\Gamma(\,\cdot \mid \cdots \mid \cdot\,)$ for a bunched context with multiple holes.

Figure 4 also gives the type system for $\pi$BI. We organize them in four groups: the first six rules type communication primitives with multiplicative types, and the next six rules with additive types; the following three rules type branching primitives using disjunction; the final four rules type forwarding, structured parallel composition, and the structural rules.

One key design choice of our typing rules is that the processes in the multiplicative and the additive groups of rules are the same. For example, the same send action can be typed with $A * B$ or with $A \land B$. Their difference lays purely in the way they manage their available resources, possibly enabling or restricting the use of Rule Struct in other parts of the derivation.

*Rules for multiplicative constructs.* The type $A * B$ is assigned to a session that outputs a channel of type $A$ and continues as $B$. Rule Sep-r states that to *provide* a session of type $A * B$ on $x$, a process must output on $x$ a new name $y$ and continue with a process providing a session of type $A$ on $y$ in parallel with a process providing the continuation session $B$ on $x$. Rule Sep-l describes how to *use* a session of type $A * B$ on $x$: a process must input on $x$ a new name $y$ which is to be used for the session of type $A$, after which the process must provide the continuation session $B$ on $x$.

Rules Wand-r and Wand-l describe the type $A \mathrel{-\!\!*} B$. These rules are dual to the rules for $A * B$: providing $A \mathrel{-\!\!*} B$ requires an input, and using it requires an output.

Rule Emp-r states how to close a session of type $\mathbf{1}_{\mathsf{m}}$ using an empty output, followed by termination. The dual Rule Emp-l uses the empty input prefix. Note that Rule Emp-r requires the context to be $\emptyset_{\mathsf{m}}$, effectively forcing processes to consume all the sessions they use before terminating.

*Rules for additive constructs.* As already mentioned, the rules for sessions of additive type, are identical to the ones for multiplicative types, except that the latter (de)composes bunches using '**,**' while the former uses '**;**'. In particular, the process interpretation of the rules is identical for both counterparts. The difference has effect elsewhere in the derivation, where the choice between '**;**' and '**,**' affects the possibility of using Rule Struct (explained last).

*Rules for disjunction.* Disjunction types branching constructs. To provide on $x$ a session of type $A \lor B$, the process must select either $\mathsf{inl}/\mathsf{inr}$ on $x$ and continues by providing $A/B$, respectively. Using a session of type $A \lor B$ on $x$ requires a branching on $x$, where the left branch uses $x$ as $A$ and the right branch as $B$. Curiously, there is no dual construct for disjunction in BI, meaning that there is no way to type a selection on a channel that is being used, or a branch on a channel that is being provided. There is no canonical way of adding such a dual construct; there are however

— Types, bunches, and contexts

$$\begin{array}{rcll} A,B,C & ::= & \mathbf{1}_{\mathsf{m}} \mid (A * B) \mid (A \mathrel{-\!\!*} B) & \text{(multiplicatives)} \\ & \mid & \mathbf{1}_{\mathsf{a}} \mid (A \land B) \mid (A \to B) \mid (A \lor B) & \text{(additives)} \\ \Delta, \Theta & ::= & \emptyset_{\mathsf{m}} \mid \emptyset_{\mathsf{a}} \mid x : A \mid \Delta \,,\, \Delta \mid \Delta \,;\, \Delta & \text{(bunches)} \\ \Gamma(\cdot) & ::= & (\cdot) \mid \Delta \,,\, \Gamma(\cdot) \mid \Delta \,;\, \Gamma(\cdot) \mid \Gamma(\cdot) \,,\, \Delta \mid \Gamma(\cdot) \,;\, \Delta & \text{(bunched contexts)} \end{array}$$

— Typing

SEP-R
$$\frac{\Delta_1 \vdash P_1 :: y : A \qquad \Delta_2 \vdash P_2 :: x : B}{\Delta_1 \,,\, \Delta_2 \vdash \overline{x}[y].(P_1 \mid P_2) :: x : (A * B)}$$

SEP-L
$$\frac{\Gamma(x : B \,,\, y : A) \vdash P :: z : C}{\Gamma(x : A * B) \vdash x(y).P :: z : C}$$

WAND-R
$$\frac{\Delta \,,\, y : A \vdash P :: x : B}{\Delta \vdash x(y).P :: x : A \mathrel{-\!\!*} B}$$

WAND-L
$$\frac{\Delta \vdash P :: y : A \qquad \Gamma(x : B) \vdash Q :: z : C}{\Gamma(\Delta \,,\, x : A \mathrel{-\!\!*} B) \vdash \overline{x}[y].(P \mid Q) :: z : C}$$

EMP-R
$$\emptyset_{\mathsf{m}} \vdash \overline{x}\langle\rangle :: x : \mathbf{1}_{\mathsf{m}}$$

EMP-L
$$\frac{\Gamma(\emptyset_{\mathsf{m}}) \vdash P :: x : C}{\Gamma(x : \mathbf{1}_{\mathsf{m}}) \vdash x().P :: x : C}$$

CONJ-R
$$\frac{\Delta_1 \vdash P_1 :: y : A \qquad \Delta_2 \vdash P_2 :: x : B}{\Delta_1 \,;\, \Delta_2 \vdash \overline{x}[y].(P_1 \mid P_2) :: x : A \land B}$$

CONJ-L
$$\frac{\Gamma(x : B \,;\, y : A) \vdash P :: z : C}{\Gamma(x : A \land B) \vdash x(y).P :: z : C}$$

IMPL-R
$$\frac{\Delta \,;\, y : A \vdash P :: x : B}{\Delta \vdash x(y).P :: x : A \to B}$$

IMPL-L
$$\frac{\Delta \vdash P :: y : A \qquad \Gamma(x : B) \vdash Q :: z : C}{\Gamma(\Delta \,;\, x : A \to B) \vdash \overline{x}[y].(P \mid Q) :: z : C}$$

TRUE-R
$$\emptyset_{\mathsf{a}} \vdash \overline{x}\langle\rangle :: x : \mathbf{1}_{\mathsf{a}}$$

TRUE-L
$$\frac{\Gamma(\emptyset_{\mathsf{a}}) \vdash P :: y : A}{\Gamma(x : \mathbf{1}_{\mathsf{a}}) \vdash x().P :: y : A}$$

DISJ-R-INL
$$\frac{\Delta \vdash P :: x : A}{\Delta \vdash x \triangleleft \mathsf{inl}.P :: x : A \lor B}$$

DISJ-R-INR
$$\frac{\Delta \vdash P :: x : B}{\Delta \vdash x \triangleleft \mathsf{inr}.P :: x : A \lor B}$$

DISJ-L
$$\frac{\Gamma(x : A) \vdash P :: z : C \qquad \Gamma(x : B) \vdash Q :: z : C}{\Gamma(x : A \lor B) \vdash x \triangleright \mathsf{case}(P, Q) :: z : C}$$

FWD
$$y : A \vdash [x \leftarrow y] :: x : A$$

CUT
$$\frac{\Delta \vdash P :: x : A \qquad \Gamma(x : A) \vdash Q :: z : C}{\Gamma(\Delta) \vdash (\boldsymbol{\nu} x).(P \mid Q) :: z : C}$$

STRUCT
$$\frac{\Delta_2 \vdash P :: z : C \qquad \sigma \colon \Delta_1 \rightsquigarrow \Delta_2}{\Delta_1 \vdash \boldsymbol{\rho}[\sigma].P :: z : C}$$

BUNCH-EQUIV
$$\frac{\Delta_2 \vdash P :: x : C \qquad \Delta_2 \equiv \Delta_1}{\Delta_1 \vdash P :: x : C}$$

— Spawn binding

SPAWN-CONTRACT
$$[x \mapsto \{x_1, \ldots, x_n\} \mid x \in \Delta] \colon \Gamma(\Delta) \rightsquigarrow \Gamma\big(\Delta^{(1)} \,;\, \cdots \,;\, \Delta^{(n)}\big)$$

SPAWN-WEAKEN
$$[x \mapsto \emptyset \mid x \in \Delta_1] \colon \Gamma(\Delta_1 \,;\, \Delta_2) \rightsquigarrow \Gamma(\Delta_2)$$

SPAWN-MERGE
$$\frac{\sigma_1 \colon \Delta_0 \rightsquigarrow \Delta_1 \qquad \sigma_2 \colon \Delta_1 \rightsquigarrow \Delta_2}{(\sigma_1 \ltimes \sigma_2) \colon \Delta_0 \rightsquigarrow \Delta_2}$$

Fig. 4. Types, typing rules and spawn binding rules for $\pi$BI.

$$\textsc{Weakening}\quad \frac{\Gamma(\Delta_2) \vdash P :: z : C \qquad \sigma = [x \mapsto \emptyset \mid x \in \Delta_1]}{\Gamma(\Delta_1 \,;\, \Delta_2) \vdash \boldsymbol{\rho}[\sigma].P :: z : C}$$

$$\textsc{Contraction}\quad \frac{\Gamma(\Delta^{(1)} \,;\, \Delta^{(2)}) \vdash P :: z : C \qquad \sigma = [x \mapsto x_1, x_2 \mid x \in \Delta]}{\Gamma(\Delta) \vdash \boldsymbol{\rho}[\sigma].P :: z : C}$$

Fig. 5. Usual presentations of weakening and contraction for BI sequent calculi.

extensions of BI that incorporate one—see, e.g., [Brotherston 2012; Brotherston and Calcagno 2010; Brotherston and Villard 2015; Docherty 2019; Pym 2002].

*Forwarders, Cut, and structural rules.* Rule Fwd types the forwardes $[x \leftarrow y]$ as providing a session of type $A$ on $x$ as a copycat of a session of the same type on $y$ in the context. Rule Cut connects processes $P$ and $Q$ along the channel $x$: $P$ must *provide* a session of type $A$ on $x$, whereas $Q$ must *use* the session of the same type on the same channel.

Rule Bunch-equiv closes typing under bunch equivalence. Rule Struct extends indexed renaming (Definition 2.3) to bunches as follows.

*Definition 2.6 (Indexed bunch renaming).* Let $\Delta$ be a bunch with $\mathrm{fn}(\Delta) = \{a, b, \ldots, z\}$. Assuming $a_i, b_i, \ldots, z_i \notin \mathrm{fn}(\Delta)$, we define $\Delta^{(i)} \triangleq \Delta[a_i/a,\ b_i/b,\ \ldots,\ z_i/z]$, where $\Delta\theta$ is the bunch obtained by applying the substitution $\theta$ to all the leaves of $\Delta$.

Rule Struct subsumes and generalizes the two structural rules of weakening and contraction. To unpack the meaning of the rule, Figure 5 gives rules for weakening and contraction as usually presented for BI sequent calculi. Rule Weakening discards the unused resources in $\Delta_1$. The process interpretation is a spawn that terminates the providers of sessions on channels in $\Delta_1$. Rule Contraction allows the duplication of the resources in $\Delta$. These resources need to be renamed to keep the names unique, hence the substitutions $\Delta^{(1)}$ and $\Delta^{(2)}$ in the premise. The process interpretation is again a spawn prefix that generates two indexed variants of each name in $\Delta$, representing the duplicated resources. For both rules, it is crucial that the affected bunches are combined using ‘;’.

Both Rules Weakening and Contraction transform bunches according to the spawn binding of the involved names. The idea behind Rule Struct is to generalize weakening and contraction, and allow more general spawn bindings. As such, the rule combines in a single application a number of consecutive or independent applications of Rules Weakening and Contraction.

To relate spawn bindings and their corresponding transformations of bunches, we define a *spawn binding typing* judgment $\sigma\colon \Delta_1 \rightsquigarrow \Delta_2$; the bottom of Figure 4 gives their rules.

The idea is to consider a binding $\sigma$ as the merge of a sequence of bindings $\sigma = \sigma_1 \ltimes \ldots \ltimes \sigma_k$, where each $\sigma_i$ is either a weakening or a contraction binding. The weakening and contraction bindings are typed using Rules spawn-weaken and spawn-contract. In case of contraction, when $n = 2$ we get pure contraction, when $n > 2$ it might represent a number of consecutive contractions applied to the same bunch; the corner case when $n = 1$ just renames the variables in the bunch, and might arise as the by-product of a contraction and a weakening (partially) canceling each other out.

Rules spawn-weaken and spawn-contract combined with Rule Struct offer a justification of the specialized Rules Weakening and Contraction, respectively. In the former case, the justification is direct. The latter case holds for $n = 2$, i.e., for pure contraction.

We wrap up the explanation of Rule Struct by giving an example typing derivation.

*Example 2.7.* Consider the following process, with contraction and weakening in one spawn:

$$P \triangleq (\boldsymbol{\nu}x).(z().Q \mid_x (\boldsymbol{\nu}y).(\overline{y}\langle\rangle \mid_y \boldsymbol{\rho}[x \mapsto x_1, x_2;\, y \mapsto \emptyset].R))$$

This process is well-typed, assuming $\Delta \vdash Q :: x : A$ and $\Gamma(x_1 : A\,;x_2 : A) \vdash R :: v : B$, as follows:

$$\dfrac{\dfrac{\dfrac{\Delta \vdash Q :: x : A}{\Delta\,, \emptyset_\mathrm{m} \vdash Q :: x : A}}{\Delta\,, z : \mathbf{1}_\mathrm{m} \vdash z().Q :: x : A} \quad \dfrac{\dfrac{\emptyset_\mathrm{a} \vdash \overline{y}\langle\rangle :: y : \mathbf{1}_\mathrm{a} \quad \dfrac{\Gamma(x_1 : A\,;x_2 : A) \vdash R :: v : B \quad \Psi}{\Gamma(x : A\,;y : \mathbf{1}_\mathrm{a}) \vdash \boldsymbol{\rho}[x \mapsto x_1, x_2;\, y \mapsto \emptyset].R :: v : B}}{\Gamma(x : A\,;\emptyset_\mathrm{a}) \vdash (\boldsymbol{\nu} y).(\overline{y}\langle\rangle \mid_y \boldsymbol{\rho}[x \mapsto x_1, x_2;\, y \mapsto \emptyset].R) :: v : B}}{\Gamma(x : A) \vdash (\boldsymbol{\nu} y).(\overline{y}\langle\rangle \mid_y \boldsymbol{\rho}[x \mapsto x_1, x_2;\, y \mapsto \emptyset].R) :: v : B}}{\Gamma(\Delta\,, z : \mathbf{1}_\mathrm{m}) \vdash P :: v : B}$$

where $\Psi$ is as follows:

$$\dfrac{\begin{array}{c}[x \mapsto x_1, x_2] : \Gamma(x : A\,;y : \mathbf{1}_\mathrm{a}) \rightsquigarrow \Gamma(x_1 : A\,;x_2 : A\,;y : \mathbf{1}_\mathrm{a}) \\ [y \mapsto \emptyset] : \Gamma(x_1 : A\,;x_2 : A\,;y : \mathbf{1}_\mathrm{a}) \rightsquigarrow \Gamma(x_1 : A\,;x_2 : A)\end{array}}{([x \mapsto x_1, x_2] \ltimes [y \mapsto \emptyset]) : \Gamma(x : A\,;y : \mathbf{1}_\mathrm{a}) \rightsquigarrow \Gamma(x_1 : A\,;x_2 : A)}$$

Notice how the spawn binding must be split into a contracting and a weakening spawn binding to justify the transformation of the bunch.

*Empty spawn.* We briefly discuss a corner case: according to the typing rules for spawn bindings, we can type the empty spawn $\boldsymbol{\rho}[\emptyset]$. It is tempting to add a structural congruence or reduction that removes it, since an empty spawn does not do much operationally: an empty spawn can only propagate along cuts and silently merge into other spawns. However, adding a reduction such as $\boldsymbol{\rho}[\emptyset].P \longrightarrow P$ will cause complications because the empty spawn prefix, though operationally vacuous, can influence the typing. An example is the following application of weakening:

$$\dfrac{\Gamma(\emptyset_\mathrm{a}) \vdash P :: x : A}{\Gamma(\emptyset_\mathrm{m}) \vdash \boldsymbol{\rho}[\emptyset].P :: x : A}$$

Thus, such a reduction might slightly change the typing of a process across reductions, disproving type preservation. This would unnecessarily complicate the system and, arguably, would not be in line with the Curry-Howard correspondence.

The empty spawn prefixes are but a minor annoyance: reductions can still happen behind spawn prefixes. We do have to take extra care of the empty spawn when we show deadlock-freedom in Section 3.1 and weak normalization in Section 3.2. Next, we discuss additional examples.

## 2.4 Examples and Comparisons

$\pi$BI is expressive enough to represent many useful concurrency patterns. Here we show three significant examples and contrast $\pi$BI's approach to related calculi. Below we write $P \longrightarrow^k Q$ to mean that $P$ reduces to $Q$ in $k > 1$ consecutive steps.

*Server and clients.* Recall from Example 2.2 the process $R = (\boldsymbol{\nu} x).(z().P \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q)$. We can interpret $z().P$ as a server providing a service on $x$ while relying on another server providing a service on $z$, and the spawn as a request for two copies of the server to be used in $Q$ on $x_1$ and $x_2$.

In $\pi$DILL and CP, servers and clients are expressed using replicated input $!x(y).P$, which upon receiving a channel $y$ replicates $P$ to provide its session on $y$. A client must then explicitly request a copy of the server by sending a fresh channel over $x$. The $\pi$DILL analog of $R$ would then be $R' \triangleq (\boldsymbol{\nu} u).(!u(x).\overline{z}[z'].z'().P \mid \overline{u}[x_1].\overline{u}[x_2].Q)$. In general, $\pi$DILL's servers and clients can be expressed in $\pi$BI by removing the replicated inputs (i.e. $!x(y).P$ becomes $P[x/y]$) and replacing request outputs with spawns (i.e., $\overline{x}[x_1].Q$ becomes $\boldsymbol{\rho}[x \mapsto x_1, x_2].Q[x_2/x]$).

There is a crucial difference in the two models of servers: in $\pi$DILL, the server itself is responsible for creating a new instance of the session it provides, and thus needs to make sure that the sessions on which the new instance depends are themselves provided by servers. In $\pi$BI the responsibility for duplication lies with the client; the server does not need to make special arrangements to allow for duplication, and its dependencies are duplicated on-the-fly by the spawn semantics.

The on-the-fly nature of spawn propagation makes the server/clients pattern more concurrent in $\pi$BI than in $\pi$DILL. Suppose we connect $R$ to a process providing $z$. The communication on $z$ can take place before the spawn reduction, such that the spawn no longer needs to propagate to $z$:

$$(\boldsymbol{\nu} z).(\overline{z}\langle\rangle \mid R) \longrightarrow (\boldsymbol{\nu} x).(P \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q).$$

This is not possible in $\pi$DILL: the replicated input of the server is blocking the communication on $z$.

*Failures.* An important aspect of (distributed) programming is coping with failure. For example, consider $P \triangleq x(y).x().\overline{z}[w].([w \leftarrow y] \mid z().\overline{v}\langle\rangle)$, i.e., a process that receives a channel $y$ over $x$ and forwards it over $z$. Suppose that the process providing $x$ is unreliable, and might not be able to send the channel $y$. This provider process indicates availability by a selection on $x$: left means availability and right means the converse. We can then embed $P$ in a branch on $x$, where the right branch propagates the failure to forward a channel by means of spawn: $P' \triangleq x \triangleright \mathsf{case}(P, x().\boldsymbol{\rho}[z \mapsto \emptyset].\overline{v}\langle\rangle)$. Let $z(q).R$ denote the process providing the session on $z$, which expects to receive a channel. The following is an example where the behavior on $x$ is indeed available:

$$\begin{aligned}
&(\boldsymbol{\nu} z).(z(q).R \mid (\boldsymbol{\nu} x).(x \triangleleft \mathsf{inl}.\overline{x}[u].(\overline{u}\langle\rangle \mid \overline{x}\langle\rangle) \mid P') \\
\longrightarrow^3\ &(\boldsymbol{\nu} z).(z(q).R \mid (\boldsymbol{\nu} u).(\overline{u}\langle\rangle \mid \overline{z}[w].([w \leftarrow u] \mid z().\overline{v}\langle\rangle))) \\
\longrightarrow^3\ &(\boldsymbol{\nu} u).(\overline{u}\langle\rangle \mid (\boldsymbol{\nu} z).(R[u/q] \mid z().\overline{v}\langle\rangle))
\end{aligned}$$

In contrast, in the following example the behavior on $x$ is not available:

$$(\boldsymbol{\nu} z).(z(q).R \mid (\boldsymbol{\nu} x).(x \triangleleft \mathsf{inr}.\overline{x}\langle\rangle \mid P')) \longrightarrow^2 (\boldsymbol{\nu} z).(z(q).R \mid \boldsymbol{\rho}[z \mapsto \emptyset].\overline{v}\langle\rangle) \longrightarrow \boldsymbol{\rho}[\emptyset].\overline{v}\langle\rangle$$

The principle sketched in this example is inspired by the typed framework by Caires and Pérez [2017], which supports communication primitives for non-deterministically available or unavailable behavior via a Curry-Howard interpretation of Classical LL with dedicated modalities.

*Interaction between session delegation and spawn.* Session delegation (also known as higher-order session communication) is the mechanism that enables to exchange channels themselves over channels, dynamically changing the communication topology. In $\pi$BI, delegation interacts with spawn, in that changing process connections influences the propagation of spawn. Let $P \triangleq (\boldsymbol{\nu} x).(\overline{x}[y].(\overline{y}\langle\rangle \mid \overline{x}\langle\rangle) \mid (\boldsymbol{\nu} z).(x(w).x().w().\overline{z}\langle\rangle \mid \boldsymbol{\rho}[z \mapsto \emptyset].\overline{v}\langle\rangle))$. From $P$, we could either reduce the spawn prefix or synchronize on $x$. If we first reduce the spawn, the spawn propagates to $x$:

$$P \longrightarrow (\boldsymbol{\nu} x).(\overline{x}[y].(\overline{y}\langle\rangle \mid \overline{x}\langle\rangle) \mid \boldsymbol{\rho}[x \mapsto \emptyset].\overline{v}\langle\rangle).$$

However, if we first synchronize on $x$, the spawn propagates to the delegated channel $y$:

$$P \longrightarrow^2 (\boldsymbol{\nu} y).(\overline{y}\langle\rangle \mid (\boldsymbol{\nu} z).(y().\overline{z}\langle\rangle \mid \boldsymbol{\rho}[z \mapsto \emptyset].\overline{v}\langle\rangle)) \longrightarrow (\boldsymbol{\nu} y).(\overline{y}\langle\rangle \mid \boldsymbol{\rho}[y \mapsto \emptyset].\overline{v}\langle\rangle).$$

*Incomparability with $\pi$DILL.* As shown by O'Hearn [2003], DILL and BI are incomparable. Examining two canonical distinguishing examples can shed some light on the fundamental differences of the two logics, and their interpretations as session type systems.

As we remarked in Section 1, DILL admits a "number of uses" interpretation, where linear resources have to be used exactly once. This interpretation is not supported by BI:

*Example 2.8.* In $\pi$BI it is possible to input linearly (i.e. with $-\!\!*$) a session and use it twice. The process $P \triangleq z(a).z(y).\boldsymbol{\rho}[a \mapsto a_1, a_2].\overline{y}[a'_1].\big([a'_1 \leftarrow a_1] \mid \overline{y}[a'_2].([a'_2 \leftarrow a_2] \mid [z \leftarrow y])\big)$ can be typed as providing a session $A \mathrel{-\!\!*} (A \to A \to B) \to B$ on $x$:

$$
\dfrac{\dfrac{\dfrac{a_1 : A \vdash [a_1' \leftarrow a_1] :: a_1' : A \qquad \dfrac{a_2 : A \vdash [a_2' \leftarrow a_2] :: a_2' : A \qquad y : B \vdash [z \leftarrow y] :: z : B}{a_2 : A \,;\, y : A \to B \vdash \overline{y}[a_2'].([a_2' \leftarrow a_2] \mid [z \leftarrow y]) :: z : B}}{a_1 : A \,;\, a_2 : A \,;\, y : A \to A \to B \vdash \overline{y}[a_1'].([a_1' \leftarrow a_1] \mid \overline{y}[a_2'].(\ldots)) :: z : B}}{a : A \,;\, y : A \to A \to B \vdash \rho[a \mapsto a_1, a_2].(\overline{y}[a_1'].(\ldots)) :: z : B}}{\dfrac{a : A \vdash z(y).\rho[a \mapsto a_1, a_2].(\ldots) :: z : (A \to A \to B) \to B}{\emptyset_{\mathrm{m}} \vdash z(a).z(y).\rho[a \mapsto a_1, a_2].(\ldots) :: z : A \mathrel{-\!\!*} (A \to A \to B) \to B}}
$$

The process receives a single session of type $A$ over $a$ through linear input. The session type of $y$ inputs $A$ twice, but allows these two $A$-typed sessions to share a common origin. The process can thus spawn two copies of $a : A$ and use them to interact with $y$.

The corresponding LL proposition $A \multimap (A \to A \to B) \to B$ is not derivable: as LL forbids using twice a resource obtained through linear input. However, the notion of linearity in $\pi$BI has a more subtle reading: it restricts the origin of sessions. In Example 2.8, the use of $\to$ allows the duplication of the session at $a$ into its copies $a_1$ and $a_2$; this information about the "origin" of $a_1$ and $a_2$ is recorded in the bunch by the use of (;).

On the other hand, there are types provable in DILL that are not provable in BI. A simple example is $A \multimap B \vdash A \to B$, converting an implication from linear to non-linear. A "number of uses" interpretation of the conversion makes sense: $A \multimap B$ promises to use $A$ exactly once to produce $B$; $A \to B$ declares to produce $B$ using $A$ an unspecified number of times, including exactly once. The corresponding judgment $A \mathrel{-\!\!*} B \vdash A \to B$ is not provable in BI (and thus in $\pi$BI). Intuitively, this is because $A \to B$ allows $A$ to be obtained with resources which share their origin with the resource $A \mathrel{-\!\!*} B$; however, $A \mathrel{-\!\!*} B$ can only be applied to resources that do not share its own origin.

In Section 5 we elucidate formally the difference between linear and non-linear connectives by giving a denotational semantics which allows tracking the origin of sessions.

## 3 META-THEORETICAL PROPERTIES

A distinguishing feature of the propositions-as-sessions approach is that the main meta-theoretical properties of session-typed processes (e.g., type preservation and deadlock-freedom) follow immediately from the cut elimination property in the underlying logic. In this section we show that $\pi$BI satisfies these properties, which serves to validate the appropriateness of our interpretation. We consider type preservation and deadlock-freedom, but also *weak normalization*. Appendix B gives additional properties and detailed proofs.

### 3.1 Type Preservation and Deadlock-Freedom

Essential correctness properties in session-based concurrency are that (i) processes correctly implement the sessions specified by its types (session fidelity) and (ii) there are no communication errors or mismatches (communication safety). Both these properties follow from the *type preservation* property, which ensures that typing is consistent across structural congruence and reduction.

THEOREM 3.1. *If $\Delta \vdash P :: x : C$, then $P \equiv Q$ and $P \longrightarrow Q$ imply $\Delta \vdash Q :: x : C$.*

The theorem above is a consequence of the tight correspondence between $\pi$BI and the BI proof theory, as structural congruence and reduction of typed processes correspond to proof equivalences and (principal) cut reductions in the BI sequent calculus (see Appendix B.3 for details).

Another important correctness property is *deadlock-freedom*, the guarantee that processes never get stuck waiting on pending communications. In general, deadlock-freedom holds for well-typed $\pi$BI processes where all names are bound, except for the provided name, which must be used only to close a session. Any process satisfying these typing conditions can then either reduce, or it is *inactive*: only the closing of the session on the provided name is left, possibly prefixed by an empty spawn. Because of bunches, a process with all names bound but one is typable in more ways than just under an empty typing context:

*Definition 3.2 (Empty bunch).* An *empty bunch* $\Sigma$ is a bunch such that $\mathrm{fn}(\Sigma) = \emptyset$. Equivalently, a bunch is empty if each of its leaves is $\emptyset_{\mathrm{m}}$ or $\emptyset_{\mathrm{a}}$.

Theorem 3.3 (Deadlock-freedom). *Given an empty bunch $\Sigma$, if $\Sigma \vdash P :: z : A$ with $A \in \{\mathbf{1}_{\mathrm{m}}, \mathbf{1}_{\mathrm{a}}\}$, then either (i) $P \equiv \overline{z}\langle\rangle$, or (ii) $P \equiv \boldsymbol{\rho}[\emptyset].\overline{z}\langle\rangle$, or (iii) there exists $S$ such that $P \longrightarrow S$.*

The property stated above is an important feature of $\pi$BI derived from its logical origin. The $\pi$BI interpretation of Rule Cut combines restriction and parallel, ensuring that parallel processes never share more than one channel and thus preventing processes such as $(\boldsymbol{\nu}x).(\boldsymbol{\nu}y).(y().\overline{x}\langle\rangle \mid x().\overline{y}\langle\rangle)$ where the subprocesses are stuck waiting for each other. The proof follows from a property that we call *progress*, which ensures that processes of a given syntactical shape can reduce. Although weak by itself, this property is useful in providing a reduction strategy for practical implementation of $\pi$BI. Moreover, it simplifies the proof of deadlock-freedom (given in Appendix B.4), which reduces to proving that processes typable under empty bunches are in the right syntactical shape to invoke progress.

### 3.2 Weak Normalization

We now turn our attention to proving that our typed calculus is weakly normalizing, that is, for every typeable process $P$ there exists some process $Q$ such that $P \longrightarrow^* Q \not\longrightarrow$. Apart from being of independent interest, we will use this result to show soundness of denotational semantics in Section 5. The normalization proof that we give here is of combinatorial nature. Before writing out the necessary auxiliary definitions and lemmas, we first outline the main ideas.

Suppose that we have a process $P$ typed as $\Delta \vdash P :: x : A$. What kind of reductions can this process make and can we come up with some kind of measure that would strictly decrease and disallow infinite reduction sequences? If we did not have the spawn prefix, then the answer to this problem would be simple: each reduction is an instance of communication (or a forwarder reduction), which decreases the total number of communication prefixes in the process. However, in presence of spawn, counting the total number of prefixes does not work. For example, consider the following reduction, where $\mathrm{fn}(R) = \{x, y\}$,

$$(\boldsymbol{\nu}x).\big(R \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q\big) \longrightarrow \boldsymbol{\rho}[y \mapsto y_1, y_2].(\boldsymbol{\nu}x_1).(R^{(1)} \mid (\boldsymbol{\nu}x_2).(R^{(2)} \mid Q)). \tag{1}$$

In this reduction the prefixes in the sub-process $R$ get duplicated, so the total number of prefixes increases. What has also changed is that the spawn prefix $\boldsymbol{\rho}[x \mapsto x_1, x_2]$ turned into the prefix $\boldsymbol{\rho}[y \mapsto y_1, y_2]$ with a larger scope. As a result, the communication prefixes in $Q$ went from being guarded directly by $\boldsymbol{\rho}[x \mapsto x_1, x_2]$, to being guarded by a prefix $\boldsymbol{\rho}[y \mapsto y_1, y_2]$, with the latter prefix being "smaller" in the sense that it is closer to the top-level of the process.

Furthermore, if the reduction (1) occurs in some evaluation context $\mathcal{K}$, then we can use Rules red-spawn-r and red-spawn-l to actually propagate the spawn prefix to the top-level:

$$\begin{aligned}\mathcal{K}[(\boldsymbol{\nu}x).(R \mid \boldsymbol{\rho}[x \mapsto x_1, x_2].Q)] &\longrightarrow \mathcal{K}[\boldsymbol{\rho}[y \mapsto y_1, y_2].(\boldsymbol{\nu}x_1).(R^{(1)} \mid (\boldsymbol{\nu}x_2).(R^{(2)} \mid Q))] \\ &\longrightarrow^* \boldsymbol{\rho}[y \mapsto y_1, y_2].\mathcal{K}[(\boldsymbol{\nu}x_1).(R^{(1)} \mid (\boldsymbol{\nu}x_2).(R^{(2)} \mid Q))],\end{aligned} \tag{2}$$

assuming $\mathcal{K}$ has no other spawn prefixes that would interfere with $\boldsymbol{\rho}[y \mapsto y_1, y_2]$.

Following this observation, the trick is to stratify the number of prefixes at each $\boldsymbol{\rho}$*-depth*, which is the number of spawn prefixes behind which the said prefix occurs. So, if we examine the previous reduction sequence (2) and ignore the top-level spawn prefix, the communication prefixes in $Q$ went from being at depth $n + 1$ to being at depth $n$. While the number of prefixes at depth $n$ has increased, the number of prefixes at depth $n + 1$ has decreased. This suggests that we should consider a progress measure that aggregates the number of prefixes, giving more weight to prefixes at greater $\boldsymbol{\rho}$-depths.

Our reduction strategy for weak normalization is then as follows. If a process can perform a communication reduction or a forwarder reduction, then we do exactly that reduction. If a process can only perform a reduction that involves a spawn prefix, then we (1) select (an active) spawn prefix with the least depth; (2) perform the spawn reduction; (3) propagate the newly created spawn prefix to the very top-level, merging it with other spawn prefixes along the way.

To show that this reduction strategy terminates, we adopt a measuring function that assigns to each process $P$ a finite mapping $\mu(P) : \mathbb{N} \to \mathbb{N}$ assigning to each number $n$ the number of communication prefixes at depth $n$ and above. In order to handle the special case of a top-level prefix, the measure function simply skips it, i.e. $\mu(\boldsymbol{\rho}[\sigma].P) = \mu(P)$ for a top-level $\boldsymbol{\rho}[\sigma]$. We then define an ordering $<$ on such mappings which prioritizes the number of prefixes at greater depths, and show that it is well-founded.

Then, we argue that each clause of our reduction strategy strictly decreases the measure. Since the relation $<$ is well-founded, it guarantees that our strategy terminates. If we perform a communication reduction, then the number of communication prefixes at a given depth decreases, which strictly decreases the measure. If we perform a spawn reduction, then the number of prefixes at some depth $n + 1$ might decrease, but the number of prefixes at depth $n$ might increase, because of the propagated spawn prefix. In this case, we keep propagating the spawn prefix to the top-level as much as possible, either leaving it at the top-level (to be skipped by the measure function), or merging it with an existing top-level prefix. In both cases, the maximal prefix depth of the process decreases, which results in a strictly decreased measure.

Due to space limitations, we refer the interested reader to Appendix B.5 for the full details.

Theorem 3.4. *If $\Delta \vdash P :: z : A$ is a typed process, then $P$ is weakly normalizable.*

## 4 TRANSLATING THE $\alpha\lambda$-CALCULUS INTO $\pi$BI

The $\alpha\lambda$-calculus is a functional calculus that is in a Curry-Howard correspondence with the natural deduction representation of BI [O'Hearn 2003; Pym 2002]. Here we develop a type-preserving translation from the $\alpha\lambda$-calculus to $\pi$BI, and establish its correctness in a very strong sense: the translation satisfies an *operational correspondence* property, which asserts how reduction steps in the source and target calculi are preserved and reflected (cf. Theorems 4.3 and 4.5, respectively).

### 4.1 The $\alpha\lambda$-calculus and its Translation into $\pi$BI

We first recall the statics and dynamics of the $\alpha\lambda$-calculus. Our formulation of the type system is based on the presentations by O'Hearn [2003] and Pym [2002, Chapter 2].

We use $M, N, L, \ldots$ for terms, and $a, b, c, \ldots, x, y, z, \ldots$ for variables. The $\alpha\lambda$-calculus is based on the $\lambda$-calculus, but with two separate kinds of function binders: $\lambda x.\, M$ with its corresponding function application $M\ N$ for the magic wand $A \mathrel{-\!\!*} B$, and $\alpha x.\, M$ with its corresponding function application $M@N$ for the intuitionistic implication $A \to B$. Selected typing rules are given in the top of Figure 6; the full type system can be found in Appendix C.

We write $\mathrm{fv}(M)$ to denote the free variables of $M$. As usual, substitution of a term $N$ for a variable $x$ in a term $M$ is denoted $M[N/x]$. We write $M[N_1/x_1,\ \ldots,\ N_n/x_n]$ for the sequence of substitutions $M[N_1/x_1] \ldots [N_n/x_n]$. The reduction semantics of the $\alpha\lambda$-calculus, denoted $\rightarrowtail$,

— Type system —

$$\text{N-ID}\quad x : A \vdash x : A \qquad \text{N-W}\ \frac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta\,;\Delta') \vdash M : A} \qquad \text{N-C}\ \frac{\Gamma(\Delta^{(1)}\,;\Delta) \vdash M : A}{\Gamma(\Delta) \vdash M[\mathrm{fn}(\Delta)/\mathrm{fn}(\Delta^{(1)})] : A} \qquad {-\!\!*}\mathrm{I}\ \frac{\Delta\,, x : A \vdash M : B}{\Delta \vdash \lambda x.\, M : A \mathrel{-\!\!*} B}$$

$$\to\mathrm{I}\ \frac{\Delta\,; x : A \vdash M : B}{\Delta \vdash \alpha x.\, M : A \to B} \qquad {-\!\!*}\mathrm{E}\ \frac{\Delta_1 \vdash M : A \mathrel{-\!\!*} B \qquad \Delta_2 \vdash N : A}{\Delta_1\,, \Delta_2 \vdash M\ N : B} \qquad \to\mathrm{E}\ \frac{\Delta_1 \vdash M : A \to B \qquad \Delta_2 \vdash N : A}{\Delta_1\,; \Delta_2 \vdash M@N : B}$$

— Reduction rules —

$$\text{RED-BETA-}\lambda\quad (\lambda x.\, M)\ N \rightarrowtail M[N/x] \qquad \text{RED-BETA-}\alpha\quad (\alpha x.\, M)@N \rightarrowtail M[N/x] \qquad \frac{M \rightarrowtail M'}{M\ N \rightarrowtail M'\ N} \qquad \frac{M \rightarrowtail M'}{M@N \rightarrowtail M'@N}$$

Fig. 6. Selected rules of the type system and reduction rules for the $\alpha\lambda$-calculus.

| $\alpha\lambda$-calculus typing of $M_0$ | $\pi$BI encoding $\mathcal{T}_z(M_0)$ |
|---|---|
| $x : A \vdash x : A$ | $x : A \vdash [z \leftarrow x] :: z : A$ |
| $\dfrac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta\,;\Delta') \vdash M : A}$ | $\dfrac{\Gamma(\Delta) \vdash \mathcal{T}_z(M) :: z : A}{\Gamma(\Delta\,;\Delta') \vdash \rho[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].\mathcal{T}_z(M) :: z : A}$ |
| $\dfrac{\Delta\,, x : A \vdash M : B}{\Delta \vdash \lambda x.\, M : A \mathrel{-\!\!*} B}$ | $\dfrac{\Delta\,, x : A \vdash \mathcal{T}_z(M) :: z : B}{\Delta \vdash z(x).\mathcal{T}_z(M) :: z : A \mathrel{-\!\!*} B}$ |
| $\dfrac{\Delta\,; x : A \vdash M : B}{\Delta \vdash \alpha x.\, M : A \to B}$ | $\dfrac{\Delta\,; x : A \vdash \mathcal{T}_z(M) :: z : B}{\Delta \vdash z(x).\mathcal{T}_z(M) :: z : A \to B}$ |
| $\dfrac{\Delta_1 \vdash M : A \mathrel{-\!\!*} B \qquad \Delta_2 \vdash N : A}{\Delta_1\,, \Delta_2 \vdash M\ N : B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_x(M) :: x : A \mathrel{-\!\!*} B \qquad \dfrac{\Delta_2 \vdash \mathcal{T}_y(N) :: y : A \qquad x : B \vdash [z \leftarrow x] :: z : B}{x : A \mathrel{-\!\!*} B\,, \Delta_2 \vdash \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x]) :: z : B}}{\Delta_1\,, \Delta_2 \vdash (\nu x).(\mathcal{T}_x(M) \mid \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x])) :: z : B}$ |
| $\dfrac{\Delta_1 \vdash M : A \to B \qquad \Delta_2 \vdash N : A}{\Delta_1\,; \Delta_2 \vdash M@N : B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_x(M) :: x : A \to B \qquad \dfrac{\Delta_2 \vdash \mathcal{T}_y(N) :: y : A \qquad x : B \vdash [z \leftarrow x] :: z : B}{x : A \to B\,; \Delta_2 \vdash \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x]) :: z : B}}{\Delta_1\,; \Delta_2 \vdash (\nu x).(\mathcal{T}_x(M) \mid \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x])) :: z : B}$ |

Fig. 7. Translation from $\alpha\lambda$-calculus to $\pi$BI (selected clauses).

follows a call-by-name strategy for the $\lambda$-calculus, extended to cover two kinds of function binders. Selected reduction rules are given in the bottom of Figure 6.

*Typed translation.* Given a typed term $\Gamma \vdash M : A$ and a variable $z \notin \mathrm{fv}(M)$, we inductively translate the typing derivation of $M$ to a $\pi$BI typing derivation, denoted $\Gamma \vdash \mathcal{T}_z(\Gamma \vdash M : A) :: z : A$. As customary in translations of $\lambda$ into $\pi$ (cf. [Milner 1992; Sangiorgi and Walker 2003; Wadler 2014]), the parameter $z$ is a name on which the behavior of the source term $M$ is made available. By abuse of notation, we often write $\Gamma \vdash \mathcal{T}_z(M) :: z : A$. The translation is inspired by a canonical translation of proofs in natural deduction from into sequent calculus from (cf. [Pym 2002, Section 6.3]), and it is type-preserving by construction. The translations of selected rules from Figure 6 is given in Figure 7. The identity derivation is translated into a forwarder, and the introduction rules are translated using right rules for the associated connectives. The elimination rules are translated using the corresponding left rule in combination with a cut. The weakening and contraction rules, which use implicit substitutions in $\alpha\lambda$-calculus, are translated explicitly using the STRUCT rule.

*Example 4.1.* Consider the following $\alpha\lambda$-calculus derivation for the term $M \triangleq \lambda a.\, \alpha y.\, (y@a)@a$:

$$\dfrac{\dfrac{\dfrac{\dfrac{a_1 : A \vdash a_1 : A \qquad \dfrac{a_2 : A \vdash a_2 : A \qquad y : A \to A \to B \vdash y : A \to A \to B}{a_2 : A\,;\, y : A \to A \to B \vdash y@a_2 : A \to B}}{a_1 : A\,;\, a_2 : A\,;\, y : A \to A \to B \vdash (y@a_2)@a_1 : B}}{a : A\,;\, y : A \to A \to B \vdash (y@a)@a : B}}{a : A \vdash \alpha y.\, (y@a)@a : (A \to A \to B) \to B}}{\emptyset_{\mathrm{m}} \vdash M \triangleq \lambda a.\, \alpha y.\, (y@a)@a : A \mathrel{-\!\!*} (A \to A \to B) \to B}$$

The translation of $M$ into $\pi$BI is

$$\begin{aligned}\mathcal{T}_z(M) = z(a).z(y).\boldsymbol{\rho}[a \mapsto a_1, a_2].(\boldsymbol{\nu} x).((\boldsymbol{\nu} w).([w \leftarrow y] \mid \overline{w}[a_2'].([a_2' \leftarrow a_2] \mid [x \leftarrow w]))\\ \mid \overline{x}[a_1'].([a_1' \leftarrow a_1] \mid [z \leftarrow x])).\end{aligned}$$

This corresponds to the $\pi$BI derivation in Example 2.8, modulo additional cuts on forwarders due to the translation of variables and function applications.

### 4.2 Operational Correspondence

Here we show that the translation $\mathcal{T}_z(-)$ preserves and reflects behavior of processes and terms. We formulate this important property in terms of an *operational correspondence* result, following established criteria (cf. [Gorla 2010; Peters 2019]). Concretely, we establish the result in two parts: *completeness* and *soundness*. The former states that reduction of $\alpha\lambda$-calculus terms induces corresponding reductions of their process translations into $\pi$BI; conversely, the latter states that reductions of translated terms are reflected by corresponding reductions of the source terms in the $\alpha\lambda$-calculus. Appendix C gives detailed proofs.

*4.2.1 Completeness.* For completeness, we want to mimic every $\alpha\lambda$-calculus reduction with one or multiple $\pi$BI reductions. That is, we would like to show that the translation induces a simulation. To accurately characterize this, we need to address the discrepancy between the way the substitutions and function application are handled in $\alpha\lambda$-calculus and in $\pi$BI. Unfortunately, the reductions of the translated term (a $\pi$BI process) might diverge from the source term, due to the way the substitution and function application are handled in the $\alpha\lambda$-calculus. A function application $(\alpha x.\, M)\; N$ results in a term $M[N/x]$ with a substitution. If there are multiple occurrences of $x$ in $M$—which is possible due to contraction—, they all get substituted with $N$. On the $\pi$BI side, substitution is represented as a composition $(\boldsymbol{\nu} x).(\mathcal{T}_x(N) \mid \mathcal{T}_z(M))$, in which *one copy* of $\mathcal{T}_x(N)$ gets connected with the body $\mathcal{T}_z(M)$ through the endpoint $x$. The contraction of the multiple occurrences of $x$ in $M$ is handled with a spawn prefix in $\mathcal{T}_z(M)$. To address this discrepancy, we formulate completeness in a generalized way: following the approach by Toninho et al. [2012], we define a *substitution lifting* relation which we show to be a simulation.

*Definition 4.2 (Substitution lifting).* Given a term $M$ and a process $P$ of the same typing, we say $P$ lifts the substitutions of $M$, denoted $\Delta \vdash P \triangleright M :: z : A$, or $P \triangleright M$ for short, if:

(1) $P \equiv \boldsymbol{\rho}[\sigma_s]. \ldots .\boldsymbol{\rho}[\sigma_1].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid \mathcal{T}_z(M')) \ldots)$ where for each $i \in [1, s]$, $\sigma_i = [y_1 \mapsto \emptyset;\ \ldots;\ y_m \mapsto \emptyset]$ (only weakening) or $\sigma_i = [y_1 \mapsto y_1, y_1';\ \ldots;\ y_m \mapsto y_m, y_m']$ (only contraction);
(2) $M = M'[N_1/x_1,\ \ldots,\ N_n/x_n][\tilde{\sigma}_1, \ldots, \tilde{\sigma}_s]$ where for each $i \in [1, s]$, the substitution $\tilde{\sigma}_i$ denotes a substitution corresponding to the spawn binding $\sigma_i$. Specifically, $\tilde{\sigma}_i$ is an empty substitution if $\sigma_i$ is weakening, and is the substitution $[y_1/y_1',\ \ldots,\ y_m/y_m']$ if $\sigma_i$ is contraction.

That is, both $P$ and $M$ are composed of $n$ cuts with (the translations of) the terms $M, N_1, \ldots, N_n$. Note that for any well-typed $N$ we have $\mathcal{T}_z(N) \triangleright N$.

We then show the completeness result.

Theorem 4.3 (Completeness). *Given $\Delta \vdash M : A$ and $\Delta \vdash P :: z : A$ such that $P \triangleright M$, if $M \rightarrowtail N$, then there exists $Q$ such that $P \longrightarrow^* Q \triangleright N$.*

*4.2.2 Soundness.* The completeness theorem shows that the reductions of terms are preserved by the translation. We now show that reductions of translated processes are reflected by reductions of source terms. There is a caveat, though: the translated processes are "more concurrent", and have more possible reductions that cannot be immediately matched in source terms.

*Example 4.4.* For some term $M[N/x]$ and a corresponding substitution-lifted process $P \triangleq (\nu x).(\mathcal{T}_x(N) \mid \mathcal{T}_z(M))$, suppose that the subterm $N$ has a reduction $N \rightarrowtail N'$. The process $P$ can mimic this reduction:

$$P \longrightarrow (\nu x).(Q \mid \mathcal{T}_z(M)),$$

for some $Q$. However, we do not necessarily have a corresponding reduction $M[N/x] \rightarrowtail M[N'/x]$, since the variable $x$ might occur at a position where it is not enabled (e.g., under a $\lambda$-binder).

In order to be able to reflect all the reductions in translated processes, we state soundness in terms of an extended class of reductions for terms, denoted $\hookrightarrow$ (with reflexive, transitive closure denoted $\hookrightarrow^*$). To be precise, let $\mathfrak{C}$ be an arbitrary $\alpha\lambda$-calculus context. In addition to the reductions in Figure 6, we consider reductions under arbitrary contexts:

$$\frac{M \hookrightarrow M'}{\mathfrak{C}[M] \hookrightarrow \mathfrak{C}[M']}$$

Theorem 4.5 (Soundness). *Given $\Delta \vdash P \triangleright M :: z : A$, if $P \longrightarrow^* Q$, then there exist $N$ and $R$ such that $M \hookrightarrow^* N$ and $Q \longrightarrow^* R \triangleright N$.*

Note that the premise in the theorem above permits arbitrarily many reduction steps from $P$ to $Q$ (i.e., $P \longrightarrow^* Q$), assuring that *every* sequence of reductions of $P$ is reflected by a corresponding sequence of reductions of the source term $M$. The alternative with a single reduction in the premise (i.e., $P \longrightarrow Q$) being a much weaker property. The proof of soundness proceeds by cases on the possible reductions of $P$, informed by the structure and typing of the source term $M$. The key point in the proof is to postpone certain independent reductions of the target process, which cannot be immediately matched by reductions in the source term.

## 5 OBSERVATIONAL EQUIVALENCE AND DENOTATIONAL SEMANTICS

Here we develop the theory of observational equivalence for $\pi$BI processes. To this end, we first define barbed equivalence and observational equivalence. Then, we provide a denotational semantics and show that processes that have the same denotation are observationally equivalent.

We first define *barbs*—observations that we can make on processes. Their formulation is standard:

$$\alpha ::= x \triangleleft \mathsf{inl} \mid x \triangleleft \mathsf{inr} \mid x \triangleright \mathsf{inl} \mid x \triangleright \mathsf{inr} \mid x \mid \overline{x} \mid x() \mid \overline{x}\langle\rangle$$

By $\mathrm{chan}(\alpha)$ we denote the channel associated to the barb $\alpha$. We say that process $P$ has a barb $\alpha$, if the relation $P{\downarrow_\alpha}$ is derivable from the rules in Figure 8. Now we define observational equivalence.

*Definition 5.1 (Barbed equivalence).* Barbed equivalence is the largest equivalence relation $\simeq_b$ on processes of the same type that is closed under reductions and that satisfies the following condition. If $P \simeq_b Q$ and $P \downarrow_\alpha$ then there exists $Q'$ such that $Q \longrightarrow^* Q' \downarrow_\alpha$.

$$\textsc{barb-select}\ \frac{\ell \in \{\mathsf{inl}, \mathsf{inr}\}}{x \triangleleft \ell.P{\downarrow_{x\triangleleft\ell}}} \qquad \textsc{barb-branch}\ \frac{\ell \in \{\mathsf{inl}, \mathsf{inr}\}}{x \triangleright \mathsf{case}(Q_{\mathsf{inl}}, Q_{\mathsf{inr}}){\downarrow_{x\triangleright\ell}}} \qquad \textsc{barb-send}\ \overline{x}[y].(P \mid Q){\downarrow_{\overline{x}}} \qquad \textsc{barb-recv}\ x(y).Q{\downarrow_x} \qquad \textsc{barb-wait}\ x().P{\downarrow_{x()}}$$

$$\textsc{barb-close}\ \overline{x}\langle\rangle{\downarrow_{\overline{x}\langle\rangle}} \qquad \textsc{barb-restr-l}\ \frac{P{\downarrow_\alpha} \quad x \neq \mathrm{chan}(\alpha)}{(\boldsymbol{\nu} x).(P \mid Q){\downarrow_\alpha}} \qquad \textsc{barb-restr-r}\ \frac{Q{\downarrow_\alpha} \quad x \neq \mathrm{chan}(\alpha)}{(\boldsymbol{\nu} x).(P \mid Q){\downarrow_\alpha}} \qquad \textsc{barb-spawn}\ \frac{P{\downarrow_\alpha} \quad \mathrm{chan}(\alpha) \notin \mathrm{restr}(\sigma)}{\boldsymbol{\rho}[\sigma].P{\downarrow_\alpha}}$$

Fig. 8. Barbs for $\pi$BI processes.

We will be mainly concerned with barbed equivalence of *closed processes*. A closed process is a process $P$ that is typeable as $\Sigma \vdash P :: y : B$, where $\mathrm{fn}(\Sigma) = \emptyset$, i.e. $\Sigma$ is an empty bunch (c.f. Definition 5.3). Note that a closed process can only have barbs associated to its provided channel $y$.

A program context $C[\,\cdot\,]$ is a $\pi$BI process with a hole in it. Given $\Delta \vdash P :: x : A$, a *closing program context* $C$ is a program context such that $\Sigma \vdash C[P] :: y : B$ for some empty bunch $\Sigma$.

*Definition 5.2 (Observational equivalence).* Two processes $\Delta \vdash P :: T$ and $\Delta \vdash Q :: T$ are observationally equivalent, denoted $\Delta \vdash P \simeq_o Q :: T$, if, for any closing program context $C$ and any type $A$ such that $\Sigma \vdash C[P] :: z : A$ and $\Sigma \vdash C[Q] :: z : A$, it is the case that $C[P]$ and $C[Q]$ are barbed equivalent.

Observational equivalence is a strong notion, because it relates two processes in *any* well-typed program context. As a consequence, proving observational equivalence of two processes directly is challenging, as it requires reasoning about an arbitrary context $C$. Observational equivalence is usually established using more local methods (i.e. methods that do not involve reasoning about a program in a context), such as bisimulations [Kouzapas et al. 2011] or logical relations [Caires et al. 2013; Derakhshan et al. 2021; Pérez et al. 2014]. Next, we describe another approach, based on denotational semantics of $\pi$BI.

## 5.1 Denotational Semantics

Our motivation for developing a denotational semantics for $\pi$BI is two-fold. First, it will provide a sound technique for establishing observational equivalence. Second, it will prove useful to illustrate the aspects of separation and sharing through tracking of the origins of different processes, thus explaining the fundamental differences between multiplicative and additive connectives in $\pi$BI. Intuitively, if we have a closed process of the type $\Sigma \vdash P :: x : A * B$, then this process outputs a fresh channel $y$ on $x$, and then separates into two processes providing $A$ and $B$. These two resulting processes will have a *different origin*: they are not results of duplication via a spawn prefix. Crucially, the two processes obtained from the prefix $\boldsymbol{\rho}[x \mapsto x_1, x_2]$ on channels $x_1$ and $x_2$ do have the same origin as the process on channel $x$.

In order to make this insight precise, we extend the type system with *atomic types*, denoted $\mathfrak{a}_1, \mathfrak{a}_2, \ldots$, which we use to represent abstract channels/resources. The extension is conservative, as we do not introduce any rules for atomic types. Also, we slightly modify our notion of empty bunches (c.f. Definition 5.3) to allow names as long as they are associated with atomic types.

*Definition 5.3 (Atomic bunch).* An *atomic bunch* $\Sigma$ is a bunch such that any type assignment $x : A$ in $\Sigma$ is such that $A$ is an atomic type.

This way, e.g., we consider $\Sigma = (x : \mathfrak{a}_1 \,;\, y : \mathfrak{a}_2)\,,\, z : \mathfrak{a}_3$ an atomic bunch. Since we do not add any typing rules for atomic processes, well-typed processes cannot "break down" sessions $\mathfrak{a}_i$ and cannot communicate/block on the channels associated with atomic types. In fact, the only thing that a well-typed process can do with a channel associated to an atomic type is forwarding. Hence, this extension retains the essential properties of the system (e.g., Theorem 3.3).

We now define the denotational semantics of types and processes. We start with a fixed set Tag of primitive tags. A tag represents an origin of a process, as in, e.g., the ID of a node in which a particular process is executed. As such, these tags will represent different origins/provenances of processes. We interpret every type as a $\wp(\mathsf{Tag})$-valued set: $[\![A]\!] : \wp(\mathsf{Tag}) \to \mathsf{Set}$. For an atomic type $\mathfrak{a}$ we set $[\![\mathfrak{a}]\!](D) = D$. We further have:

$$[\![1_m]\!](D) = [\![1_a]\!](D) = \{\epsilon\} \qquad [\![A \vee B]\!](D) = [\![A]\!](D) + [\![B]\!](D)$$
$$[\![A \wedge B]\!](D) = [\![A]\!](D) \times [\![B]\!](D) \qquad [\![A * B]\!](D) = \Sigma_{D=D_1 \cup D_2, D_1 \perp D_2}. [\![A]\!](D_1) \times [\![B]\!](D_2)$$
$$[\![A \to B]\!](D) = [\![A]\!](D) \to [\![B]\!](D) \qquad [\![A \mathrel{-\!\!*} B]\!](D) = \Pi_{D' \perp D}. [\![A]\!](D') \to [\![B]\!](D \cup D')$$

Where $D' \perp D$ denotes $D' \cap D = \emptyset$. The interpretation of bunches $[\![\Delta]\!]$ is defined standard.

The interpretation of processes is straightforward, and follows the structure of typing derivations. A process is interpreted as a function polymorphic in a set of tags $D$:

$$[\![\Delta \vdash P :: x : A]\!]_D : [\![\Delta]\!](D) \to [\![A]\!](D).$$

For space reasons, we omit the details of the interpretation, but note that it follows the usual interpretation of BI in doubly closed categories [Pym 2002, Chapter 3.3]. Specifically, we interpret types as presheaves $\mathsf{Set}^{\wp(\mathsf{Tag})}$, where $\wp(\mathsf{Tag})$ is interpreted as a discrete category. The interpretation of type formers corresponds to the Cartesian closed structure and a closed monoidal structure on $\mathsf{Set}^{\wp(\mathsf{Tag})}$.

We sometimes write $[\![P]\!]$ for $[\![\Delta \vdash P :: x : A]\!]$ when $\Delta$ and $x : A$ are unambiguous. Our interpretation is indeed a valid model of $\pi$BI, as it satisfies the following lemmas:

Lemma 5.4. *Let $\Delta \vdash P :: z : A$ and $\Delta \vdash Q :: z : A$ be processes such that $P \equiv Q$. Then $[\![P]\!] = [\![Q]\!]$.*

Lemma 5.5. *Let $\Delta \vdash P :: z : A$ and $\Delta \vdash Q :: z : A$ be processes such that $P \longrightarrow Q$. Then $[\![P]\!] = [\![Q]\!]$.*

The denotational semantics explains the "provenance tracking" aspect of the system. Specifically, we have the following result, immediate from the definition of the denotational semantics. Let $\mathfrak{a}_1, \mathfrak{a}_2$ be atomic types, and let $\Delta \vdash P :: x : \mathfrak{a}_1 * \mathfrak{a}_2$, then for any $D \in \wp(\mathsf{Tag})$, $v \in [\![\Delta]\!](D)$:

$$[\![P]\!]_D(v) = (t_1, t_2) \qquad \text{and} \qquad t_1 \neq t_2.$$

In other words, the two sessions $\mathfrak{a}_1$ and $\mathfrak{a}_2$ that are sent over the channel $x$ by the process $P$ have a *disjoint provenance*. Note that this would not hold for a process typed $\Delta \vdash P :: x : \mathfrak{a}_1 \wedge \mathfrak{a}_2$. For example, for a process $P$ of the form $y : \mathfrak{a} \vdash \boldsymbol{\rho}[y \mapsto y_1, y_2].\overline{x}\langle y_1\rangle.[x \leftarrow y_2] :: x : \mathfrak{a} \wedge \mathfrak{a}$, then we have $[\![P]\!](t) = (t, t)$, i.e., the sessions that are sent over the channel $x$ by $P$ do share their origin.

Furthermore, since the denotational semantics is fundamentally compositional, we can generalize the argument above. Suppose we place $P$ in some enclosing context $C$ such that $y : \mathfrak{a}'_1 \,, z : \mathfrak{a}'_2 \vdash C[P] :: x : \mathfrak{a}_1 * \mathfrak{a}_2$. Then $[\![C[P]]\!]_{\{t_1,t_2\}}(t_1, t_2) = (t'_1, t'_2)$ and either $t_1 = t'_1, t_2 = t'_2$ or $t_1 = t'_2, t_2 = t'_1$, as these are the only available functions in the denotational semantics. That is, the process $C[P]$ sends over $x$ the sessions either with the same origin as $y$ and $z$, or with swapped origin. If we instead consider a program with additive conjunction $y : \mathfrak{a}'_1 \,; z : \mathfrak{a}'_2 \vdash C[P] :: x : \mathfrak{a}_1 \wedge \mathfrak{a}_2$, then it may further send over $x$ two sessions both with the same origin either as in $y$ or as in $z$.

*Denotational semantics and observational equivalence.* As already mentioned, we will use denotational semantics to verify observational equivalences of processes. Formally, we have:

Theorem 5.6. *Given two processes $\Delta \vdash P :: z : C$ and $\Delta \vdash Q :: z : C$, if $[\![P]\!] = [\![Q]\!]$, then $\Delta \vdash P \simeq_o Q :: z : C$.*

In order to prove this theorem we will need the following two lemmas:

Lemma 5.7. *Suppose given a process $P$ such that $\Gamma \vdash P :: z : C$, where $P \not\longrightarrow$ and $P$ does not have any barbs on channels from $\Gamma$. Then $P$ has a barb on the channel $z$.*

Lemma 5.8 (Observability). *Suppose $\Sigma \vdash P :: z : C$ such that $\Sigma$ is an atomic bunch. Then there exists a process $Q$ such that $P \longrightarrow^* Q$ where $Q \downarrow_{\alpha(z)}$.*

Proof. By weak normalization and subject reduction (Theorems 3.1 and 3.4), and Lemma 5.7. □

We can use the observability lemma to show Theorem 5.6:

Proof (of Theorem 5.6). Let $C$ be a closing program context. We are to show $\Sigma \vdash C[P] \simeq_b C[Q] :: z : D$ for some context $C$.

Since the denotational semantics is compositional, we have $[\![C[P]]\!] = [\![C[Q]]\!]$. Furthermore, the relation $P, Q \mapsto [\![C[P]]\!] = [\![C[Q]]\!]$ is an equivalence relation and is closed under reductions (Lemma 5.5). Therefore, it suffices to consider only the main clause of barbed equivalence. That is, if $[\![P]\!] = [\![Q]\!]$ and $P \downarrow_{\alpha(z)}$, then there exists $Q'$ such that $Q \longrightarrow^* Q' \downarrow_{\alpha(z)}$.

For simplicity, let us consider a case where the type $D$, on which we are making observations, is of the form $A \vee B$. Then the only possible observations for $P$ and for $Q$ are $z \triangleleft \mathsf{inl}$ and $z \triangleleft \mathsf{inr}$. Suppose, without loss of generality, that $P \downarrow_{z \triangleleft \mathsf{inl}}$. Then, by Lemma 5.8, we have $Q \longrightarrow^* Q' \downarrow_{\alpha(z)}$.

By Lemma 5.5, we have $[\![P]\!] = [\![Q]\!] = [\![Q']\!]$. Because $P$ has a barb $z \triangleleft \mathsf{inl}$, the interpretation $[\![P]\!]$ must be of the form $\mathsf{inl} \circ (\dots)$, where $\mathsf{inl}$ is the embedding $[\![A]\!] \to [\![A]\!] + [\![B]\!]$. It the must be the case that $[\![Q']\!]$ is also of the same shape, and, hence, $Q' \downarrow_{z \triangleleft \mathsf{inl}}$. □

*Equivalence induced from the translation of the $\alpha\lambda$-calculus.* We close this section by demonstrating an application of denotational semantics in the context of the correctness of the translation from Section 4. Specifically, we show that the relation $\triangleright$ from Section 4 decomposes as a translation $\mathcal{T}_z(-)$ and an observational equivalence $\simeq_o$:

Theorem 5.9. *If $\Delta \vdash P \triangleright M :: z : A$, then $[\![P]\!] = [\![\mathcal{T}_z(M)]\!]$, and, consequently, $P \simeq_o \mathcal{T}_z(M)$.*

Proof. Essentially, we need to show that for any $\Gamma(x : A) \vdash M : B$ and $\Delta \vdash N : A$, we have

$$[\![\mathcal{T}_z(M[N/x])]\!] = [\![(\nu x).(\mathcal{T}_x(N) \mid \mathcal{T}_z(M))]\!].$$

We do this by induction on the typing derivation, generalizing to multiple substitutions. □

Recall that in Section 4 we could not use the translation function $\mathcal{T}_z(-)$ itself to establish a simulation; instead we had to take a coarser relation $\triangleright$. Theorem 5.9 shows that this does not introduce any observable difference.

## 6 RELATED WORK

We have already discussed some of the most closely related works, and we have given some comparisons with previous works by means of examples in Section 2.4. Here we discuss other related literature along several dimensions.

*BI and process calculi.* To our knowledge, the work of Anderson and Pym [2016] is the only prior work that connects BI with process calculi. Their technical approach and results are very different from ours. They introduce a process calculus (a synchronous CCS) with an explicit representation of (bunched) resources, in which processes and resources evolve hand-in-hand. Rather than a typed framework for processes or an interpretation in the style of propositions-as-types, they use a logic related to BI to specify rich properties of processes, in the style of Hennessy-Milner logic.

*BI and Curry-Howard correspondence.* The works of O'Hearn [2003] and Pym [2002] are, to our knowledge, the only prior investigations into (non-concurrent) Curry-Howard correspondences based on BI. These works were later extended to cover polymorphism [Collinson et al. 2008] and store with strong update [Berdine and O'Hearn 2006]. An extension $\lambda_{\mathsf{sep}}$ of an affine version of the $\alpha\lambda$-calculus with a more fine-grained notion of separation was studied by Atkey [2004, 2006].

*Previous works on propositions-as-sessions.* Starting with the works by Caires and Pfenning [2010] and Wadler [2012], the line of work on propositions-as-sessions has exclusively relied on (variants of) LL, which is incomparable to BI; this immediately separates those prior works from our novel approach based on BI.

Our work adapts to the BI setting key design principles in [Caires and Pfenning 2010; Wadler 2012]: the interpretation of multiplicative conjunction as output, linear implication as input, and the interpretation of 'cut' as the coalescing of restriction and parallel composition. Those works use input-guarded replication to accommodate non-linear sessions, typed with the modality $!A$; in contrast, $\pi$BI handles structural principles directly at the process level with the new spawn prefix.

Our adaptation is novel and non-trivial, and cannot be derived from prior interpretations based on LL. Still, certain aspects of $\pi$BI bear high-level similarities with elements from those interpretations. The semantics of our spawn prefix borrows inspiration from the treatment of aliases in Pruiksma and Pfenning's interpretation of asynchronous binary sessions, based on adjoint logic, in which structural rules are controlled via modalities [Pruiksma and Pfenning 2021]. Thanks to spawn binders (Definition 2.1), our semantics explicitly handles duplication and disposal of services; this is similar in spirit to the syntax and semantics of replicated servers in HCP, an interpretation based on a hypersequent presentation of classical LL [Kokke et al. 2019]. The behavioral theory of HCP consists of a labeled transition semantics for processes, a denotational semantics for processes, and a full abstraction result. The work of Qian et al. [2021] extends linear logic with coexponentials with the aim of capturing client-server interactions not expressible in preceding interpretations of linear logic. Precise comparisons between the expressivity of such interactions and the connection patterns enabled by our spawn prefix remains to be determined. Concerning failures, as discussed in Section 2.4, the work of Fowler et al. [2019] develops a linear functional language with asynchronous communication and support for failure handling, closely related to Wadler's CP.

## 7 CONCLUDING REMARKS AND FUTURE PERSPECTIVES

In this paper we present a fresh look at logical foundations for message-passing concurrency. We have cast the essential principles of propositions-as-sessions, initially developed upon LL, in the unexplored context of BI. We introduced the typed process calculus $\pi$BI, explored its operational and type-theoretical contents, illustrated its expressiveness, and established the meta-theoretical framework needed to study the behavioral consequences of the BI typing discipline for concurrency.

Our results unlock a number of enticing future directions. First, because $\pi$BI targets binary session types (between two parties) with synchronous communication, it would be interesting to study variants of $\pi$BI with multiparty, asynchronous communication [Honda et al. 2008; Scalas and Yoshida 2019]. An asynchronous version of $\pi$BI could be defined by following the work of DeYoung et al. [2012] to maximize concurrency. Also, the works [Caires and Pérez 2016; Carbone et al. 2016] already provide insights on how to exploit $\pi$BI to analyze multiparty protocols.

Second, variations and extensions of BI could provide new insights. For example, the $!A$ modality is not incompatible with BI, and can be added to obtain a type $!A \simeq A * \cdots * A$. The intuitive interpretation is that the provider of $!A$ can create an instance of $A$ from scratch, thus not sharing its origin with the other instances. This new type would seem incomparable with the corresponding modality of LL, which makes it interesting to study what interpretations could admit.

## REFERENCES


Gabrielle Anderson and David Pym. 2016. A calculus and logic of bunched resources and processes. *Theor. Comput. Sci.* 614 (2016), 63–96. https://doi.org/10.1016/j.tcs.2015.11.035

Robert Atkey. 2004. A $\lambda$-Calculus for Resource Separation. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, 158–170. https://doi.org/10.1007/978-3-540-27836-8_16

Robert Atkey. 2006. *Substructural Simple Type Theories for Separation and In-place Update.* Ph. D. Dissertation. University of Edinburgh.

Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

Josh Berdine and Peter O'Hearn. 2006. Strong Update, Disposal, and Encapsulation in Bunched Typing. In *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 158)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 81–98. https://doi.org/10.1016/j.entcs.2006.04.006

Stephen Brookes and Peter O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65.

James Brotherston. 2012. Bunched Logics Displayed. *Studia Logica* 100, 6 (2012), 1223–1254. https://doi.org/10.1007/s11225-012-9449-0

James Brotherston and Cristiano Calcagno. 2010. Classical BI: Its Semantics and Proof Theory. *Logical Methods in Computer Science* 6, 3 (2010), 42 pages. https://doi.org/10.2168/LMCS-6(3:3)2010

James Brotherston and Jules Villard. 2015. Sub-classical Boolean Bunched Logics and the Meaning of Par. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany (LIPIcs, Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 325–342. https://doi.org/10.4230/LIPIcs.CSL.2015.325

Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9688)*, Elvira Albert and Ivan Lanese (Eds.). Springer, 74–95. https://doi.org/10.1007/978-3-319-39570-8_6

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.). Springer, Berlin, Heidelberg, 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 330–349.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (March 2016), 367–423. https://doi.org/10.1017/S0960129514000218

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Matthew Collinson, David Pym, and Edmund Robinson. 2008. Bunched polymorphism. *Mathematical Structures in Computer Science* 18, 6 (2008), 1091–1132. https://doi.org/10.1017/S0960129508007159

Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. 2021. Session Logical Relations for Noninterference. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470654

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 228–242. https://doi.org/10.4230/LIPIcs.CSL.2012.228

Simon Docherty. 2019. *Bunched Logics: A Uniform Approach.* Ph. D. Dissertation. University College London.

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Daniele Gorla. 2010. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation* 208, 9 (2010), 1031–1053. https://doi.org/10.1016/j.ic.2010.05.002

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1381)*, Chris Hankin (Ed.). Springer, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.* 3, POPL (2019), 24:1–24:29. https://doi.org/10.1145/3290337

Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. 2011. On Asynchronous Session Semantics. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6722)*, Roberto Bruni and Jürgen Dingel (Eds.). Springer, 228–243. https://doi.org/10.1007/978-3-642-21461-5_15

Robin Milner. 1992. Functions as Processes. *Mathematical Structures in Computer Science* 2, 2 (1992), 119–141. https://doi.org/10.1017/S0960129500001407

Peter O'Hearn. 2003. On bunched typing. *Journal of Functional Programming* 13, 4 (2003), 747–796. https://doi.org/10.1017/S0956796802004495

Peter O'Hearn. 2019. Separation logic. *CACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Peter O'Hearn and David Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *Information and Computation* 239 (Dec. 2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001

Kirstin Peters. 2019. Comparing Process Calculi Using Encodings. In *Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019 (EPTCS, Vol. 300)*, Jorge A. Pérez and Jurriaan Rot (Eds.). 19–38. https://doi.org/10.4204/EPTCS.300.2

Klaas Pruiksma and Frank Pfenning. 2021. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming* 120 (2021), 100637. https://doi.org/10.1016/j.jlamp.2020.100637

David Pym. 2002. *The semantics and proof theory of the logic of bunched implications.* Applied logic series, Vol. 26. Kluwer.

David Pym, Peter O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1 (2004), 257–305. https://doi.org/10.1016/j.tcs.2003.11.020

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-server sessions in linear logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–31. https://doi.org/10.1145/3473567

Davide Sangiorgi and David Walker. 2003. *The pi-calculus: A Theory of Mobile Processes.* Cambridge University Press.

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Bernardo Toninho, Luis Caires, and Frank Pfenning. 2012. Functions as Session-Typed Processes. In *Foundations of Software Science and Computational Structures (Lecture Notes in Computer Science)*, Lars Birkedal (Ed.). Springer, Berlin, Heidelberg, 346–360. https://doi.org/10.1007/978-3-642-28729-9_23

Vasco Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

Philip Wadler. 2012. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 273–286. https://doi.org/10.1145/2364527.2364568

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418. https://doi.org/10.1017/S095679681400001X

# Appendix

## A THE $\pi$BI CALCULUS

In Section 2 we have omitted a number of reductions from Figure 3. They can be found in Figure 9.

RED-SPAWN-L
$$\frac{x \notin \mathrm{dom}(\sigma)}{(\boldsymbol{\nu} x).\big(\boldsymbol{\rho}[\sigma].P \mid_x Q\big) \longrightarrow \boldsymbol{\rho}[\sigma].(\boldsymbol{\nu} x).(P \mid_x Q)}$$

RED-RESTR-L
$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu} x).(P \mid Q) \longrightarrow (\boldsymbol{\nu} x).(P' \mid Q)}$$

RED-UNIT-L
$$(\boldsymbol{\nu} x).\big(\overline{x}\langle\rangle \mid_x x().Q\big) \longrightarrow Q$$

RED-FWD-L
$$\frac{x \neq y \qquad y \notin \mathrm{fn}(P)}{(\boldsymbol{\nu} x).([x \leftarrow y] \mid_x P) \longrightarrow P[y/x]}$$

RED-COMM-L
$$(\boldsymbol{\nu} x).\big(\overline{x}[y].(P_1 \mid P_2) \mid_x x(y).Q\big) \longrightarrow (\boldsymbol{\nu} x).\big(P_2 \mid_x (\boldsymbol{\nu} y).(P_1 \mid_y Q)\big)$$

Fig. 9. Omitted reduction rules

## B META-THEORETICAL PROPERTIES

In this section we first introduce auxiliary lemmas about (typed) spawn bindings and free/bound names of processes. We then present omitted proofs of subject congruence and subject reduction in Appendix B.3. After that we prove deadlock-freedom (Theorem 3.3) in Appendix B.4, introducing a *progress* lemma. Finally, in Appendix B.5 we detail our proof of weak normalization.

### B.1 Spawn bindings

We will need the following properties for proving subject reduction, all shown by induction on spawn binding typing.

Lemma B.1. *If $\sigma\colon \Delta_1 \rightsquigarrow \Delta_2$, then $\sigma\colon \Gamma(\Delta_1) \rightsquigarrow \Gamma(\Delta_2)$ for any bunched context $\Gamma$.*

Lemma B.2. *Suppose that $\sigma\colon \Gamma_1(x : A) \rightsquigarrow \Delta_2$ and $x \notin \mathrm{dom}(\sigma)$. Then $\Delta_2 = \Gamma_2(x : A)$ for some $\Gamma_2$. Furthermore, for any $\Delta$ we have $\sigma\colon \Gamma_1(\Delta) \rightsquigarrow \Gamma_2(\Delta)$.*

Proof. We proceed by induction on $\sigma\colon \Gamma_1(x : A) \rightsquigarrow \Delta_2$.

- **Case** SPAWN-WEAKEN**.** If $[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta)]\colon \Gamma(\Delta;\Delta') \rightsquigarrow \Gamma(\Delta')$, then, from the assumption that $x \notin \mathrm{dom}(\sigma)$, we know that $x$ does not occur $\Delta$. That means that $x$ it is either part of $\Delta$ or $\Gamma$. In either case, we can freely replace $x : A$ with an arbitrary bunch.
- **Case** SPAWN-CONTRACT**.** Similar to the previous case.
- **Case** SPAWN-MERGE**.** Suppose we have $(\sigma_1 \ltimes \sigma_2)\colon \Gamma_0(x : A) \rightsquigarrow \Delta_2$ with $\sigma_1\colon \Gamma_0(x : A) \rightsquigarrow \Delta_1$ for some intermediate bunch $\Delta_1$.
  Since $x \notin \mathrm{dom}(\sigma_1 \ltimes \sigma_2)$, we know that $x \notin \mathrm{dom}(\sigma_1)$. Hence, by induction hypothesis, $\Delta_1 = \Gamma_1(x : A)$, and we have $\sigma_1\colon \Gamma_1(\Delta) \rightsquigarrow \Gamma_2(\Delta)$ for any bunch $\Delta$.
  Furthermore, $x \notin \mathrm{restr}(\sigma_1)$ (otherwise we would not be able to replace $x : A$ with an arbitrary bunch $\Delta$). Hence, $x \notin \mathrm{dom}(\sigma_2)$, and by induction hypothesis we have $\Delta_2 = \Gamma_2(x : A)$ for some $\Gamma_2$, and $\sigma_2\colon \Gamma_1(\Delta) \rightsquigarrow \Gamma_2(\Delta)$ for any bunch $\Delta$. The desired result then follows by using SPAWN-MERGE again. ◻

Lemma B.3. *Suppose that* $\sigma\colon \Delta_1 \rightsquigarrow \Delta_2$ *and* $\sigma(x) = \{x_1, \ldots, x_n\}$. *Then* $\Delta_1 = \Gamma_1(x : A)$ *and* $\Delta_2 = \Gamma_2(x_1 : A \mid \cdots \mid x_n : A)$, *for some* $\Gamma_1$ *and* $\Gamma_2$. *Furthermore, for any bunch* $\Delta$ *we have:*

$$(\sigma \setminus \{x\}) \cup [y \mapsto \{y_1, \ldots, y_n\} \mid y \in \mathrm{fn}(\Delta)] \colon \Gamma_1(\Delta). \rightsquigarrow \Gamma_2\big(\Delta^{(1)} \mid \cdots \mid \Delta^{(n)}\big)$$

Proof. Similar to the previous lemma. □

## B.2 Names and substitutions

Lemma B.4. *If* $\Delta \vdash P :: x : C$, *then* $\mathrm{fn}(P) = \mathrm{fn}(\Delta) \cup \{x\}$.

Lemma B.5. *If* $(\nu x).(P \mid (\nu y).(Q \mid R))$ *is a well-typed process, then* $x$ *is shared either between* $P$ *and* $Q$*, or between* $P$ *and* $R$*, but not between all the three subprocesses, and* $y$ *is shared between* $Q$ *and* $R$*. That is, either* $x \in \mathrm{fn}(P) \cap \mathrm{fn}(Q)$ *and* $x \notin \mathrm{fn}(R)$*, or* $x \in \mathrm{fn}(P) \cap \mathrm{fn}(R)$ *and* $x \notin \mathrm{fn}(Q)$*. And in both cases we have* $y \notin \mathrm{fn}(P)$

This lemma implies that whenever we have a typed process $(\nu x).(P \mid (\nu y).(Q \mid R))$, then one of the congruences cong-assoc-l or congr-assoc-r apply.

Lemma B.6. *If* $(\nu x).(P \mid \rho[\sigma].Q)$ *is a well-typed process, and* $x \notin \sigma$*, then* $\mathrm{fn}(P) \cap \mathrm{restr}(\sigma) = \emptyset$.

Similarly to the previous lemma, this lemma implies that for a well-typed process of the form $(\nu x).(P \mid \rho[\sigma].Q)$ either red-spawn or red-spawn-r apply.

Lemma B.7. *If* $(\nu x).(\rho[\sigma_1].P \mid \rho[\sigma_2].Q)$ *is a well-typed process, then* $\sigma_1$ *and* $\sigma_2$ *are independent.*

Lemma B.8. *If* $\Delta \vdash P :: x : C$ *and* $\theta$ *is an injective substitution, then* $\Delta\theta \vdash P\theta :: \theta(x) : C$.

## B.3 Proof of subject reduction

Theorem 3.1. *If* $\Delta \vdash P :: x : C$*, then* $P \equiv Q$ *and* $P \longrightarrow Q$ *imply* $\Delta \vdash Q :: x : C$.

Proof (structural congruence). We proceed by induction on $P \equiv Q$, examining the possible typing derivations of $P$.

**Case** cong-assoc-l**.** This congruence states that the order of independent cuts does not matter. Corresponds to the following proof conversion:

$$\dfrac{\Delta_1 \vdash P :: x : A \qquad \dfrac{\Delta_2 \vdash Q :: y : B \qquad \Gamma(x : A \mid y : B) \vdash R :: z : C}{\Gamma(x : A \mid \Delta_2) \vdash (\nu y).(Q \mid_y R) :: z : C}}{\Gamma(\Delta_1 \mid \Delta_2) \vdash (\nu x).\big(P \mid_x (\nu y).(Q \mid_y R)\big) :: z : C}$$

$$\leftrightsquigarrow$$

$$\dfrac{\Delta_2 \vdash Q :: y : B \qquad \dfrac{\Delta_1 \vdash P :: x : A \qquad \Gamma(x : A \mid y : B) \vdash R :: z : C}{\Gamma(\Delta_1 \mid y : B) \vdash (\nu x).(P \mid_x R) :: z : C}}{\Gamma(\Delta_1 \mid \Delta_1) \vdash (\nu y).\big(Q \mid_y (\nu x).(P \mid_x R)\big) :: z : C}$$

**Case** congr-assoc-r**.** This congruence states that the order of subsequent cuts does not matter. Corresponds to the following proof conversion:

$$\dfrac{\Delta_1 \vdash P :: x : A \qquad \dfrac{\Delta_2(x : A) \vdash Q :: y : B \qquad \Gamma(y : B) \vdash R :: z : C}{\Gamma(\Delta_2(x : A)) \vdash (\nu y).(Q \mid_y R) :: z : C}}{\Gamma(\Delta_2(\Delta_1)) \vdash (\nu x).\big(P \mid_x (\nu y).(Q \mid_y R)\big) :: z : C}$$

$$\rightsquigarrow\!\!\!\!\leftarrow$$

$$\dfrac{\dfrac{\Delta_1 \vdash P :: x : A \qquad \Delta_2(x : A) \vdash Q :: y : B}{\Delta_2(\Delta_1) \vdash (\boldsymbol{\nu} x).(P \mid_x Q) :: y : B} \qquad \Gamma(y : B) \vdash R :: z : C}{\Gamma(\Delta_2(\Delta_1) \vdash (\boldsymbol{\nu} y).\big((\boldsymbol{\nu} x).(P \mid_x Q) \mid_y R\big) :: z : C}$$

**Case** CONGR-SPAWN-SWAP**.** Similarly to the previous case, if two spawn bindings $\sigma$ and $\sigma'$ are independent, than they correspond to two independent applications of STRUCT that can be commuted past each other. For example,

$$\dfrac{\dfrac{\Gamma(\emptyset_{\mathrm{a}} \mid \Delta^{(1)}\,;\Delta^{(2)}) \vdash P :: z : C}{\Gamma(\emptyset_{\mathrm{a}} \mid \Delta) \vdash \boldsymbol{\rho}\,[x \mapsto \{x_1, x_2\} \mid x \in \mathrm{fn}(\Delta)]\,.P :: z : C}}{\Gamma(\Delta' \mid \Delta) \vdash \boldsymbol{\rho}\,[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')]\,.\boldsymbol{\rho}\,[x \mapsto \{x_1, x_2\} \mid x \in \mathrm{fn}(\Delta)]\,.P :: z : C}$$

$$\rightsquigarrow\!\!\!\!\leftarrow$$

$$\dfrac{\dfrac{\Gamma(\emptyset_{\mathrm{a}} \mid \Delta^{(1)}\,;\Delta^{(2)}) \vdash P :: z : C}{\Gamma(\Delta' \mid \Delta^{(1)}\,;\Delta^{(2)}) \vdash \boldsymbol{\rho}\,[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')]\,.P :: z : C}}{\Gamma(\Delta' \mid \Delta) \vdash \boldsymbol{\rho}\,[x \mapsto \{x_1, x_2\} \mid x \in \mathrm{fn}(\Delta)]\,.\boldsymbol{\rho}\,[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')]\,.P :: z : C}$$

**Closure under program contexts.** Using the induction hypothesis. □

PROOF (REDUCTION). By induction on the reduction relation $P \longrightarrow Q$ and the typing derivation. The case RED-CONGR follows using the previous theorem and the induction hypothesis. The cases RED-RESTR-L and RED-EVAL-CTXT for evaluation contexts of the form $\mathcal{K} = (\boldsymbol{\nu} x).(\mathcal{K}'[\,\cdot\,] \mid P)$ and $\mathcal{K} = \boldsymbol{\rho}[\sigma].\mathcal{K}'[\,\cdot\,]$ follow from the induction hypothesis.

**Case** RED-UNIT-L**.** It corresponds to the following reduction of proofs, or its additive version:

$$\dfrac{\emptyset_{\mathrm{m}} \vdash \overline{x}\langle\rangle :: x : \mathbf{1}_{\mathrm{m}} \qquad \dfrac{\Gamma(\emptyset_{\mathrm{m}}) \vdash Q :: z : C}{\Gamma(x : \mathbf{1}_{\mathrm{m}}) \vdash x().Q :: z : C}}{\Gamma(\emptyset_{\mathrm{m}}) \vdash (\boldsymbol{\nu} x).(\overline{x}\langle\rangle \mid_x x().Q) :: z : C} \quad \rightsquigarrow \quad \Gamma(\emptyset_{\mathrm{m}}) \vdash Q :: z : C$$

**Case** RED-COMM-L**.** It corresponds to the following reduction of proofs, or its additive version:

$$\dfrac{\dfrac{\Delta_1 \vdash P_1 :: y : A \qquad \Delta_2 \vdash P_2 :: x : B}{\Delta_1\,, \Delta_2 \vdash \overline{x}[y].(P_1 \mid P_2) :: x : A * B} \qquad \dfrac{\Gamma(y : A\,, x : B) \vdash Q :: z : C}{\Gamma(x : A * B) \vdash x(y).Q :: z : C}}{\Gamma(\Delta_1\,, \Delta_2) \vdash (\boldsymbol{\nu} x).\big(\overline{x}[y].(P_1 \mid P_2) \mid x(y).Q\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\Delta_2 \vdash P_2 :: x : B \qquad \dfrac{\Delta_1 \vdash P_1 :: y : A \qquad \Gamma(y : A\,, x : B) \vdash Q :: z : C}{\Gamma(\Delta_1\,, x : B) \vdash (\boldsymbol{\nu} y).(P_1 \mid Q) :: z : C}}{\Gamma(\Delta_1\,, \Delta_2) \vdash (\boldsymbol{\nu} x).\big(P_2 \mid (\boldsymbol{\nu} y).(P_1 \mid Q)\big) :: z : C}$$

**Case** RED-COMM-R. It corresponds to the following reduction of proofs, or its additive version:

$$\dfrac{\dfrac{\Delta\,,y : A \vdash Q :: x : B}{\Delta \vdash x(y).Q :: x : A \mathrel{-\!\!*} B} \qquad \dfrac{\Delta_1 \vdash P_1 :: y : A \qquad \Gamma(x : B\,,\Delta_2) \vdash P_2 :: z : C}{\Gamma(\Delta_1\,,x : A \mathrel{-\!\!*} B\,,\Delta_2) \vdash \overline{x}[y].(P_1 \mid P_2) :: z : C}}{\Gamma(\Delta_1\,,\Delta\,,\Delta_2) \vdash (\boldsymbol{\nu}x).\big(x(y).Q \mid \overline{x}[y].(P_1 \mid P_2)\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\dfrac{\Delta_1 \vdash P_1 :: y : A \qquad \Delta\,,y : A \vdash Q :: x : B}{\Delta\,,\Delta_1 \vdash (\boldsymbol{\nu}y).(P_1 \mid Q) :: x : B} \qquad \Gamma(x : B\,,\Delta_2) \vdash P_2 :: z : C}{\Gamma(\Delta\,,\Delta_1\,,\Delta_2) \vdash (\boldsymbol{\nu}x).\big((\boldsymbol{\nu}y).(P_1 \mid Q) \mid P_2\big) :: z : C}$$

**Case** RED-CASE. Corresponds to the following reduction of proofs (for $\ell = \mathsf{inl}$):

$$\dfrac{\dfrac{\Delta \vdash P :: x : A}{\Delta \vdash x \triangleleft \mathsf{inl}.P :: x : A \vee B} \qquad \dfrac{\Gamma(x : A) \vdash Q_1 :: z : C \qquad \Gamma(x : B) \vdash Q_2 :: z : C}{\Gamma(x : A \vee B) \vdash x \triangleright \mathsf{case}(Q_1, Q_2) :: z : C}}{\Gamma(\Delta) \vdash (\boldsymbol{\nu}x).\big(x \triangleleft \mathsf{inl}.P \mid x \triangleright \mathsf{case}(Q_1, Q_2)\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\Delta \vdash P :: x : A \qquad \Gamma(x : A) \vdash Q_1 :: z : C}{\Gamma(\Delta) \vdash (\boldsymbol{\nu}x).\big(P \mid Q_1\big) :: z : C}$$

**Case** RED-FWD-L. It corresponds to the following reduction of proofs, using Lemma B.8:

$$\dfrac{y : A \vdash [x \leftarrow y] :: x : A \qquad \Delta(x : A) \vdash P :: z : C}{\Delta(y : A) \vdash (\boldsymbol{\nu}x).\big([x \leftarrow y] \mid P\big) :: z : C} \quad \rightsquigarrow \quad \Delta(y : A) \vdash P[y/x] :: z : C$$

**Case** RED-FWD-R. It corresponds to the following reduction of proofs:

$$\dfrac{\Delta \vdash P :: x : A \qquad x : A \vdash [y \leftarrow x] :: y : A}{\Delta \vdash (\boldsymbol{\nu}x).\big(P \mid [y \leftarrow x]\big) :: y : A} \quad \rightsquigarrow \quad \Delta \vdash P[y/x] :: y : A$$

**Case** RED-SPAWN. This corresponds to pushing application of STRUCT past the cut. By Lemma B.3, the derivation that we have is of the following shape:

$$\dfrac{\Delta \vdash P :: x : A \qquad \dfrac{\Gamma(x_1 : A \mid \cdots \mid x_n : A) \vdash Q :: z : C \qquad \sigma \colon \Gamma'(x : A) \rightsquigarrow \Gamma(x_1 : A \mid \cdots \mid x_n : A)}{\Gamma'(x : A) \vdash \rho[\sigma].Q :: z : C}}{\Gamma'(\Delta) \vdash (\boldsymbol{\nu}x).\big(P \mid \rho[\sigma].Q\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta^{(n)} \vdash P^{(n)} :: x_n : A \qquad \Gamma(x_1 : A \mid \cdots \mid x_n : A) \vdash Q :: z : C}{\Gamma(x_1 : A \mid \cdots \mid \Delta^{(n)}) \vdash (\boldsymbol{\nu}x_n).(P^{(n)} \mid Q) :: z : C}}{\Delta^{(1)} \vdash P^{(1)} :: x_1 : A \qquad \ldots}}{\Gamma(\Delta^{(1)} \mid \cdots \mid \Delta^{(n)}) \vdash (\boldsymbol{\nu}x_1).(P^{(1)} \mid \ldots (\boldsymbol{\nu}x_n).(P^{(n)} \mid Q) \ldots) :: z : C} \qquad \sigma' \colon \Gamma'(\Delta) \rightsquigarrow \Gamma(\Delta^{(1)} \mid \cdots \mid \Delta^{(n)})}{\Gamma'(\Delta) \vdash \rho[\sigma'].(\boldsymbol{\nu}x_1).(P^{(1)} \mid \ldots (\boldsymbol{\nu}x_n).(P^{(n)} \mid Q) \ldots) :: z : C}$$

The spawn binding $\sigma' = (\sigma \setminus x) \cup \big[y \mapsto \{y_1, \ldots, y_n\}\big]_{y \in \mathrm{fn}(\Pi)}$ is well-typed according to Lemma B.3.

**Case** RED-SPAWN-L. As in the previous case, this reduction corresponds to moving STRUCT past a cut. Since $\boldsymbol{\rho}[\sigma].P$ appears on the left side of the composition, we know that the application of STRUCT was independent from $Q$ and from the cut. The corresponding proof transformation is as follows:

$$\dfrac{\dfrac{\Pi \vdash P :: x : A \qquad \sigma \colon \Pi' \rightsquigarrow \Pi}{\Pi' \vdash \boldsymbol{\rho}[\sigma].P :: x : A} \qquad \Gamma(x : A) \vdash Q :: z : C}{\Gamma(\Pi') \vdash (\boldsymbol{\nu} x).\big(\boldsymbol{\rho}[\sigma].P \mid Q\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\dfrac{\Pi \vdash P :: x : A \qquad \Gamma(x : A) \vdash Q :: z : C}{\Gamma(\Pi) \vdash (\boldsymbol{\nu} x).(P \mid Q) :: z : C} \qquad \sigma \colon \Gamma(\Pi') \rightsquigarrow \Gamma(\Pi)}{\Gamma(\Pi') \vdash \boldsymbol{\rho}[\sigma].(\boldsymbol{\nu} x).\big(P \mid Q\big) :: z : C}$$

Where we know that $\sigma \colon \Gamma(\Pi') \rightsquigarrow \Gamma(\Pi)$ by Lemma B.1.

**Case** RED-SPAWN-R. Similar to the previous case. Since $x \notin \sigma$ we know that the application of STRUCT is independent of the cut. That is, by Lemma B.2, the contexts for $Q$ are of the shape $\Gamma(x : A)$ and $\Gamma'(x : A)$.

$$\dfrac{\Delta \vdash P :: x : A \qquad \dfrac{\Gamma(x : A) \vdash Q :: z : C \qquad \sigma \colon \Gamma'(x : A) \rightsquigarrow \Gamma(x : A)}{\Gamma'(x : A) \vdash \boldsymbol{\rho}[\sigma].Q :: z : C}}{\Gamma(\Delta) \vdash (\boldsymbol{\nu} x).\big(P \mid \boldsymbol{\rho}[\sigma].Q\big) :: z : C}$$

$$\rightsquigarrow$$

$$\dfrac{\dfrac{\Delta \vdash P :: x : A \qquad \Gamma(x : A) \vdash Q :: z : C}{\Gamma(\Delta) \vdash (\boldsymbol{\nu} x).(P \mid Q) :: z : C} \qquad \sigma \colon \Gamma'(\Delta) \rightsquigarrow \Gamma(\Delta)}{\Gamma'(\Delta) \vdash \boldsymbol{\rho}[\sigma].\big((\boldsymbol{\nu} x).(P \mid Q)\big) :: z : C}$$

Where $\sigma$ has the appropriate type by Lemma B.2.

**Case** RED-SPAWN-MERGE. Directly using the rules for spawn prefix typing.

□

### B.4 Proof of deadlock-freedom

To prove deadlock-freedom, we first need to analyze when a process is *not* stuck, i.e. when it can reduce. We define the *readiness* of a process, which is a means to syntactically determine whether a process can reduce. This notion of readiness[1] is useful when implementing $\pi$BI as, e.g., a programming language: a reduction can be derived by simply analyzing the syntax of a program.

To define readiness, we need to know which names can be used for a communication. We define this as a process' set of *active names*: free names used for communication prefixes not guarded by other communication prefixes.

[1]In some literature this notion is referred to as "liveness", but we did not want to confused it here with a more semantic notion of liveness.

*Definition B.9 (Active Names).* Given a process $P$, we define the set of *active names* of $P$, denoted $\mathsf{an}(P)$, as follows:

$$\begin{aligned}
\mathsf{an}(\overline{x}\langle\rangle) &= \{x\} & \mathsf{an}((\boldsymbol{\nu}x).(P \mid_x Q)) &= (\mathsf{an}(P) \cup \mathsf{an}(Q)) \setminus \{x\} \\
\mathsf{an}(x().P) &= \{x\} & \mathsf{an}(\boldsymbol{\rho}[\sigma].P) &= \mathsf{dom}(\sigma) \cup (\mathsf{an}(P) \setminus \mathsf{restr}(\sigma)) \\
\mathsf{an}(\overline{x}[y].(P \mid Q)) &= \{x\} & \mathsf{an}(x(y).P) &= \{x\} \\
\mathsf{an}(x \triangleleft \mathsf{inl}.P) = \mathsf{an}(x \triangleleft \mathsf{inr}.P) &= \{x\} & \mathsf{an}(x \triangleright \mathsf{case}(P, Q)) &= \{x\} \\
\mathsf{an}([x \leftarrow y]) &= \{x, y\}
\end{aligned}$$

Lemma B.10. *If $P \equiv Q$ then $\mathsf{an}(P) = \mathsf{an}(Q)$.*

Proof. There are no rules of structural congruence that add or remove prefixes. Moreover, the only rule of structural congruence that affects names only affects *bound* names, and active names are free by definition. □

*Definition B.11 (Ready process).* A process $P$ is *ready*, denoted $\mathsf{ready}(P)$, if it is expected to reduce. Formally, the ready predicate is defined by the following rules:

$$\frac{\mathsf{ready}(Q)}{\mathsf{ready}(\boldsymbol{\rho}[\sigma].Q)} \qquad \frac{}{\mathsf{ready}(\boldsymbol{\rho}[\sigma].\boldsymbol{\rho}[\sigma'].Q)} \qquad \frac{\mathsf{ready}(Q) \qquad P \equiv Q}{\mathsf{ready}(P)} \qquad \frac{x \in \mathsf{an}(P) \cap \mathsf{an}(Q)}{\mathsf{ready}((\boldsymbol{\nu}x).(P \mid_x Q))}$$

$$\frac{\mathsf{ready}(P) \lor \mathsf{ready}(Q)}{\mathsf{ready}((\boldsymbol{\nu}x).(P \mid_x Q))} \qquad \frac{}{\mathsf{ready}((\boldsymbol{\nu}x).(\boldsymbol{\rho}[\sigma].P \mid_x Q))} \qquad \frac{}{\mathsf{ready}((\boldsymbol{\nu}x).(P \mid_x \boldsymbol{\rho}[\sigma].Q))}$$

$$\frac{}{\mathsf{ready}((\boldsymbol{\nu}x).([x \leftarrow y] \mid_x Q))} \qquad \frac{}{\mathsf{ready}((\boldsymbol{\nu}x).(P \mid_x [x \leftarrow y]))}$$

The following then assures that well-typed, ready processes can reduce:

Lemma B.12 (Progress). *Suppose given a process $P$ such that $\Gamma \vdash P :: z : C$ and $\mathsf{ready}(P)$. Then, there exists a process $S$ such that $P \longrightarrow S$.*

Proof. By induction on the derivation of $\mathsf{ready}(P)$.

- **Case $P = \boldsymbol{\rho}[\sigma].Q$ and $\mathsf{ready}(Q)$.** By the IH, there exists $S'$ such that $Q \longrightarrow S'$. By Rule red-eval-ctxt, $P \longrightarrow \boldsymbol{\rho}[\sigma].S'$.
- **Case $P = \boldsymbol{\rho}[\sigma].\boldsymbol{\rho}[\sigma'].Q$.** By Rule red-spawn-merge, $P \longrightarrow \boldsymbol{\rho}[\sigma \ltimes \sigma'].Q$.
- **Case $P \equiv Q$ and $\mathsf{ready}(Q)$.** By the IH, there exists $S$ such that $Q \longrightarrow S$. By Rule red-congr, $P \longrightarrow S$.
- **Case $P = (\boldsymbol{\nu}x).(Q \mid R)$ and $x \in \mathsf{an}(Q) \cap \mathsf{an}(R)$.** We have $\Gamma = \Gamma'(\Delta)$ where $\Delta \vdash Q :: x : A$ and $\Gamma'(x : A) \vdash R :: z : C$. Since $x \in \mathsf{an}(Q) \cap \mathsf{an}(R)$, there is an unguarded prefix with subject $x$ in both $Q$ and $R$. Being unguarded, the prefix in $Q$ appears inside a sequence of $n$ cuts and spawns. Similarly, the prefix in $R$ appears inside a sequence of $m$ cuts and spawns. By induction on $n$ and $m$, we show that there exists a process $S$ such that $P \longrightarrow S$.
  - If $n = 0$ and $m = 0$, the analysis depends on whether $Q = [x \leftarrow y]$ or $R = [y \leftarrow x]$, or neither. If so, this case is analogous the appropriate of the latter two cases of this proof.
    Otherwise, neither $Q$ nor $R$ is a forwarder. In that case, $Q$ is typable with a right rule for output, input, selection, or branching on $x$, depending on the type $A$. Similarly, $R$ is typable with a dual left rule on $x$. Suppose, as a representative example, that $A = B_1 * B_2$. Then, $Q$ is typable with Rule Sep-r, i.e. $Q = \overline{x}[y].(Q_1 \mid Q_2)$. Similarly, $R$ is typable with Rule Sep-l, i.e. $R =\in x(y').R'$. Then $P = (\boldsymbol{\nu}x).(\overline{x}[y].(Q_1 \mid Q_2)| \in x(y').R)$. Let $S = (\boldsymbol{\nu}x).(Q_2 \mid (\boldsymbol{\nu}y).(Q_1 \mid R'[y/y']))$. By Rule red-comm-l, $P \longrightarrow S$.

- If $n = n' + 1$, then the analysis depends on whether the outermost construct in $Q$ is a cut or a spawn. We thus consider these two cases:
  - If the outermost construct is a cut, then $P = (\boldsymbol{\nu}x).((\boldsymbol{\nu}w).(Q_1 \mid Q_2) \mid R)$. The prefix on $x$ appears in $Q_2$, under a sequence of $n'$ sequence of and spawns. Since $x \in \mathrm{fn}((\boldsymbol{\nu}w).(Q_1 \mid Q_2))$, we know $w \neq x$. This means that $x \notin \mathrm{fn}(Q_1)$. Hence, by Rule CONGR-ASSOC-R, $P \equiv (\boldsymbol{\nu}w).(Q_1 \mid (\boldsymbol{\nu}x).(Q_2 \mid R))$. By the IH, there exists $S'$ such that $(\boldsymbol{\nu}x).(Q_2 \mid R) \longrightarrow S'$. Let $S = (\boldsymbol{\nu}w).(Q_1 \mid S')$. Then, by Rules RED-EVAL-CTXT and RED-CONGR, $P \longrightarrow S$.
  - If the outermost construct is a spawn, then $Q = \boldsymbol{\rho}[\sigma].Q'$ and the proof follows as in the case where $P = (\boldsymbol{\nu}x).(\boldsymbol{\rho}[\sigma].Q \mid R)$.
- If $m = m' + 1$, the analysis is analogous to the case above.

– **Case** $P = (\boldsymbol{\nu}x).(Q \mid R)$ **and** $\mathrm{ready}(Q)$ **or** $\mathrm{ready}(R)$**.** W.l.o.g., assume $\mathrm{ready}(Q)$. By the IH, there exists $S'$ such that $Q \longrightarrow S'$. By Rule RED-EVAL-CTXT, $P \longrightarrow (\boldsymbol{\nu}x).(S' \mid R)$.

– **Case** $P = (\boldsymbol{\nu}x).(\boldsymbol{\rho}[\sigma].Q \mid R)$**.** By typability, $x \notin \sigma$. Hence, by Rule RED-SPAWN-L, $P \longrightarrow \boldsymbol{\rho}[\sigma].(\boldsymbol{\nu}x).(Q \mid R)$.

– **Case** $P = (\boldsymbol{\nu}x).(Q \mid \boldsymbol{\rho}[\sigma].R)$**.** The analysis depends on wheter $x \in \sigma$ or not, so we consider two cases:
- If $x \in \sigma$, then $\sigma(x) = \{x_1, \ldots, x_n\}$. Let $\sigma' = (\sigma \setminus \{x\}) \cup [w => \{w_1, \ldots, w_n\} \mid w \in (\mathrm{fn}(Q) \setminus \{x\})]$ and $S = \boldsymbol{\rho}[\sigma'].(\boldsymbol{\nu}x_1).(Q^{(1)} \mid \ldots (\boldsymbol{\nu}x_n).(Q^{(n)} \mid R)\ldots)$. By Rule RED-SPAWN, $P \longrightarrow S$.
- If $x \notin \sigma$, by Rule RED-SPAWN-R, $P \longrightarrow \boldsymbol{\rho}[\sigma].(\boldsymbol{\nu}x).(Q \mid R)$.

– **Case** $P = (\boldsymbol{\nu}x).([x \leftarrow y.\mid Q)]$ By Rule RED-FWD-L, $P \longrightarrow Q[y/x]$.

– **Case** $P = (\boldsymbol{\nu}x).(Q \mid [y \leftarrow x.)]$ By Rule RED-FWD-R, $P \longrightarrow Q[y/x]$. □

THEOREM 3.3 (DEADLOCK-FREEDOM). *Given an empty bunch $\Sigma$, if $\Sigma \vdash P :: z : A$ with $A \in \{\mathbf{1}_\mathrm{m}, \mathbf{1}_\mathrm{a}\}$, then either (i) $P \equiv \overline{z}\langle\rangle$, or (ii) $P \equiv \boldsymbol{\rho}[\emptyset].\overline{z}\langle\rangle$, or (iii) there exists $S$ such that $P \longrightarrow S$.*

PROOF. If the process $P$ is ready, then the result follows from Lemma B.12. Otherwise, towards a contradiction, assume $P \not\equiv \overline{z}\langle\rangle$ and $P \not\equiv \boldsymbol{\rho}[\emptyset].\overline{z}\langle\rangle$. W.l.o.g., assume $P$ is not prefixed by an empty spawn. Since $\Sigma$ contains no names, and $P$ is not an empty output on $z$, the only possibility is that $P$ is a cut: $P \equiv (\boldsymbol{\nu}x).(Q \mid_x R)$. There are several possibilities for $Q$ and $R$: they can be communication prefixes on $x$, they can be spawn prefixes with only $x$ in the domain, they can be cuts, or they can be forwarders on $x$.

- $Q$ or $R$ is a spawn prefix with only $x$ in the domain. By definition, $P$ is ready.
- $Q$ or $R$ is a forwarder on $x$. By definition, $P$ is ready.
- $Q$ is a communication prefix on $x$. By typability, $x$ must be free in $R$, so $R$ must contain an action on $x$: a spawn prefix with $x$ in the domain, a forwarder on $x$, or a communication prefix on $x$. If $x \in \mathrm{an}(R)$, then $x \in \mathrm{an}(Q) \cap \mathrm{an}(R)$, so $P$ is ready by definition. Otherwise, there is a name $y$ such that the action on $x$ in $R$ is guarded by a spawn prefix with $y$ in the domain or by a communication prefix on $y$. Either way, there must be a cut on $y$ in $R$, i.e., $R \equiv (\boldsymbol{\nu}y).(R_1 \mid_y R_2)$. We show by induction on the structures of $R_1$ and $R_2$ that $R$ is ready.
  - If $R_2$ is a spawn with $y$ in the domain, then $R$ is ready by definition; $R_2$ cannot be a spawn with $y$ in the domain.
  - If $R_1$ or $R_2$ is a forwarder on $x$, then $R$ is ready by definition.
  - If $R_1$ is a communication prefix on $y$, the analysis depends on whether $y \in \mathrm{an}(R_2)$. If so, $R$ is ready by defition. Otherwise, there is a name $z$ such that the action on $y$ in $R_2$ is guarded by a spawn prefix with $z$ in the domain or by a communication prefix on $z$. Either way, there must be a cut on $z$ in $R_2$, i.e., $R_2 \equiv (\boldsymbol{\nu}z).(R_2' \mid_z R_2'')$. By the IH, $R_2'$ or $R_2''$ is ready, so $R_2$ is ready by definition. Hence, $R$ is ready by definition.
  - The case where $R_2$ is a communication prefix on $y$ is analogous to the previous case.

  - If $R_1$ or $R_2$ is a cut, then by the IH, $R_1$ or $R_2$ is ready, so $R$ is ready by definition.
  Since $R$ is ready, also $P$ is ready.
- $R$ is a communication prefix on $x$. This case is analogous to the previous case.

In each case, the assumption that $P$ is not ready is contradicted, so $P \equiv \overline{z}\langle\rangle$ or $P \equiv \boldsymbol{\rho}[\emptyset].\overline{z}\langle\rangle$. □

### B.5 Weak normalization

Recall our normalization strategy: If a process can perform a communication reduction or a forwarder reduction, then we do exactly that reduction. If a process can only perform a reduction that involves a spawn prefix, then we (1) select (an active) spawn prefix with the least depth; (2) perform the spawn reduction; (3) propagate the newly created spawn prefix to the very top-level, merging it with other spawn prefixes along the way.

We will show that this reduction strategy terminates, by assigning a particular lexicographical measure to the processes and showing that our strategy strictly reduces this measure. This measure counts the number of communication prefixes in a process at a given depth, where depth is determined by spawn prefixes. Let us make this precise.

For a process $P$ we consider its *skeleton* $\mathrm{skel}(P)$, which is a finite map assigning to each number $n$ the amount of communication prefixes at depth $n$ and above. Since the processes are finite, each communication prefix occurs at a finite depth. That means that $\mathrm{skel}(P)(k) = 0$ for any $k$ greater than the maximal depth of the process. Formally, we define skeletons as follows.

*Definition B.13 (Skeleton).* A function $s\colon \mathbb{N} \to \mathbb{N}$ is a *skeleton of depth* $k$, if $\forall i > k\colon s(i) = 0$.

We define $\mathbb{S}_k$ as the set of all skeletons of depth $k$. Moreover, we define $\mathbb{S} \triangleq \bigcup_{k\in\mathbb{N}} \mathbb{S}_k$ and equip it with the strict quasi order $<$ such that

$$s_1 < s_2 \Leftrightarrow \exists j\colon \big(s_1(j) < s_2(j) \land \forall i > j\colon s_1(i) = s_2(i)\big)$$

We also define $s_1 \leq s_2 \triangleq (s_1 = s_2 \lor s_1 < s_2)$.

The following lemmas allow us to do well-founded recursion on skeletons.

Lemma B.14. *If $s_2 \in \mathbb{S}_k$ and $s_2 > s_1$ then $s_1 \in \mathbb{S}_k$.*

Proof. From $s_2 > s_1$ we get some $j$ such that $s_1(j) < s_2(j)$ and $\forall i > j\colon s_1(i) = s_2(i)$.

- **Case $j \leq k$.** Then $\forall i > k\colon s_1(i) = s_2(i) = 0$ which proves $s_1 \in \mathbb{S}_k$.
- **Case $j > k$.** This is impossible as we would have $s_1(j) < s_2(j) = 0$. □

Lemma B.15. *$(\mathbb{S}_k, <)$ is well-founded.*

Proof. Towards a contradiction, assume $s_0 > s_1 > \dots$ is an infinite descending chain of $\mathbb{S}_k$. Let $j_n$ be the witness for $s_n > s_{n+1}$, i.e. $s_n(j_n) > s_{n+1}(j_n)$ and $\forall i > j_n\colon s_n(i) = s_{n+1}(i)$. From $s_n, s_{n+1} \in \mathbb{S}_k$ we get $j_n < k$. Since there are finitely many natural numbers below $k$, by the pigeonhole principle, the sequence $j_0, j_1, \dots$ contains at least one number that repeats infinitely often. Among the ones that do, pick the greatest to be $m$. By definition, all the numbers larger than $m$ appear finitely often in $j_0, j_1, \dots$ and so there is a position $p$ such that $\forall n \geq p\colon j_n \leq m$. We obtain that $\forall n \geq p\colon s_n(m) \geq s_{n+1}(m)$. Moreover, let $i_0 < i_1 < \dots$ be such that $j_{i_0}, j_{i_1}, \dots$ consists of the infinite subsequence of the occurrences of $m$ in $j_p, j_{p+1}, \dots$, i.e. $m = j_{i_0} = j_{i_1} = \dots$. We have $s_{i_0}(m) > s_{i_0+1}(m) \geq \dots \geq s_{i_1}(m) > s_{i_1+1}(m) \geq \dots$. We obtain that $s_{i_0}(m) > s_{i_1}(m) > \dots$ is an infinite descending chain of $\mathbb{N}$, which is a contradiction. □

Lemma B.16. *$(\mathbb{S}, <)$ is well-founded.*

Proof. Towards a contradiction, assume $s_0 > s_1 > \dots$ is an infinite descending chain of $\mathbb{S}$. Since $s_0 \in \mathbb{S}_k$ for some $k$, by Lemma B.14 $\forall i\colon s_i \in \mathbb{S}_k$. Therefore we have an infinite descending chain of $\mathbb{S}_k$ which contradicts Lemma B.15. □

For a process $P$, its *skeleton* $\mathrm{skel}(P)$ is a finite map assigning to each number $n$ the amount of communication prefixes at depth $n$.

*Definition B.17 (Skeleton of P).* Given $s, s_1, s_2 \in \mathbb{S}$, we define:

$$[1](i) \triangleq \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases} \qquad (s_1 \oplus s_2)(i) \triangleq s_1(i) + s_2(i) \qquad (\blacktriangleright s)(i) \triangleq \begin{cases} s(0) & \text{if } i = 0 \\ s(i-1) & \text{if } i > 0 \end{cases}$$

The *skeleton of a process* $P$, written $\mathrm{skel}(P)$, is then defined as:

$$\mathrm{skel}(P) \triangleq \begin{cases} [1] & \text{if } P = [x \leftarrow y] \vee P = \overline{x}\langle\rangle \\ [1] \oplus \mathrm{skel}(Q_1) \oplus \mathrm{skel}(Q_2) & \text{if } P = \overline{x}[y].(Q_1 \mid Q_2) \vee P = x \triangleright \mathrm{case}(Q_1, Q_2) \\ [1] \oplus \mathrm{skel}(Q) & \text{if } P = x().Q \vee P = x(y).Q \vee P = x \triangleleft \ell.Q \\ \mathrm{skel}(Q_1) \oplus \mathrm{skel}(Q_2) & \text{if } P = (\boldsymbol{\nu} x).(Q_1 \mid Q_2) \\ \blacktriangleright \mathrm{skel}(Q) & \text{if } P = \boldsymbol{\rho}[\sigma].Q \end{cases}$$

Note that if $s = \mathrm{skel}(P)$ then $\forall i\colon s(i) \geq s(i+1)$.

For example, the skeleton

$$\mathrm{skel}((\boldsymbol{\nu} x).(\overline{x}\langle\rangle \mid \boldsymbol{\rho}[\sigma].(\boldsymbol{\nu} y).(\overline{y}\langle\rangle \mid \boldsymbol{\rho}[\sigma'].x().y().\overline{k}\langle\rangle))) = [0 \mapsto 5, 1 \mapsto 4; 2 \mapsto 3; \_ \mapsto 0].$$

*The measure.* Recall from the main part of the paper, that when computing a measure associated to the process we have to take special care of the top-level spawn prefix. We define the measure function $\mu$ as follows.1

$$\mu(P) \triangleq \begin{cases} \mathrm{skel}(Q) & \text{if } P = \boldsymbol{\rho}[\sigma].Q \\ \mathrm{skel}(P) & \text{otherwise} \end{cases}$$

Lemma B.18. *If $P \equiv Q$ then $\mu(P) = \mu(Q)$.*

Proof. None of the congruences can change whether the top-level construct is a spawn prefix. Furthermore, none of the congruences change depth of any communication prefixes. □

Lemma B.19. *The communication reductions strictly decrease the measure. That is, reductions* red-unit-l, red-comm-l, red-comm-r, red-case, red-fwd-l, red-fwd-r *decrease the measure $\mu$, even when occurring under arbitrary evaluation contexts.*

*Similarly, reduction* red-spawn-merge *strictly decreases the measure.*

Proof. Each of those reductions reduce the amount of communication prefixes at a given depth, and, as such, decrease the skeleton of the process. The only thing that we need to note is the special spawn prefix condition on $\mu$ in cases red-unit-l, red-fwd-l, and red-fwd-r. In those cases, the reduction might introduce a spawn prefix in front of the process. However, in that case the measure $\mu$ will still strictly decrease. □

As we have seen, the spawn reductions red-spawn, red-spawn-l, and red-spawn-r might temporarily increase the measure, but if we repeat them long enough then the measure will actually decrease.

Lemma B.20. *Let $\mathcal{K}_0[\cdot]$ be a non-empty evaluation context which may contain a $(\rho[\sigma_0].[\cdot])$ sub-context only at the top level. Let $\mathcal{K}_0[\rho[\sigma].Q]$ be a process. In other words, $\rho[\sigma]$ is an active prefix spawn at the least depth in $\mathcal{K}_0[\rho[\sigma].Q]$. Then there exists a spawn binding $\sigma'$, and an evaluation context $\mathcal{K}_1[\cdot]$ which is free of $(\rho[\sigma_1].[\cdot])$ sub-contexts for any $\sigma_1$, such that*

$$\mathcal{K}_0[\rho[\sigma].Q] \longrightarrow^* \rho[\sigma'].\mathcal{K}_1[Q] \qquad \textit{and} \qquad \mu(\mathcal{K}_0[\rho[\sigma].Q]) > \mu(\rho[\sigma'].\mathcal{K}_1[Q]).$$

Proof. We first show that the last condition follows from the previous ones. Note that $\mu(\rho[\sigma'].\mathcal{K}_1[Q]) = \mathrm{skel}(\mathcal{K}_1[Q])$. We then consider two situations. If $\mathcal{K}_0[\rho[\sigma].Q]$ does not have a spawn prefix at the top level, then

$$\mu(\mathcal{K}_0[\rho[\sigma].Q]) = \mathrm{skel}(\mathcal{K}_0[\rho[\sigma].Q]) > \mathrm{skel}(\mathcal{K}_1[Q]),$$

as the later process has less spawn prefixes. On the other hand, if $\mathcal{K}_0[\rho[\sigma].Q]$ begins with a spawn prefix at the top level, that prefix cannot be $\rho[\sigma]$ itself, as $\mathcal{K}_0$ is non-empty. Then the process $\mathcal{K}_0[\rho[\sigma].Q]$ is of the form $\rho[\sigma'].\mathcal{K}[\rho[\sigma].Q]$, and we have

$$\mu(\rho[\sigma'].\mathcal{K}[\rho[\sigma].Q]) = \mathrm{skel}(\mathcal{K}[\rho[\sigma].Q]) > \mathrm{skel}(\mathcal{K}_1[Q]).$$

Thus, we only need to find an adequate context $\mathcal{K}_1[\cdot]$ and establish the reduction. We prove this by induction on the size of the evaluation context $\mathcal{K}_0[\cdot]$. We do a case analysis on the "tail" of the evaluation context.

- **Case $\mathcal{K}_0$ is of the form $\mathcal{K}_0'[\rho[\sigma'].[\cdot]]$.** If $\mathcal{K}_0$ contains the $\rho[\sigma'].[\cdot]$, then by our assumption, it is on the top level. That means that $\mathcal{K}_0'$ is empty. We then apply the reduction RED-SPAWN-MERGE:

$$\rho[\sigma'].\rho[\sigma].Q \longrightarrow \rho[\sigma' \ltimes \sigma].Q.$$

  Then pick $\mathcal{K}_1$ to be empty.
- **Case $\mathcal{K}_0[\rho[\sigma].Q]$ is of the form $\mathcal{K}_0'[(\nu x).(P \mid \rho[x \mapsto x_1, x_2].Q)]$.** We then have a reduction

$$\mathcal{K}_0'[(\nu x).(P \mid \rho[x \mapsto x_1, x_2].Q)] \longrightarrow \mathcal{K}_0'[\rho[z \mapsto z_1, z_2].(\nu x_1).(P^{(1)} \mid (\nu x_2).(P^{(2)} \mid Q))],$$

  if $\mathrm{fn}(P) = \{x, z\}$.
  If $\mathcal{K}_0'$ is empty, then we are done. If it is not, then by the induction hypothesis we then have

$$\mathcal{K}_0'[\rho[z \mapsto z_1, z_2].(\nu x_1).(P^{(1)} \mid (\nu x_2).(P^{(2)} \mid Q))] \longrightarrow^* \\ \rho[\sigma'].\mathcal{K}_1[(\nu x_1).(P^{(1)} \mid (\nu x_2).(P^{(2)} \mid Q))],$$

  which we chain with the original RED-SPAWN reduction.
  Other cases are handled similarly. □

Theorem 3.4. *If $\Delta \vdash P :: z : A$ is a typed process, then $P$ is weakly normalizable.*

Proof. We give a normalization procedure as follows. Given a process $P$, we consider its possible reductions, and apply them in order that would decrease the measure $\mu$. We repeat this until we reach a normal form. Since the measure is strictly decreasing, this procedure will terminate by Lemma B.16.

Thanks to Lemma B.18 we can consider possible reductions of $P$ up to congruence. Let us consider which reductions can apply to $P$.

- **Case RED-SPAWN, RED-SPAWN-L, RED-SPAWN-R, or RED-SPAWN-MERGE.** In that case we find a spawn prefix, involved with such a reduction, with the least depth. Then this spawn prefix will satisfy the conditions of Lemma B.20, and we pull out this active prefix upfront, decreasing the measure.
- **Case communication or forwarder reductions.** In that case we apply that exact reduction, which by Lemma B.19 will decrease the measure. □

$$\textsc{N-id}\quad x : A \vdash x : A \qquad \textsc{N-}\equiv\ \frac{\Delta \vdash M : A \qquad \Delta \equiv \Gamma}{\Gamma \vdash M : A} \qquad \textsc{N-W}\ \frac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta\,;\Delta') \vdash M : A} \qquad \textsc{N-C}\ \frac{\Gamma(\Delta^{(1)}\,;\Delta) \vdash M : A}{\Gamma(\Delta) \vdash M[\mathrm{fn}(\Delta)/\mathrm{fn}(\Delta^{(1)})] : A}$$

$${-\!\!*}\mathrm{I}\ \frac{\Delta\,, x : A \vdash M : B}{\Delta \vdash \lambda x.\, M : A \mathrel{-\!\!*} B} \qquad {\to}\mathrm{I}\ \frac{\Delta\,; x : A \vdash M : B}{\Delta \vdash \alpha x.\, M : A \to B} \qquad {-\!\!*}\mathrm{E}\ \frac{\Delta_1 \vdash M : A \mathrel{-\!\!*} B \qquad \Delta_2 \vdash N : A}{\Delta_1\,, \Delta_2 \vdash M\,N : B}$$

$${\to}\mathrm{E}\ \frac{\Delta_1 \vdash M : A \to B \qquad \Delta_2 \vdash N : A}{\Delta_1\,; \Delta_2 \vdash M@N : B} \qquad \mathbf{1}_{\mathrm{m}}\mathrm{I}\quad \emptyset_{\mathrm{m}} \vdash ()_{\mathrm{m}} : \mathbf{1}_{\mathrm{m}} \qquad \mathbf{1}_{\mathrm{a}}\mathrm{I}\quad \emptyset_{\mathrm{a}} \vdash ()_{\mathrm{a}} : \mathbf{1}_{\mathrm{a}} \qquad \mathbf{1}_{\mathrm{m}}\mathrm{E}\ \frac{\Delta \vdash M : \mathbf{1}_{\mathrm{m}} \qquad \Gamma(\emptyset_{\mathrm{m}}) \vdash N : A}{\Gamma(\Delta) \vdash \mathsf{let}\ ()_{\mathrm{m}} = M \,\mathsf{in}\, N : A}$$

$$\mathbf{1}_{\mathrm{a}}\mathrm{E}\ \frac{\Delta \vdash M : \mathbf{1}_{\mathrm{a}} \qquad \Gamma(\emptyset_{\mathrm{a}}) \vdash N : A}{\Gamma(\Delta) \vdash \mathsf{let}\ ()_{\mathrm{a}} = M \,\mathsf{in}\, N : A} \qquad {*}\mathrm{I}\ \frac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1\,, \Delta_2 \vdash \langle M, N\rangle : A * B} \qquad {\wedge}\mathrm{I}\ \frac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1\,; \Delta_2 \vdash (M, N) : A \wedge B}$$

$${*}\mathrm{E}\ \frac{\Delta \vdash M : A * B \qquad \Gamma(x : A\,, y : B) \vdash N : C}{\Gamma(\Delta) \vdash \mathsf{let}\ \langle x, y\rangle = M \,\mathsf{in}\, N : C} \qquad {\wedge}\mathrm{E}\ \frac{\Delta \vdash M : A_1 \wedge A_2 \qquad i \in \{0, 1\}}{\Delta \vdash \pi_i(M) : A_i}$$

$${\vee}\mathrm{I}\ \frac{\Delta \vdash N : A_i \qquad i \in \{1, 2\}}{\Delta \vdash \mathbf{in}_i(N) : A_1 \vee A_2} \qquad {\vee}\mathrm{E}\ \frac{\Delta \vdash M : A_1 \vee A_2 \qquad \Gamma(x_1 : A_1) \vdash N_1 : C \qquad \Gamma(x_2 : A_2) \vdash N_2 : C}{\Gamma(\Delta) \vdash \mathsf{case}\, M \,\mathsf{of}\, \mathbf{in}_1(x_1) \Rightarrow N_1 \,\mathsf{or}\, \mathbf{in}_2(x_2) \Rightarrow N_2}$$

$$\textsc{N-cut}\ \frac{\Delta \vdash M : A \qquad \Gamma(x : A) \vdash N : C}{\Gamma(\Delta) \vdash N[M/x] : C}$$

Fig. 10. Typing system for $\alpha\lambda$-calculus.

## C ENCODING THE $\alpha\lambda$-CALCULUS

The $\alpha\lambda$-calculus type system follows the presentations of $\alpha\lambda$-calculus by O'Hearn [2003] and Pym [2002, Chapter 2], but we adjusted the elimination rule for additive units to match the corresponding rule for multiplicative units. For reasons of space, we have omitted products and coproducts from Section 4; these are present here. The type system is given in Figure 10. Note that the N-cut is admissible (cf. [O'Hearn 2003]). We consider call-by-name reduction strategy, the reduction relation for which is given in Figure 11.

The translation function $\mathcal{T}_z(-)$ is defined by recursion on the typing derivation and is given in Figures 12 and 13.

### C.1 Operational correspondence

We split the proof of completeness into two parts. First, we show that if a term can reduce, then this reduction is matched by the translated process, and the resulting term and process diverge up to substitution lifting.

Lemma C.1 (Basic completeness). *Given $\Delta \vdash M : A$, if $M \rightarrowtail N$, then there exists $Q$ such that $\mathcal{T}_z(M) \longrightarrow^* Q \triangleright N$.*

— Primitive reductions

RED-BETA-$\lambda$
$$(\lambda x.\,M)\ N \rightarrowtail M[N/x]$$

RED-BETA-$\alpha$
$$(\alpha x.\,M)@N \rightarrowtail M[N/x]$$

RED-PROJ
$$\frac{}{\pi_i(M_1, M_2) \rightarrowtail M_i}\ i \in \{1,2\}$$

RED-UNITM
$$\mathsf{let}\ ()_{\mathsf{m}} = ()_{\mathsf{m}}\ \mathsf{in}\ M \rightarrowtail M$$

RED-UNITA
$$\mathsf{let}\ ()_{\mathsf{a}} = ()_{\mathsf{a}}\ \mathsf{in}\ M \rightarrowtail M$$

RED-PAIR
$$\mathsf{let}\ \langle x, y\rangle = \langle M_1, M_2\rangle\ \mathsf{in}\ N \rightarrowtail N[M_1/x,\ M_2/y]$$

RED-CASE
$$\frac{}{\mathsf{case}\ \mathbf{in}_i(M)\ \mathsf{of}\ \mathbf{in}_1(x_1) \Rightarrow N_1\ \mathsf{or}\ \mathbf{in}_2(x_2) \Rightarrow N_2 \rightarrowtail N_i[M/x_i]}\ i \in \{1,2\}$$

— Lifted reductions

$$\frac{M \rightarrowtail M'}{M\ N \rightarrowtail M'\ N} \qquad \frac{M \rightarrowtail M'}{M@N \rightarrowtail M'@N} \qquad \frac{M \rightarrowtail M'}{\mathsf{let}\ p = M\ \mathsf{in}\ N \rightarrowtail \mathsf{let}\ p = M'\ \mathsf{in}\ N} \qquad \frac{M \rightarrowtail M'}{\pi_i M \rightarrowtail \pi_i M'}$$

$$\frac{M \rightarrowtail M'}{\mathsf{case}\ M\ \mathsf{of}\ \mathbf{in}_1(x_1) \Rightarrow N_1\ \mathsf{or}\ \mathbf{in}_2(x_2) \Rightarrow N_2 \rightarrowtail \mathsf{case}\ M'\ \mathsf{of}\ \mathbf{in}_1(x_1) \Rightarrow N_1\ \mathsf{or}\ \mathbf{in}_2(x_2) \Rightarrow N_2}$$

Fig. 11. Reduction rules for $\alpha\lambda$-calculus.

| $\alpha\lambda$-calculus typing of $M_0$ | $\pi$BI encoding $\mathcal{T}_z(M_0)$ |
|---|---|
| $x : A \vdash x : A$ | $x : A \vdash [z \leftarrow x] :: z : A$ |
| $\dfrac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta\,;\Delta') \vdash M : A}$ | $\dfrac{\Gamma(\Delta) \vdash \mathcal{T}_z(M) :: z : A}{\Gamma(\Delta\,;\Delta') \vdash \boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].\mathcal{T}_z(M) :: z : A}$ |
| $\dfrac{\Gamma(\Delta^{(1)}\,;\Delta^{(2)}) \vdash M : A}{\Gamma(\Delta) \vdash M[\Delta/\Delta^{(1)},\ \Delta/\Delta^{(2)}] : A}$ | $\dfrac{\Gamma(\Delta^{(1)}\,;\Delta^{(2)}) \Vdash \mathcal{T}_z(M) :: z : A}{\Gamma(\Delta) \vdash \boldsymbol{\rho}[x \mapsto x_1, x_2 \mid x \in \mathrm{fn}(\Delta)].\mathcal{T}_z(M) :: z : A}$ |
| $\dfrac{\Delta\,, x : A \vdash M : B}{\Delta \vdash \lambda x.\,M : A \twoheadrightarrow B}$ | $\dfrac{\Delta\,, x : A \vdash \mathcal{T}_z(M) :: z : B}{\Delta \vdash z(x).\mathcal{T}_z(M) :: z : A \twoheadrightarrow B}$ |
| $\dfrac{\Delta\,; x : A \vdash M : B}{\Delta \vdash \alpha x.\,M : A \rightarrow B}$ | $\dfrac{\Delta\,; x : A \vdash \mathcal{T}_z(M) :: z : B}{\Delta \vdash z(x).\mathcal{T}_z(M) :: z : A \rightarrow B}$ |
| $\dfrac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1\,, \Delta_2 \vdash \langle M, N\rangle : A * B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_y(M) :: y : A \qquad \Delta_2 \vdash \mathcal{T}_z(N) :: z : B}{\Delta_1\,, \Delta_2 \vdash \overline{z}[y].(\mathcal{T}_y(M) \mid \mathcal{T}_z(N)) :: z :: A * B}$ |
| $\dfrac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1\,; \Delta_2 \vdash (M, N) : A \wedge B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_y(M) :: y : A \qquad \Delta_2 \vdash \mathcal{T}_z(N) :: z : B}{\Delta_1\,; \Delta_2 \vdash \overline{z}[y].(\mathcal{T}_y(M) \mid \mathcal{T}_z(N)) :: z :: A \wedge B}$ |

Fig. 12. Translation from $\alpha\lambda$-calculus to $\pi$BI (1/2).

Proof. By induction on the derivation of $M \rightarrowtail N$. There are six base cases:

- **Case** RED-BETA-$\lambda$. We have $(\lambda x.\, M)\; N \rightarrowtail M[N/x]$. We then have

$$\begin{aligned} \mathcal{T}_z((\lambda x.\, M)\; N) &= (\boldsymbol{\nu} y).(y(x).\mathcal{T}_y(M) \mid \overline{y}[x].(\mathcal{T}_x(N) \mid [z \leftarrow y])) \\ &\longrightarrow (\boldsymbol{\nu} y).((\boldsymbol{\nu} x).(\mathcal{T}_x(N) \mid \mathcal{T}_y(M)) \mid [z \leftarrow y]) \\ &\longrightarrow (\boldsymbol{\nu} x).(\mathcal{T}_x(N) \mid \mathcal{T}_z(M)) \triangleright M[N/x]. \end{aligned}$$

- **Case** RED-BETA-$\alpha$. Analogous to Case RED-BETA-$\lambda$.
- **Case** RED-PROJ. We have $\pi_i(M_1, M_2) \rightarrowtail M_i$ for $i \in \{1,2\}$. Let $i' \in \{1,2\} \setminus \{i\}$.

$$\begin{aligned} \mathcal{T}_z(\pi_i(M_1, M_2)) &= (\boldsymbol{\nu} x_1).(\overline{x_1}[x_2].(\mathcal{T}_{x_2}(M_2) \mid \mathcal{T}_{x_1}(M_1)) \mid x_1(x_2).\boldsymbol{\rho}[x_{i'} \mapsto \emptyset].[z \leftarrow x_i]) \\ &\longrightarrow (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(M_2) \mid \boldsymbol{\rho}[x_{i'} \mapsto \emptyset].[z \leftarrow x_i])) \\ &\longrightarrow \boldsymbol{\rho}[z \mapsto \emptyset \mid z \in \mathrm{fn}(M_{i'})].(\boldsymbol{\nu} x_i).(\mathcal{T}_{x_i}(M_i) \mid [z \leftarrow x_i]) \\ &\longrightarrow \boldsymbol{\rho}[z \mapsto \emptyset \mid z \in \mathrm{fn}(M_{i'})].\mathcal{T}_z(M_i) \\ &\triangleright\; M_i \end{aligned}$$

- **Case** RED-UNITM. We have let $()_\mathsf{m} = ()_\mathsf{m}$ in $M \rightarrowtail M$.

$$\begin{aligned} \mathcal{T}_z(\mathsf{let}\ ()_\mathsf{m} = ()_\mathsf{m}\ \mathsf{in}\ M \longrightarrow M) &= (\boldsymbol{\nu} x).(\overline{x}\langle\rangle \mid x().\mathcal{T}_z(M)) \\ &\longrightarrow \mathcal{T}_z(M) \\ &\triangleright\; M \end{aligned}$$

- **Case** RED-PAIR. We have let $\langle x, y\rangle = \langle M_1, M_2\rangle$ in $N \rightarrowtail N[M_1/x,\ M_2/y]$.

$$\begin{aligned} \mathcal{T}_z(\mathsf{let}\ \langle x, y\rangle = \langle M_1, M_2\rangle\ \mathsf{in}\ N) &= (\boldsymbol{\nu} x).(\overline{x}[y].(\mathcal{T}_y(M_2) \mid \mathcal{T}_x(M_1)) \mid x(y).\mathcal{T}_z(N)) \\ &\longrightarrow (\boldsymbol{\nu} x).(\mathcal{T}_x(M_1) \mid (\boldsymbol{\nu} y).(\mathcal{T}_y(M_2) \mid \mathcal{T}_z(N))) \\ &\equiv (\boldsymbol{\nu} y).(\mathcal{T}_y(M_2) \mid (\boldsymbol{\nu} x).(\mathcal{T}_x(M_1) \mid \mathcal{T}_z(N))) \\ &\triangleright\; N[M_1/x,\ M_2/y] \end{aligned}$$

- **Case** RED-CASE. We have case $\mathbf{in}_i(M)$ of $\mathbf{in}_1(x_1) \Rightarrow N_1$ or $\mathbf{in}_2(x_2) \Rightarrow N_2 \rightarrowtail N_i[M/x_i]$ for $i \in \{1,2\}$.

$$\begin{aligned} &\mathcal{T}_z(\mathsf{case}\ \mathbf{in}_i(M)\ \mathsf{of}\ \mathbf{in}_1(x_1) \Rightarrow N_1\ \mathsf{or}\ \mathbf{in}_2(x_2) \Rightarrow N_2) \\ =\; &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(\mathbf{in}_i(M)) \mid x_1 \triangleright \mathsf{case}(\mathcal{T}_z(N_1), (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2)))) \end{aligned}$$

There are two cases for $i \in \{1,2\}$.

- **Case** $i = 1$.

$$\begin{aligned} &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(\mathbf{in}_1(M)) \mid x_1 \triangleright \mathsf{case}(\mathcal{T}_z(N_1), (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2)))) \\ =\; &(\boldsymbol{\nu} x_1).(x_1 \triangleleft \mathsf{inl}.\mathcal{T}_{x_1}(M) \mid x_1 \triangleright \mathsf{case}(\mathcal{T}_z(N_1), (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2)))) \\ \longrightarrow\; &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M) \mid \mathcal{T}_z(N_1)) \\ \triangleright\; &N_1[M/x_1] \end{aligned}$$

- **Case** $i = 2$.

$$\begin{aligned} &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(\mathbf{in}_2(M)) \mid x_1 \triangleright \mathsf{case}(\mathcal{T}_z(N_1), (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2)))) \\ =\; &(\boldsymbol{\nu} x_1).(x_1 \triangleleft \mathsf{inr}.\mathcal{T}_{x_1}(M) \mid x_1 \triangleright \mathsf{case}(\mathcal{T}_z(N_1), (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2)))) \\ \longrightarrow\; &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M) \mid (\boldsymbol{\nu} x_2).([x_2 \leftarrow x_1] \mid \mathcal{T}_z(N_2))) \\ \longrightarrow\; &(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M) \mid \mathcal{T}_z(N_2)[x_1/x_2]) \\ \equiv\; &(\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(M) \mid \mathcal{T}_z(N_2)) \\ \triangleright\; &N_2[M/x_2] \end{aligned}$$

The inductive cases all concern the lifted reductions. Each case is analogous, so we only detail the arbitrarily chosen case of reduction lifting under $\lambda$-application. Assume $M \rightarrowtail M'$. We have $M\ N \rightarrowtail M'\ N$. By the IH, $\mathcal{T}_y(M) \longrightarrow^* P \triangleright M'$. Hence, assuming $M' = M''[N_1/x_1, \ldots, N_n/x_n]$, we have $P \equiv (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid \mathcal{T}_y(M'')) \ldots)$. Moreover,

$$M'\ N = (M''[N_1/x_1, \ldots, N_n/x_n])\ N = (M''\ N)[N_1/x_1, \ldots, N_n/x_n].$$

We have the following:

$$\begin{array}{rcl} \mathcal{T}_z(M\ N) & = & (\boldsymbol{\nu} y).(\mathcal{T}_y(M) \mid y(w).\mathcal{T}_z(N)) \\ & \longrightarrow^* & (\boldsymbol{\nu} y).((\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid \mathcal{T}_y(M'')) \ldots) \mid y(w).\mathcal{T}_z(N)) \\ & \equiv & (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid (\boldsymbol{\nu} y).(\mathcal{T}_y(M'') \mid y(w).\mathcal{T}_z(N))) \ldots) \\ & \triangleright & M'\ N \end{array}$$

□

The statement of Lemma C.1 cannot be chained to form a simulation diagram, since the premise does not start with the substitution relation as in the result. The full version of completeness starts with a term and a process that are related via substitution lifting:

Theorem 4.3 (Completeness). *Given $\Delta \vdash M : A$ and $\Delta \vdash P :: z : A$ such that $P \triangleright M$, if $M \rightarrowtail N$, then there exists $Q$ such that $P \longrightarrow^* Q \triangleright N$.*

Proof. Since $P \triangleright M$, we can write the latter as $M'[N_1/x_1, \ldots, N_n/x_n]$. We then consider two cases, depending on whether the reduction $M \rightarrowtail N$ already happens in $M'$, or whether this reduction is triggered by one of the substitutions.

In the former case we already have a reduction $M' \rightarrowtail N'$ that is "lifted" to the reduction $M'[N_1/x_1, \ldots, N_n/x_n] \rightarrowtail N'[N_1/x_1, \ldots, N_n/x_n]$. We can then appeal directly to the previous lemma to obtain a process $Q$ such that $\mathcal{T}_z(M') \longrightarrow^* Q \triangleright N'$. Then,

$$\begin{array}{l} (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1 \mid \ldots (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid \mathcal{T}_z(M')) \ldots) \longrightarrow^* \\ (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1 \mid \ldots (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(N_n) \mid Q) \ldots) \triangleright N'[N_1/x_1, \ldots, N_n/x_n]. \end{array}$$

In the second case, the reduction in the term is only enabled after some substitution $[N_i/x_i]$ is performed. The idea is to reduce this to the first case, by explicitly performing the substitution $[N_i/x_i]$ in the corresponding processes.

If $[N_i/x_i]$ is the substitution that enables the reduction $M'[N_1/x_1, \ldots, N_n/x_n] \rightarrowtail N$, then the variable $x_i$ is located at a head position in the term $M'$. This means that in the translation, the corresponding process $\mathcal{T}_c(x_i)$ will not occur under an input/output prefix, which will allow us to eagerly perform the substitution by using the forwarder reduction, combined with the structural congruences and red-spawn-r.

Let us demonstrate what we mean by an example. Suppose that $M = (x_1\ M'')[N_1/x_1, N_2/x_2]$ and $N_1 = \lambda a. T$. Clearly, in this case the beta reduction is enabled only after the substitution. The corresponding substitution-lifted process can reduce as follows:

$$\begin{array}{l} (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(N_2) \mid (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid \mathcal{T}_z(x_1\ M''))) = \\ (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(N_2) \mid (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid (\boldsymbol{\nu} c).(\mathcal{T}_c(x_1) \mid \overline{c}[b].(\mathcal{T}_b(M'') \mid [z \leftarrow c])))) = \\ (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(N_2) \mid (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid (\boldsymbol{\nu} c).([c \leftarrow x_1] \mid \overline{c}[b].(\mathcal{T}_b(M'') \mid [z \leftarrow c])))) \longrightarrow \\ (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(N_2) \mid (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(N_1) \mid \overline{x_1}[b].(\mathcal{T}_b(M'') \mid [z \leftarrow x_1]))) = \\ (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(N_2) \mid \mathcal{T}_z(N_1\ M'')) \quad \triangleright \quad (N_1\ M'')[N_2/x_2]. \end{array}$$

The forwarder reduction in that sequence correspond to explicitly performing the substitution $[N_1/x_1]$. After that we get a term $(N_1\ M'')[N_2/x_2]$ in which the reduction is enabled prior to the substitution, thus leaving us with the scenario from the case of this theorem. □

Theorem 4.5 (Soundness). *Given $\Delta \vdash P \triangleright M :: z : A$, if $P \longrightarrow^* Q$, then there exist $N$ and $R$ such that $M \hookrightarrow^* N$ and $Q \longrightarrow^* R \triangleright N$.*

Proof. By definition, $P \equiv \boldsymbol{\rho}[\sigma_s]. \ldots \boldsymbol{\rho}[\sigma_1].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid \mathcal{T}_z(M')) \ldots)$ where $M = M'[M_1/x_1, \ldots, M_n/x_n][\sigma_1, \ldots, \sigma_s]$. Let us consider possible reductions of $P$. First, each parallel subprocess of $P$ may reduce internally. Second, one of the subprocesses may be a forwarder, in which case a forwarder reduction is applicable. Third, one of the subprocesses may start with a spawn prefix, which can interact with the cuts. Note that no message-passing communication between the subprocesses of $P$ is possible, as follows from the definition of the translation. We discuss each possible case:

- $\mathcal{T}_{x_i}(M_i)$ for $i \in [1, n]$ reduces internally, i.e., $\mathcal{T}_{x_i}(M_i) \longrightarrow Q_i$. We apply induction on the derivation of $\Delta \vdash M : A$. Clearly, $\mathcal{T}_{x_i}(M_i) \triangleright M_i$. Since the typing derivation of $M_i$ is a subderivation of the typing derivation of $M$, the IH applies: there exist $N_i$ and $R_i$ such that $M_i \hookrightarrow^* N_i$ and $Q_i \longrightarrow^* R_i \triangleright N_i$.
  Let $Q' = \boldsymbol{\rho}[\sigma_s]. \ldots \boldsymbol{\rho}[\sigma_1].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) | \ldots (\boldsymbol{\nu} x_i).(Q_i | \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) | \mathcal{T}_z(M')) \ldots) \ldots)$; we have $P \longrightarrow Q'$. From $Q'$, all the reductions that were possible from $P$ are still possible. However, these reductions are all independent, so we postpone all but further reductions of $Q'$.
  Let $R = \boldsymbol{\rho}[\sigma_s]. \ldots \boldsymbol{\rho}[\sigma_1].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_i).(R_i \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid \mathcal{T}_z(M')) \ldots) \ldots)$; we have $Q' \longrightarrow^* R$.
  By definition, $R_i \equiv \boldsymbol{\rho}[\sigma'_t]. \ldots \boldsymbol{\rho}[\sigma'_1].(\boldsymbol{\nu} y_m).(\mathcal{T}_{y_m}(L_n) | \ldots (\boldsymbol{\nu} y_1).(\mathcal{T}_{y_1}(L_1) | \mathcal{T}_{x_i}(N'_i)) \ldots)$ where $N_i = N'_i[L_1/y_1, \ldots, L_m/y_m][\sigma'_1, \ldots, \sigma'_t]$. Let $M_0 = M'[M_1/x_1, \ldots, N_i/x_i, \ldots, M_n/x_n]$; we have $M \hookrightarrow^* M_0$. Due to the shape of $R_i$, which includes substitutions, weakenings, and contractions in $N_i$, $R$ is not yet of a shape that we can relate to $M_0$.
  First, we have to move the spawn prefixes in $R_i$ to the sequence of spawn prefixes at the beginning of $R$. The procedure depends on whether there are $x_{i+1}, \ldots, x_n$ that are weakened or contracted by the spawn prefixes in $R_i$. This is largely analogous to the latter cases of spawn prefixes commuting and interacting, so here we assume that no weakening or contraction takes place. By typability, none of the substitutions in $N_i$ touch the variable $x_i$, so we can commute the cuts in $R_i$ past the cut on $x_i$ in $R$. Let $R' = \boldsymbol{\rho}[\sigma_s]. \ldots \boldsymbol{\rho}[\sigma_1].\boldsymbol{\rho}[\sigma'_t]. \ldots \boldsymbol{\rho}[\sigma'_1].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) | \ldots (\boldsymbol{\nu} y_m).(\mathcal{T}_{y_m}(L_m) | \ldots (\boldsymbol{\nu} y_1).(\mathcal{T}_{y_1}(L_1) | (\boldsymbol{\nu} x_i).(\mathcal{T}_{x_i}(N'_i) | \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) | \mathcal{T}_z(M')) \ldots O)) \ldots) \ldots)$.
  We have $R \longrightarrow^* R'$. Moreover, $M_0 = M'[M_1/x_1, \ldots, N'_i/x_i, L_1/y_1, \ldots, L_m/y_m, \ldots, M_n/x_n][\sigma_1, \ldots, \sigma_s, \sigma'_1, \ldots$ and thus $R' \triangleright M_0$. This proves the thesis.
- $\mathcal{T}_z(M')$ reduces internally. We apply induction on the derivation of $\Delta \vdash M : A$ ($\mathrm{IH}_1$); there is a case for typing rule, although not all cases may yield a reduction in $P$. In each case, we additionally apply induction on the number $k$ of reductions from $P$ to $Q$ ($\mathrm{IH}_2$), i.e., $P \longrightarrow^k Q$. Depending on the shape of $P$, and relying on the independence of reductions, we then isolate $k'$ reductions $P \longrightarrow^{k'} Q'$ such that $Q' \triangleright N'$ and $M \hookrightarrow N'$ (where $k'$ may be different in each case). We then have $Q' \longrightarrow^{k-k'} Q$, so it follows from $\mathrm{IH}_2$ that $N' \hookrightarrow^* N$ and $Q' \longrightarrow^* R$ such that $R \triangleright N$.
  Note that applications of $\mathrm{IH}_1$ yield processes with spawn prefixes that need to be commuted past cuts to bring them to the front of the process, while some of them apply weakening or contraction when meeting certain cuts. We explain such procedures in the latter cases of this proof, so here we assume that $\mathrm{IH}_1$ yields processes without spawn prefixes.
  - **Case** N-id**.** We have $M' = y$ and $\mathcal{T}_z(M') = [z \leftarrow y]$; no reductions are possible.
  - **Case** N-$\equiv$**.** The thesis follows from $\mathrm{IH}_1$ directly.
  - **Case** N-W**.** We have $\mathcal{T}_z(M') = \boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].\mathcal{T}_z(M')$. There is only one possibility of reduction: $\boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].\mathcal{T}_z(M') \longrightarrow \boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].Q'$.

By $\mathrm{IH}_1$, there exist $L$ and $R'$ such that $M' \hookrightarrow^* L$ and $Q' \longrightarrow^* R' \triangleright L$. Then

$$R' \equiv (\boldsymbol{\nu} y_m).(\mathcal{T}_{y_m}(L_m) \mid \ldots (\boldsymbol{\nu} y_1).(\mathcal{T}_{y_1}(L_1) \mid \mathcal{T}_z(L'))\ldots)$$

and $L' = L'[L_1/y_1, \ldots, L_m/y_m]$.

Let $R_0 = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid \boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].R'))$; we have $P \longrightarrow^* R_0$. Also, let $M_0 = L'[L_1/y_1, \ldots, L_m/y_m, M_1/x_1, \ldots, M_n/x_n]$; we have $M \hookrightarrow^* M_0$. At this point, $R_0$ is not of appropriate shape to relate it to $M_0$ through substitution lifting, because the weakening spawn prefix is in the middle of the substitutions. There are two possibilities for reduction here: the spawn interacts with one of the cuts on $x_i$, or the spawn commutes past them all. The former is analogous to the case of a spawn prefix in $\mathcal{T}_z(M')$ interacting with a cut, which follows the current case. In the latter case, let $R = \boldsymbol{\rho}[x \mapsto \emptyset \mid x \in \mathrm{fn}(\Delta')].(\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid R'))$. Now, $R_0 \longrightarrow^* R$ and $R \triangleright M_0$, proving the thesis.

- **Case N-C.** Analogous to Case N-W.
- **Case $\twoheadrightarrow$I.** We have $M' = \lambda x.\, M''$ and $\mathcal{T}_z(M') = z(x).\mathcal{T}_z(M'')$; no reductions are possible.
- **Case $\rightarrow$I.** Analogous to Case $\twoheadrightarrow$I.
- **Case $\twoheadrightarrow$E.** We have $M' = L_1\ L_2$ and $\mathcal{T}_z(M') = (\boldsymbol{\nu} x).(\mathcal{T}_x(L_1) \mid \overline{x}[y].(\mathcal{T}_y(L_2) \mid [z \leftarrow x]))$. There are three possible reductions: $\mathcal{T}_x(L_1)$ reduces internally, $\mathcal{T}_x(L_1)$ is prefixed by a spawn which commutes past the restriction on $x$, or the output on $x$ synchronizes with an input on $x$ in $\mathcal{T}_x(L_1)$.
  - $\mathcal{T}_x(L_1)$ reduces internally, i.e., $\mathcal{T}_x(L_1) \longrightarrow Q'$. By $\mathrm{IH}_1$, there exist $N$ and $R'$ such that $L_1 \hookrightarrow^* N$ and $Q' \longrightarrow^* R' \triangleright N$. Then

    $$R' \equiv (\boldsymbol{\nu} y_m).(\mathcal{T}_{y_m}(N_m) \mid \ldots (\boldsymbol{\nu} y_1).(\mathcal{T}_{y_1}(N_1) \mid \mathcal{T}_x(N'))\ldots)$$

    and $N = N'[N_1/y_1, \ldots, N_m/y_m]$. Let

$$R_0 = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid (\boldsymbol{\nu} x).(R' \mid \overline{x}[y].(\mathcal{T}_y(L_2) \mid [z \leftarrow x])))\ldots);$$

    we have $P \longrightarrow^* R_0$. Also, let

    $$\begin{aligned} M_0 &= (N'[N_1/y_1, \ldots, N_m/y_m]\ L_2)[M_1/x_1, \ldots, M_n/x_n] \\ &= (N'\ L_2)[N_1/y_1, \ldots, N_m/y_m, M_1/x_1, \ldots, M_n/x_n] \end{aligned}$$

    we have $M \hookrightarrow^* M_0$. We have $R_0 \equiv (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n)|\ldots(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1)|(\boldsymbol{\nu} y_m).(\mathcal{T}_{y_m}(N_m)| \ldots (\boldsymbol{\nu} y_1).(\mathcal{T}_{y_1}(N_1) \mid (\boldsymbol{\nu} x).(\mathcal{T}_x(N') \mid \overline{x}[y].(\mathcal{T}_y(L_2) \mid [z \leftarrow x])))\ldots))\ldots)$, so $R_0 \triangleright M_0$. This proves the thesis.
  - $\mathcal{T}_x(L_1)$ is prefixed by a spawn which commutes past the restriction on $x$. This case is analogous to the case of a spawn prefix in $\mathcal{T}_x(M')$ commuting past cuts, which follows the current case.
  - The output on $x$ synchronizes with an input on $x$ in $\mathcal{T}_x(L_1)$. By typability, then $L_1 = \lambda y.\, L_1'$ and $\mathcal{T}_x(L_1) = x(y).\mathcal{T}_x(L_1')$. Let $Q_0' = (\boldsymbol{\nu} x).((\boldsymbol{\nu} y).(\mathcal{T}_y(L_2) \mid \mathcal{T}_x(L_1')) \mid [z \leftarrow x])$. Then $\mathcal{T}_x(M') \longrightarrow Q_0'$.

    From $Q_0'$, there may be similar reductions as from $\mathcal{T}_x(M')$, with an additional forwarder reduction possible. All of these reductions are independent, so we postpone all but the forwarder reduction. Let $Q_1' = (\boldsymbol{\nu} y).(\mathcal{T}_y(L_2) \mid \mathcal{T}_z(L_1'))$. Then $Q_0' \longrightarrow Q_1'$.

    Let $Q' = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_1'))$; we have $P \longrightarrow^2 Q'$. Also, let $M_0 = M_1'[L_2/y, M_1/x_1, \ldots, M_n/x_n]$; we have $M \hookrightarrow M_0$. Since $Q' \longrightarrow^{k-2} Q$, the thesis then follows from $\mathrm{IH}_2$.
- **Case $\rightarrow$E.** Analogous to Case $\twoheadrightarrow$E.
- **Case $1_\mathrm{m}$I.** We have $M' = ()_\mathrm{m}$ and $\mathcal{T}_z(M') = \overline{z}\langle\rangle$; no reductions are possible.

- **Case $\mathbf{1}_{\mathsf{a}}$I.** Analogous to Case $\mathbf{1}_{\mathsf{m}}$I.
- **Case $\mathbf{1}_{\mathsf{m}}$E.** We have $M' = \text{let } ()_{\mathsf{m}} = L_1 \text{ in } L_2$ and $\mathcal{T}_z(M') = (\boldsymbol{\nu} x).(\mathcal{T}_x(L_1) \mid x().\mathcal{T}_z(L_2))$. There are three possible reductions: $\mathcal{T}_x(L_1)$ reduces internally, $\mathcal{T}_x(L_2)$ is prefixed by a spawn which commutes past the restriction on $x$, or the empty input on $x$ synchronizes with an empty output on $x$ in $\mathcal{T}_x(L_1)$. The former two subcases are analogous to the similar subcases in Case $\twoheadrightarrow$E. In the latter case, by typability, we have $L_1 = ()_{\mathsf{m}}$ and $\mathcal{T}_x(L_1) = \overline{x}\langle\rangle$. Let $Q_0 = \mathcal{T}_z(L_2)$; we have $\mathcal{T}_z(M') \longrightarrow Q_0$. Let
$$Q' = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_0)\ldots);$$
we have $P \longrightarrow Q'$.
Let $M_0 = L_2[M_1/x_1, \ldots, M_n/x_n]$; we have $M \hookrightarrow M_0$ and $Q' \triangleright M_0$. Since $Q' \longrightarrow^{k-1} Q$, the thesis follows from $\mathrm{IH}_2$.
- **Case $\mathbf{1}_{\mathsf{a}}$E.** Analogous to Case $\mathbf{1}_{\mathsf{m}}$E.
- **Case $*$I.** We have $M' = \langle L_1, L_2\rangle$ and $\mathcal{T}_z(M') = \overline{z}[y].(\mathcal{T}_y(L_1) \mid \mathcal{T}_z(L_2))$; no reductions are possible.
- **Case $\wedge$I.** Analogous to Case $*$I.
- **Case $*$E.** We have $M' = \text{let } \langle x, y\rangle = L_1 \text{ in } L_2$ and $\mathcal{T}_z(M') = (\boldsymbol{\nu} y).(\mathcal{T}_y(L_1) \mid y(x).\mathcal{T}_z(L_2))$. There are three possible reductions: $\mathcal{T}_y(L_1)$ reduces internally, $\mathcal{T}_y(L_1)$ is prefixed by a spawn which commutes past the restriction on $y$, or the input on $y$ synchronizes with an output on $y$ in $\mathcal{T}_y(L_1)$. The former two subcases are analogous to the similar subcases in Case $\twoheadrightarrow$E. In the latter case, by typability, we have $L_1 = \langle K_1, K_2\rangle$ and $\mathcal{T}_y(L_1) = \overline{y}[x].(\mathcal{T}_x(K_1) \mid \mathcal{T}_y(K_2))$. Let $Q_0 = (\boldsymbol{\nu} y).(\mathcal{T}_y(K_2) \mid (\boldsymbol{\nu} x).(\mathcal{T}_x(K_1) \mid \mathcal{T}_z(L_2)))$; we have $\mathcal{T}_z(M') \longrightarrow Q_0$. Let $Q' = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_0)\ldots)$; we have $P \longrightarrow Q'$.
Let $M_0 = L_2[K_1/x,\ K_2/y,\ M_1/x_1,\ \ldots,\ M_n/x_n]$; we have $M \hookrightarrow M_0$ and $Q' \triangleright M_0$. Since $Q' \longrightarrow^{k-1} Q$, the thesis follows from $\mathrm{IH}_2$.
- **Case $\wedge$E.** We have $M' = \pi_i L$ and $\mathcal{T}_z(M') = (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(L) \mid x_2(x_1).\boldsymbol{\rho}[x_{i'} \mapsto \emptyset].[z \leftarrow x_i])$ for $i \in \{1,2\}$ and $i' \in \{1,2\} \setminus \{i\}$. W.l.o.g., let $i = 1$ and $i' = 2$. There are three possible reductions: $\mathcal{T}_{x_2}(L)$ reduces internally, $\mathcal{T}_{x_2}(L)$ is prefixed by a spawn which commutes past the restriction on $x_2$, or the input on $x_2$ synchronizes with an output on $x_2$ in $\mathcal{T}_{x_2}(L)$. The former two subcases are analogous to the similar subcases in Case $\twoheadrightarrow$E.
In the latter case, by typability, we have $L = (L_1, L_2)$ and $\mathcal{T}_{x_2}(L) = \overline{x_2}[x_1].(\mathcal{T}_{x_1}(L_1) \mid \mathcal{T}_{x_2}(L_2))$. Let $Q_0 = (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(L_2) \mid (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(L_1) \mid \boldsymbol{\rho}[x_2 \mapsto \emptyset].[z \leftarrow x_1]))$; we have $\mathcal{T}_z(M') \longrightarrow Q_0$. From $Q_0$, there may be internal reductions of $\mathcal{T}_{x_2}(L_2)$ or $\mathcal{T}_{x_1}(L_1)$, a spawn prefix in $\mathcal{T}_{x_2}(L_2)$ may commute past the restriction on $x_2$, a spawn prefix in $\mathcal{T}_{x_1}(L_1)$ may commute past the restrictions on $x_1$ and $x_2$, and the spawn prefix $\boldsymbol{\rho}[x_2 \mapsto \emptyset]$ may commute past the restriction on $x_1$. All these reductions are independent, so we postpone all but the commute of the spawn prefix $\boldsymbol{\rho}[x_2 \mapsto \emptyset]$. Let $Q_1 = (\boldsymbol{\nu} x_2).(\mathcal{T}_{x_2}(L_2) \mid \boldsymbol{\rho}[x_2 \mapsto \emptyset].(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(L_1) \mid [z \leftarrow x_1]))$; we have $Q_0 \longrightarrow Q_1$.
From $Q_1$ we have the same possible reductions as from $Q_0$, except that there may also be weakening of $x_2$ due to the spawn prefix $\boldsymbol{\rho}[x_2 \mapsto \emptyset]$ interacting with the restriction on $x_2$. Again, we postpone all but the latter reduction. Let $Q_2 = \boldsymbol{\rho}[y \mapsto \emptyset \mid y \in \mathrm{fv}(L_2)].(\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(L_1) \mid [z \leftarrow x_1])$; we have $Q_1 \longrightarrow Q_2$.
From $Q_2$, we again have the reductions that were available from $Q_1$, but also the reduction of the forwarder $[z \leftarrow x_1]$. We postpone all but the latter. Let $Q_3 = \boldsymbol{\rho}[y \mapsto \emptyset \mid y \in \mathrm{fv}(L_2)].\mathcal{T}_z(L_1)$; we have $Q_2 \longrightarrow Q_3$.
Let $Q_0' = (\boldsymbol{\nu} x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\boldsymbol{\nu} x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_3)\ldots)$; we have $P \longrightarrow^4 Q_0'$. Also, let $M_0 = L_1[M_1/x_1, \ldots, M_n/x_n]$, where the resources used by $L_2$ have been weakened. At this point, $Q_0'$ is not of appropriate shape to relate it to $M_0$ through substitution lifting,

because of the spawn prefix in $Q_3$. There are two possibilities for reduction here: the spawn interacts with one of the cuts on $x_i$, or the spawn commutes past them all. The former is analogous to the case of a spawn prefix in $\mathcal{T}_z(M')$ interacting with a cut, which follows the current case. In the latter case, let $Q' = \rho[y \mapsto \emptyset \mid y \in \mathrm{fv}(L_2)].(\nu x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\nu x_1).(\mathcal{T}_{x_1}(M_1) \mid \mathcal{T}_z(L_1))\ldots)$. Now $Q'_0 \longrightarrow^n Q'$ and $Q' \triangleright M_0$.
Since $Q' \longrightarrow^{k-4-n} Q$, the thesis follows from $\mathrm{IH}_2$.

- **Case** $\vee$I. We have $M' = \mathbf{in}_i(N)$ for $i \in \{1,2\}$. Depending on the value of $i$, $\mathcal{T}_z(M') = z \triangleleft \mathsf{inl}.\mathcal{T}_z(N)$ or $\mathcal{T}_z(M') = z \triangleleft \mathsf{inr}.\mathcal{T}_z(N)$. Either way, no reductions are possible.
- **Case** $\vee$E. We have $M' = \mathrm{case}\, N \,\mathrm{of}\, \mathbf{in}_1(y_1) \Rightarrow L_1 \,\mathrm{or}\, \mathbf{in}_2(y_2) \Rightarrow L_2$ and $\mathcal{T}_z(M') = (\nu y_1).(\mathcal{T}_{y_1}(N) \mid y_1 \triangleright \mathrm{case}(\mathcal{T}_z(L_1), (\nu y_2).([y_2 \leftarrow y_1] \mid \mathcal{T}_z(L_2))))$. There are three possible reductions: $\mathcal{T}_{y_1}(N)$ reduces, $\mathcal{T}_{y_1}(N)$ is prefixed by a spawn which commutes past the restriction on $y_1$, or the case on $y_1$ synchronizes with a select on $y_1$ in $\mathcal{T}_{y_1}(N)$. The former two subcases are analgoous to the similar subcases in Case $-\!\!*$E.
  In the latter case, by typability, we have $N = \mathbf{in}_i(N')$ for $i \in \{1,2\}$. The rest of the analysis depends on the value of $i$:
  - **Case** $i = 1$. We have $\mathcal{T}_{y_1}(N) = y_1 \triangleleft \mathsf{inl}.\mathcal{T}_{y_1}(N')$. Let $Q_0 = (\nu y_1).(\mathcal{T}_{y_1}(N') \mid \mathcal{T}_z(L_1))$; we have $\mathcal{T}_z(M') \longrightarrow Q_0$. Let $Q' = (\nu x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\nu x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_0))$; we have $P \longrightarrow Q'$. Also, let $M_0 = L_1[N'/y_1,\ M_1/x_1,\ \ldots,\ M_n/x_n]$; we have $M \hookrightarrow M_0$. Moreover, $Q' \triangleright M_0$. Since $Q' \longrightarrow^{k-1} Q$, the thesis follows from $\mathrm{IH}_2$.
  - **Case** $i = 2$. We have $\mathcal{T}_{y_1}(N) = y_1 \triangleleft \mathsf{inr}.\mathcal{T}_{y_1}(N')$. Let $Q_0 = (\nu y_1).(\mathcal{T}_{y_1}(N') \mid (\nu y_2).([y_2 \leftarrow y_1] \mid \mathcal{T}_z(L_2)))$; we have $\mathcal{T}_z(M') \longrightarrow Q_0$.
    From $Q_0$ several reductions are possible: $\mathcal{T}_{y_1}(N')$ or $\mathcal{T}_z(L_2)$ reduce internally, a spawn prefix in $\mathcal{T}_{y_1}(N')$ commutes past the restriction on $y_1$, a spawn prefix in $\mathcal{T}_z(L_2)$ commutes past or interacts with the restriction on $y_2$, or the forwarder $[y_2 \leftarrow y_1]$ interacts with the restriction on $y_1$. These reductions are independent, so we postpone all but the forwarder reduction. Let $Q_1 = (\nu y_2).(\mathcal{T}_{y_2}(N') \mid \mathcal{T}_z(L_2))$; we have $Q_0 \longrightarrow Q_1$.
    Let $Q' = (\nu x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\nu x_1).(\mathcal{T}_{x_1}(M_1) \mid Q_1)\ldots)$; we have $P \longrightarrow^2 Q'$. Also, let $M_0 = L_2[N'/y_2,\ M_1/x_1,\ \ldots,\ M_n/x_n]$; we have $M \hookrightarrow M_0$. Moreover, $Q' \triangleright M_0$. Since $Q' \longrightarrow^{k-2} Q$, the thesis follows from $\mathrm{IH}_2$.

- A forwarder in $\mathcal{T}_{x_i}(M_i)$ for $i \in [1,n]$ interacts with a cut. We apply induction on the number $k$ of reductions from $P$ to $Q$, i.e., $P \longrightarrow^k Q$.
  We have $\mathcal{T}_{x_i}(M_i) = [x_i \leftarrow y]$ for some $y$. Hence, $M_i = y$. Let $Q' = (\nu x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\nu x_1).(\mathcal{T}_{x_1}(M_1) \mid \mathcal{T}_z(M'[y/x_i]))\ldots)$ without the cut on $x_i$. Since $\mathcal{T}_z(M')[y/x_i] = \mathcal{T}_z(M'[y/x_i])$, we have $P \longrightarrow Q'$. By typability, none of the $M_1,\ldots,M_{i-1}$ can contain the variable $y$, so we have $M'[M_1/x_1,\ \ldots,\ y/x_i,\ \ldots,\ M_n/x_n] = (M'[y/x_i])[M_1/x_1,\ \ldots,\ M_n/x_n]$ where the latter substitutions do not contain the substitution on $x_i$. Hence, $Q' \triangleright M$. Since $Q' \longrightarrow^{k-1} Q$, by the IH, there exist $N$ and $R$ such that $M \hookrightarrow^* N$ and $Q' \longrightarrow^* R \triangleright N$. This proves the thesis.
- A forwarder in $\mathcal{T}_z(M')$ interacts with a cut. We apply induction on the number $k$ of reductions from $P$ to $Q$, i.e., $P \longrightarrow^k Q$.
  We have $\mathcal{T}_z(M') = [z \leftarrow y]$ for some $y$ and there exist $i \in [1,n]$ such that $x_i = y$. Hence, $M' = y$ and $M = y[M_1/x_1,\ \ldots,\ M_i/y,\ \ldots,\ M_n/x_n]$. Let $Q' = (\nu x_n).(\mathcal{T}_{x_n}(M_n) \mid \ldots (\nu x_1).(\mathcal{T}_{x_1}(M_1) \mid \mathcal{T}_z(M_i))\ldots)$. Since $\mathcal{T}_y(M_i)[z/y] = \mathcal{T}_z(M_i)$, we have $P \longrightarrow Q'$. By typability, none of the $M_1,\ldots,M_{i-1}$ can contain the variable $y$, so we have $y[M_1/x_1,\ \ldots,\ M_i/y,\ \ldots,\ M_n/x_n] = M_i[M_1/x_1,\ \ldots,\ M_n/x_n]$ where the latter substitutions do not contain the substitution on $x_i$. Hence, $Q' \triangleright M$. Since $Q' \longrightarrow^{k-1} Q$, by the IH, there exist $N$ and $R$ such that $M \hookrightarrow^* N$ and $Q' \longrightarrow^* R \triangleright N$. This proves the thesis.

- A spawn prefix in $\mathcal{T}_z(M')$ commutes past or interacts with a cut. We apply induction on the number $k$ of reductions from $P$ to $Q$, i.e., $P \longrightarrow^k Q$. The last applied rule in the typing derivation of $\mathcal{T}_z(M')$ is N-W or N-C; the rest of the analysis depends on which:
  - **Case** N-W**.** The rule weakens the variables $y_1, \ldots, y_m$. This case follows by commuting the spawn prefix past cuts on $x_i \notin \{y_1, \ldots, y_m\}$ and performing the weakening when the spawn prefix meets cuts on $x_i \in \{y_1, \ldots, y_m\}$. Other reductions that were possible from $P$ remain possible throughout this process, but they are independent of these reductions, so we can postpone them. After $k'$ steps of reduction, we reach from $P$ a process $Q'$ with the spawn commuted to the top of the process, and some cuts removed. The cuts that were removed concern substitutions of weakened variables, so removing these substitutions from $M$ makes no difference. Similarly, the cuts that were commuted past concern substitutions that are independent of the weakening. Hence, $Q' \triangleright M$. Since $M \hookrightarrow^* M$ and $Q' \longrightarrow^{k-k'} Q$, the thesis follows from the IH.
  - **Case** N-C**.** This case is largely analogous to the case of N-W, except that interactions of the spawn with cuts duplicates the cuts, and commuting past cuts moves the substitutions related to the contraction towards the end of the list of substitutions applied in $M$.
- A spawn prefix in $\mathcal{T}_{x_i}(M_i)$ for $i \in [1, n]$ commutes past a cut. This case is largely analogous to the previous case: first, the spawn can always commute past the cut on $x_i$, after which it may commute further or interact with cuts. □

## D DENOTATIONAL SEMANTICS

The interpretation of $\pi$BI in $\mathrm{Set}^{\wp(\mathrm{Tag})}$ essentially follows the interpretation of BI proofs in doubly-closed categories (DCCs). Forwarders are interpreted as identity morphisms, and cut is interpreted as composition. Suppose we are given morphisms $[\![P]\!] : [\![\Delta]\!] \to [\![A]\!]$ and $[\![Q]\!] : [\![\Gamma(x : A)]\!] \to [\![C]\!]$. First, note that we can write $[\![\Gamma(x : A)]\!]$ as $[\![\Gamma]\!]([\![A]\!])$ where $[\![\Gamma]\!]$ is the obvious endofunctor interpreting bunched contexts. The we interpret the cut $[\![(\nu x).(P \mid Q)]\!]$ as $[\![Q]\!] \circ [\![\Gamma]\!]([\![P]\!])$. Let us spell the cases for some other propositions.

For **separating conjunction** we have:

$$[\![\Delta_1 \,,\, \Delta_2 \vdash \overline{x}[y].(P_1 \mid P_2) :: x : A * B]\!] = [\![P_1]\!] * [\![P_2]\!],$$

where $[\![P_1]\!] * [\![P_2]\!]$ is a monoidal product of two morphisms:

$$[\![\Delta_1]\!] \xrightarrow{[\![P_1]\!]} [\![A]\!]$$

$$[\![\Delta_2]\!] \xrightarrow[{[\![P_2]\!]}]{} [\![B]\!]$$

And for the left rule:

$$[\![\Gamma(x : A * B) \vdash x(y).P :: z : C]\!] = [\![P]\!],$$

since the type $[\![A * B]\!]$ and a context $[\![y : A \,,\, x : B]\!]$ are isomorphic.

For the **magic wand** we interpret the right rule by currying:

$$[\![\Delta \vdash x(y).P :: x : A \mathrel{-\!\!*} B]\!](d)(a) = [\![P]\!](d, a),$$

where $d \in [\![\Delta]\!]$ and $a \in [\![A]\!]$. For the left rule we have:

$$[\![\Gamma(\Delta \,,\, x : A \mathrel{-\!\!*} B) \vdash \overline{x}[y].(P \mid Q) :: z : C]\!] = [\![Q]\!] \circ [\![\Gamma]\!](ev \circ [\![P]\!]),$$

where $ev$ is the evaluation morphism $A * (A \mathrel{-\!\!*} B) \to B$.

For **intuitionistic conjunction and implication** the interpretation is the as above, except we are using the closed Cartesian structure on $\mathrm{Set}^{\wp(\mathrm{Tag})}$.

In order to interpret the **spawn prefix**, we need formulate the semantics of (typed) spawn bindings. Each $\sigma\colon \Delta_1 \rightsquigarrow \Delta_2$ induces a map $[\![\sigma]\!] : [\![\Delta_1]\!] \to [\![\Delta_2]\!]$. Then we interpret the Rule Struct as

$$[\![\boldsymbol{\rho}[\sigma].P]\!] = [\![P]\!] \circ [\![\sigma]\!].$$

**Units and disjunction** are interpreted as usual in Cartesian closed categories.

## D.1 Observational equivalence

Lemma 5.7. *Suppose given a process $P$ such that $\Gamma \vdash P :: z : C$, where $P \not\longrightarrow$ and $P$ does not have any barbs on channels from $\Gamma$. Then $P$ has a barb on the channel $z$.*

Proof. Note that the condition that $P$ does not have any barbs on channels from $\Gamma$ is equivalent to the statement $\mathrm{fn}(\Gamma) \cap \mathrm{an}(P) = \emptyset$. We then prove this reformulated statement by induction on the structure of $P$, using Lemma B.12 when needed.

- If $P$ is a communication prefix, then $P \equiv \alpha(x).P'$. Clearly, $x \in \mathrm{an}(P)$. Since, by assumption, $\mathrm{fn}(\Gamma) \cap \mathrm{an}(P) = \emptyset$, this means that $x \notin \mathrm{fn}(\Gamma)$. By definition, $x \in \mathrm{fn}(P)$, so it follows by Lemma B.4, that $x = z$. Then $P{\downarrow_{\alpha(z)}}$.
- If $P$ is a forwarder, then $P \equiv [x \leftarrow y]$. By typability, $x = z$ and $y \in \mathrm{fn}(\Gamma)$. By definition, $y \in \mathrm{an}(P)$. This contradicts the assumption that $\mathrm{fn}(\Gamma) \cap \mathrm{an}(P) = \emptyset$, so this case is invalid.
- If $P$ is a cut, then $P \equiv (\boldsymbol{\nu} x).(Q \mid_x R)$. We have that $P$ is not live: otherwise, $P \not\longrightarrow$ would contradict Lemma B.12 (Progress). By inversion of typing, we have $\Gamma = \Gamma_1(\Gamma_2)$ where $\Gamma_2 \vdash Q :: x : A$ and $\Gamma_1(x : A) \vdash R :: z : C$.
  By definition, $\mathrm{an}(Q) \subseteq \mathrm{an}(P) \cup \{x\}$ and $\mathrm{fn}(\Gamma_2) \subseteq \mathrm{fn}(\Gamma)$. Moreover, by typability, $x \notin \mathrm{fn}(\Gamma_2)$. Since, by assumption, $\mathrm{fn}(\Gamma) \cap \mathrm{an}(P) = \emptyset$, it follows that $\mathrm{fn}(\Gamma_2) \cap \mathrm{an}(Q) = \emptyset$. Because $P \not\longrightarrow$, we have $Q \not\longrightarrow$. Also, because $P$ is not live, $Q \not\equiv \boldsymbol{\rho}[\sigma].Q'$. Hence, by the IH, $Q{\downarrow_{\alpha'(x)}}$. Clearly, this means that $x \in \mathrm{an}(Q)$.
  By definition, $\mathrm{an}(R) \subseteq \mathrm{an}(P) \cup \{x\}$ and $\mathrm{fn}(\Gamma_1(\,\cdot\,)) \subseteq \mathrm{fn}(\Gamma)$. Since, by assumption, $\mathrm{fn}(\Gamma) \cap \mathrm{an}(P) = \emptyset$, it follows that $\mathrm{fn}(\Gamma_2(x : A)) \cap \mathrm{an}(R) \subseteq \{x\}$. Because $P$ is not live, we have $x \notin \mathrm{an}(Q) \cap \mathrm{an}(R)$. Since $x \in \mathrm{an}(Q)$, it follows that $x \notin \mathrm{an}(R)$. Hence, $\mathrm{fn}(\Gamma_2(x : A)) \cap \mathrm{an}(R) = \emptyset$. Because $P \not\longrightarrow$, we have $R \not\longrightarrow$. Also, because $P$ is not live, $R \not\equiv \boldsymbol{\rho}[\sigma].R'$. Hence, by the IH, $R{\downarrow_{\alpha(z)}}$. Therefore, $P{\downarrow_{\alpha(z)}}$.
- The case when $P$ is a spawn prefix is similar to the previous case. □

| $\alpha\lambda$-calculus typing of $M_0$ | $\pi$BI encoding $\mathcal{T}_z(M_0)$ |
|---|---|
| $\dfrac{\Delta_1 \vdash M : A \mathrel{-\!\!*} B \qquad \Delta_2 \vdash N : A}{\Delta_1 , \Delta_2 \vdash M\,N : B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_x(M) :: x : A \mathrel{-\!\!*} B \qquad \dfrac{\Delta_2 \vdash \mathcal{T}_y(N) :: y : A \qquad x : B \vdash [z \leftarrow x] :: z : B}{x : A \mathrel{-\!\!*} B , \Delta_2 \vdash \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x]) :: z : B}}{\Delta_1 , \Delta_2 \vdash (\nu x).(\mathcal{T}_x(M) \mid \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x])) :: z : B}$ |
| $\dfrac{\Delta_1 \vdash M : A \to B \qquad \Delta_2 \vdash N : A}{\Delta_1 ; \Delta_2 \vdash M\,N : B}$ | $\dfrac{\Delta_1 \vdash \mathcal{T}_x(M) :: x : A \to B \qquad \dfrac{\Delta_2 \vdash \mathcal{T}_y(N) :: y : A \qquad x : B \vdash [z \leftarrow x] :: z : B}{x : A \to B ; \Delta_2 \vdash \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x]) :: z : B}}{\Delta_1 ; \Delta_2 \vdash (\nu x).(\mathcal{T}_x(M) \mid \overline{x}[y].(\mathcal{T}_y(N) \mid [z \leftarrow x])) :: z : B}$ |
| $\dfrac{\Delta \vdash M : A_1 \wedge A_2}{\Delta \vdash \pi_1(M) : A_1}$ | $\dfrac{\Delta \vdash \mathcal{T}_{x_2}(M) :: x_2 : A_1 \wedge A_2 \qquad \dfrac{\dfrac{x_1 : A_1 \vdash [z \leftarrow x_1] :: z : A_1}{x_1 : A_1 ; x_2 : A_2 \vdash \boldsymbol{\rho}[x_2 \mapsto \emptyset].[z \leftarrow x_1] :: z : A_1}}{x : A_1 \wedge A_2 \vdash x_2(x_1).\boldsymbol{\rho}[x_2 \mapsto \emptyset].[z \leftarrow x_1] :: z : A_1}}{\Delta \vdash (\nu x_2).(\mathcal{T}_{x_2}(M) \mid x_2(x_1).\boldsymbol{\rho}[x_2 \mapsto \emptyset].[z \leftarrow x_1]) :: z : A_1}$ |
| $\dfrac{\Delta \vdash M : A_1 * A_2 \qquad \Gamma(x : A_1 , y : A_2) \vdash N : C}{\Gamma(\Delta) \vdash \mathsf{let}\, \langle x, y \rangle = M \,\mathsf{in}\, N : C}$ | $\dfrac{\Delta \vdash \mathcal{T}_y(M) :: y : A_1 * A_2 \qquad \dfrac{\Gamma(x : A_1 , y : A_2) \vdash \mathcal{T}_z(N) :: z : C}{\Gamma(y : A_1 * A_2) \vdash y(x).\mathcal{T}_z(N) :: z : C}}{\Gamma(\Delta) \vdash (\nu y).(\mathcal{T}_y(M) \mid y(x).\mathcal{T}_z(N)) :: z : C}$ |
| $\dfrac{\Delta \vdash N : A_1}{\Delta \vdash \mathbf{in}_1(N) : A_1 \vee A_2}$ | $\dfrac{\Delta \vdash \mathcal{T}_z(N) :: z : A_1}{\Delta \vdash z \triangleleft \mathsf{inl}.\mathcal{T}_z(N) :: z : A_1 \vee A_2}$ |
| $\dfrac{\Delta \vdash M : A_1 \vee A_2 \qquad \Gamma(x_1 : A_1) \vdash N_1 : C \qquad \Gamma(x_2 : A_2) \vdash N_2 : C}{\Gamma(\Delta) \vdash \mathsf{case}\, M \,\mathsf{of}\, \mathbf{in}_1(x_1) \Rightarrow N_1 \,\mathsf{or}\, \mathbf{in}_2(x_2) \Rightarrow N_2}$ | $\dfrac{\Delta \vdash \mathcal{T}_x(M) :: x : A_1 \vee A_2 \; \dfrac{\Gamma(x_1 : A_1) \vdash \mathcal{T}_z(N_1) :: z : C \qquad \Gamma(x_2 : A_2) \vdash \mathcal{T}_z(N_2) :: z : C}{\Gamma(x : A_1 \vee A_2) \vdash x \triangleright \mathsf{case}(\mathcal{T}_z(N_1)[x/x_1], \mathcal{T}_z(N_2)[x/x_2]) :: z : C}}{\Gamma(\Delta) \vdash (\nu x).(\mathcal{T}_x(M) \mid x \triangleright \mathsf{case}(\mathcal{T}_z(N_1)[x/x_1], \mathcal{T}_z(N_2)[x/x_2])) :: z : C}$ |

Fig. 13. Translation from $\alpha\lambda$-calculus to $\pi$BI (2/2).