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

DAN FRUMIN, University of Groningen, The Netherlands
EMANUELE D’OSUALDO, MPI-SWS, Germany
BAS VAN DEN HEUVEL and JORGE A. PÉREZ, 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 \( \text{spawn} \), 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.

CCS Concepts: • Theory of computation → Process calculi; Type theory.

Additional Key Words and Phrases: concurrency, session types, bunched implications, Curry-Howard correspondence

ACM Reference Format:
Dan Frumin, Emanuele D’Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. A Bunch of Sessions: A Propositions-as-Sessions Interpretation of Bunched Implications in Channel-Based Concurrency. Proc. ACM Program. Lang. 6, OOPSLA2, Article 155 (October 2022), 29 pages. https://doi.org/10.1145/3563318

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.

Authors’ addresses: Dan Frumin, d.frumin@rug.nl, University of Groningen, The Netherlands; Emanuele D’Osualdo, dosualdo@mpi-sws.org, MPI-SWS, Saarbrücken, Germany; Bas van den Heuvel, b.van.den.heuvel@rug.nl; Jorge A. Pérez, j.a.perez@rug.nl, University of Groningen, The Netherlands.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2022 Copyright held by the owner/author(s).
2475-1421/2022/10-ART155
https://doi.org/10.1145/3563318
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 \( T = !\text{int}.?\text{string}!.\text{bool}.\text{end} \) (written in the syntax of [Vasconcelos 2012]) describes a protocol that first outputs an integer (\(!\text{int}\)), then inputs a string (\(?\text{string}\)), and finally outputs a boolean (\(!\text{bool}\)). 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 \( x : !\text{int}.?\text{string}!.\text{bool}.\text{end} \) and produces a new resource \( x : ?\text{string}!.\text{bool}.\text{end} \), 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 Intutionistic 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

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 155. Publication date: October 2022.
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 worlds, 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 $\ast$ and implication $\rightarrow$, but extends it by introducing additive conjunction $\land$ 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 $\rho [x \mapsto x_1, x_2]$ denotes a request to the environment to duplicate the service on $x$ into copies on the new channels $x_1$ and $x_2$. Then, $\rho [x \mapsto x_1, x_2].Q$ is a process that first performs the request...
and then behaves as $Q$. The composition of these processes is denoted $(\nu x). (P \parallel \rho[x \mapsto x_1, x_2].Q)$, where ‘\(\parallel\)’ and ‘(\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:

$$(\nu x). (P \parallel \rho[x \mapsto x_1, x_2].Q) \rightarrow (\nu x_1). (P[x_1/x] \parallel (\nu x_2). (P[x_2/x] \parallel Q)).$$

This way, the duplication request leads to the composition of two copies of $P$ (each with an appropriate substitution $[x_1/x]$) 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 into 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 $\ast$ and additive conjunction $\land$ 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 $\ast$ and $\land$: in a process with a session of type $A \ast B$ 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 $A \land B$. 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
P, Q, R ::= \exists y. (P | Q) \quad\text{output} \quad\mid x(y).P \quad\text{input}
| \exists () \quad\text{close} \quad\mid x().P \quad\text{wait}
| x \triangleleft \text{inl}.P \quad\text{left selection} \quad\mid x \triangleright \text{case}(P, Q) \quad\text{branch}
| x \triangleleft \text{inr}.P \quad\text{right selection} \quad\mid [x \leftarrow y] \quad\text{forwarder}
| (\forall x). (P | Q) \quad\text{restriction + parallel} \quad\mid \rho[\sigma].P \quad\text{spawn}

Fig. 1. Syntax of \(\pi\)BI processes.

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 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), \(a, b, c, \ldots, x, y, z \in \text{Name}\) to denote channels. We make use of finite partial functions \(f : A \xrightarrow{\text{fin}} B\). We write \(f(x) = \bot\) if \(f\) is not defined on \(x\). We define \(\text{dom}(f) = \{x \in A \mid f(x) \neq \bot\}\). 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 processes 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 \(x \in \text{fn}(P)\), 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 \(x\).

- **Labelling choice (selection and branching)**: The processes \(x * \text{inl}.P\) and \(x * \text{inr}.P\) select left/right labels over the session at \(x\), respectively. The dual process \(x \triangleright \text{case}(P, Q)\) 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 \(\vec{x}(\emptyset)\) and \(x().P\) (empty output/input, respectively).

- **Structured parallel composition**: Parallel composition, in keeping with πDILL [Caires et al. 2016], is used jointly with restriction. In a process \(\nu x. (P | Q)\) 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 \(\nu x. (P | Q)\).

- **Forwarders**: A process \(x \leftarrow y\) provides a session at \(x\) as a copycat of the session at \(y\).

The key novel construct of πB is the **spawn prefix** \(p[\sigma].P\). It is parametrized by what we call a **spawn binding** \(\sigma\). Spawn bindings, formally defined below, are a unification and generalisation of prefixes like \(p[x \mapsto x_1, x_2]\) (copy the session at \(x\) to \(x_1\) and \(x_2\)) but also \(p[x \mapsto \emptyset]\) (drop the session 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 finite partial function \(\sigma\) : Name \(\rightarrow\) \(\emptyset\) (Name) is a **spawn binding** if:

\[
\begin{align*}
\forall x, y \in \text{dom}(\sigma) & : x \neq y \Rightarrow \sigma(x) \cap \sigma(y) = \emptyset, \text{ and} \\
\forall x \in \text{dom}(\sigma) & : \text{dom}(\sigma) \cap \sigma(x) = \emptyset.
\end{align*}
\]

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

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

**Free and bound names**. Except for the new spawn construct, the notion of free and bound names is standard: the processes \(\vec{x}(y). (P \mid Q), x(y).P\), and \(\nu y. (P \mid Q)\) all bind \(y\). For the spawn prefix, the situation is a bit different. Given a set of names \(X\), a spawn \(p[x \mapsto X].P\) 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, \(\text{fn}(p[\sigma].P) = (\text{fn}(P) \setminus \text{restr}(\sigma)) \cup \text{dom}(\sigma)\).

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 \textsc{congr-assoc-r} and \textsc{cong-assoc-l} and Rule \textsc{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 \(\vec{x}(y). (P \mid Q)\) 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 \((\forall x). (P \mid Q)\) it is important that \(P\) provides the session that governs \(x\),
CONG-ASSOC-L

\((\forall x). (P \mid x (\forall y). (Q \mid y R)) \equiv (\forall y). (Q \mid y (\forall x). (P \mid x R))\) (when \(x \notin \text{fn}(Q) \land y \notin \text{fn}(P)\))

CONG-ASSOC-R

\((\forall x). (P \mid x (\forall y). (Q \mid y R)) \equiv (\forall y). ((\forall x). (P \mid x Q) \mid y R)\) (when \(x \notin \text{fn}(R) \land y \notin \text{fn}(P)\))

CONGR-SPAWN-SWAP

\(p[\sigma_1].p[\sigma_2].Q \equiv p[\sigma_2].p[\sigma_1].Q\) (when \(\sigma_1 \neq \sigma_2\))

Fig. 2. Structural congruence.

RED-COMM-R

\((\forall x). ((x(y).Q \mid x, \Xi[y].(P_1 \mid P_2)) \rightarrow (\forall x).((\forall y).((P_1 \mid y.Q) \mid x P_2))\)

RED-COMM-L

\((\forall x). ((\forall y).((P_1 \mid x .Q) \mid y .P)) \rightarrow (\forall x).((\forall y).((P_2 \mid x .Q) \mid y .P))\)

RED-CASE

\(\ell \in \{\text{inl, inr}\}\)

\((\forall x). ([x \leftrightarrow P \mid x] \ast \text{case}(Q_{\text{inl}}, Q_{\text{inr}})) \rightarrow (\forall x). (P \mid x P_r)\)

RED-FWD-R

\((\forall x). (P \mid x [y \leftarrow x]) \rightarrow P[y/x]\)

RED-FWD-L

\((\forall y). (P \mid x [x \leftarrow y]) \rightarrow P[y/x]\)

RED-SPAWN

\(\sigma(x) = \{x_1, \ldots, x_n\}\) \(
\sigma' = (x) \cup \{z \mapsto \{z_1, \ldots, z_n\} \mid z \in \text{fn}(P) \setminus \{x\}\}\)

\((\forall x). ((P \mid x P[\sigma].Q) \rightarrow P[\sigma'].(\forall x_1).((P_1 \mid x_1 \ldots (\forall x_n).((P_n \mid x_n \ldots Q)\ldots)\)

RED-SPAWN-R

\((\forall x). (P \mid x P[\sigma].Q) \rightarrow P[\sigma].(\forall x). (P \mid x Q)\)

RED-SPAWN-L

\((\forall x). (P \mid x P[\sigma].Q) \rightarrow P[\sigma].(\forall x). (P \mid x Q)\)

RED-SPAWN-MERGE

\(p[\sigma_1].p[\sigma_2].P \rightarrow p[\sigma_1 \& \sigma_2].P\)

RED-EVAL-CTX

\(K[\cdot] := [\cdot] \mid p[\sigma].K[\cdot] \mid (\forall x).(P \mid K[\cdot]) \mid (\forall x).(K[\cdot] \mid P)\)

\(K[P] \rightarrow K[Q]\)

RED-CONGR

\(P' \equiv P \rightarrow Q \rightarrow Q'\)

Fig. 3. Reduction rules for \(\pi\text{Bl}\).

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 \(\pi\text{Bl}\) is defined in terms of a reduction relation, denoted \(\rightarrow\), which combines the usual reductions of the \(\pi\)-calculus with reductions for spawn prefixes. As usual, we shall write \(\rightarrow^\ast\) to denote the reflexive, transitive closure of \(\rightarrow\), and \(P \not\rightarrow\) when \(P\) cannot reduce.
Figure 3 gives the reduction rules. The first seven rules describe interactions along a channel. Rules \text{red-comm-r} and \text{red-comm-l} describe 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 \( y \). Rules \text{red-unit-r} and \text{red-unit-l} describe the closing of a session at \( x \). Rule \text{red-case} shows how a branch offered on \( x \) can be selected by sending \text{inl} or \text{inr}. Finally, Rules \text{red-fwd-r} and \text{red-fwd-l} explain the elimination of a forwarder connected by restriction in terms of a substitution.

The next four rules of Figure 3 define the semantics of spawn. The crucial rule is Rule \text{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 \( P \triangleq x(x_1).x(x_2).v(x) \) and \( Q \triangleq x_1(x_2).v(x) \). Now consider the following process:

\[
R \triangleq (vx).((z().P | . x) \mapsto x_1, x_2).Q)
\]

In \( R \), the process \( P \) is blocked waiting for the session on a channel \( z \) to close. By Rule \text{red-spawn},

\[
R \rightarrow \rho[z \mapsto z_1, z_2].(vx_1).((z_1().P[x_1/x] | . x_1 (vx_2).((z_2().P[x_2/x] | . x_2).Q))
\]

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 \text{red-spawn} uses the following definition.

Definition 2.3 (Indexed renaming). Given a process \( P \) with \( \text{fn}(P) = \{a, b, \ldots, z\} \), 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 \( a_i, b_i, \ldots, z_i \notin \text{fn}(P) \), \( P(i) \triangleq P[a_i/a, b_i/b, \ldots, z_i/z] \).

Note that Rule \text{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 \( Q' \triangleq \overline{v(x)} \)). Now consider the following process, obtained by replacing the spawn prefix and \( Q \) in \( R \) from Example 2.2:

\[
R' \triangleq (vx).((z().P | . x) \mapsto \emptyset).Q')
\]

By Rule \text{red-spawn}, \( R' \rightarrow \rho[z \mapsto \emptyset].Q' \). 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 \text{red-spawn-r}, \text{red-spawn-l} and \text{red-spawn-merge} show how the spawn prefix interacts with independent process compositions and with other spawn prefixes, respectively. Rules \text{red-spawn-r} and \text{red-spawn-l} are forms 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 \text{red-spawn-merge} describes how two consecutive spawn prefixes can be combined into a single spawn, by merging the spawn bindings, denoted \( \bowtie \), as follows.
Definition 2.5 (Merge). Let $\sigma[X] \triangleq \bigcup \{\sigma(x) \mid x \in X, x \in \text{dom}(\sigma)\}$. The merge of two spawn bindings $\sigma_1, \sigma_2$, written $\sigma_1 \bowtie \sigma_2$, is defined as:

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

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

The idea behind the merge operation $\sigma_1 \bowtie \sigma_2$ 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:

$$ [x \mapsto \emptyset, y \mapsto \{y_1, y_2, y_3\}] \bowtie [y_2 \mapsto \emptyset, y_3 \mapsto \{y_4, y_5\}, z \mapsto z_1] = [x \mapsto \emptyset, y \mapsto \{y_1, y_4, y_5\}, z \mapsto z_1] $$

This merge can be graphically illustrated as follows:

$$ \begin{array}{c}
x \longrightarrow x \longrightarrow y_1 \\
\begin{array}{c} \downarrow \\
y \longrightarrow y_2 \bowtie y_3 \end{array} \end{array} \quad \begin{array}{c}
y_2 \longrightarrow y_4 \\
\begin{array}{c} \downarrow \\
y_3 \longrightarrow z_1 \end{array} \end{array} \quad \begin{array}{c}
y \longrightarrow y_1 \\
\begin{array}{c} \downarrow \\
y \longrightarrow y_5 \\
\begin{array}{c} \downarrow \\
z \longrightarrow z_1 \end{array} \end{array} \end{array} $$

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 ($\emptyset_m$ or $\emptyset_a$) or typing assignments ($x : A$). We write $\text{fn}(\Delta)$ for the set of names occurring in the bunch $\Delta$, and write $x \in \Delta$ to denote $x \in \text{fn}(\Delta)$. As is standard for BI, we consider bunches modulo the least congruence on bunches closed under commutative monoid laws for ’,’ with unit $\emptyset_m$, and for ’;’ with unit $\emptyset_a$, denoted $\equiv$. For example, $(\Delta_1, \emptyset_m) ; \Delta_2 \equiv \Delta_2 ; \Delta_1$.

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 $\Gamma(\cdot | \cdots | \cdot)$ 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.
Fig. 4. Types, typing rules and spawn binding rules for πBI.
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 $A \cdot B$ or with $A \& B$. 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 $A \cdot B$ 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 $A \cdot B$ 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 $A \cdot B$ 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 W&-r and W&-l describe the type $A \& B$. These rules are dual to the rules for $A \cdot B$: providing $A \& B$ requires an input, and using it requires an output.

Rule Emp-r states how to close a session of type $1_m$ 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 $\emptyset_m$, 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 $\text{inl}$/ $\text{inr}$ 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 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 forwarder $[x \leftarrow y]$ 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 channel 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 $\text{fn}(\Delta) = \{a, b, \ldots, z\}$. Assuming $a_i, b_i, \ldots, z_i \notin \text{fn}(\Delta)$, we define $\Delta^{(i)} = \Delta[a_i/a, b_i/b, \ldots, z_i/z]$, 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 $\sigma : \Delta_1 \leadsto \Delta_2$; the bottom of Figure 4 gives their rules.

The idea is to consider a binding $\sigma$ as the merge of a sequence of bindings $\sigma = \sigma_1 \leadsto \ldots \leadsto \sigma_n$, 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:

$$P \triangleq (vx). (z() . Q | x (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R))$$

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

$$\Delta \vdash Q :: x : A \quad \frac{\Psi}{\Delta, \emptyset_m \vdash Q :: x : A} \quad \frac{\Delta, z : 1_m \vdash z().Q :: x : A}{\begin{array}{c} \Gamma(x_1 : A; x_2 : A) \vdash R :: v : B \quad \emptyset_a + \overline{y}() :: y : 1_a \\ \Gamma(x : A; \emptyset_a) \vdash (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R :: v : B \end{array}} \frac{\Gamma(x : A) \vdash (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R :: v : B} {\Gamma(\Delta, z : 1_m) \vdash P :: v : B} \frac{\Gamma(x_1 : A; x_2 : A) \vdash \emptyset_f : 1_f} {\Gamma(x_1 : A; x_2 : A) \vdash (z() . Q | x (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R) :: v : B} \frac{\Delta \vdash (z() . Q | x (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R) :: v : B} {\Delta \vdash P :: v : B} \frac{\Delta \vdash Q :: x : A} {\begin{array}{c} \Gamma(x : A; \emptyset_a) \vdash (vy). (\overline{y}()) | y \rho[x \mapsto x_1, x_2; y \mapsto \emptyset].R \vdash v : B \end{array}}$$

where $\Psi$ is as follows:

$$\begin{align*}
[ x \mapsto x_1, x_2 ] : \Gamma(x : A; y : 1_a) & \leadsto \Gamma(x_1 : A; x_2 : A; y : 1_a) \\
[ y \mapsto \emptyset ] : \Gamma(x_1 : A; x_2 : A; y : 1_a) & \leadsto \Gamma(x_1 : A; x_2 : A) \\
( [ x \mapsto x_1, x_2 ] \leadsto [ y \mapsto \emptyset ] ) : \Gamma(x : A; y : 1_a) & \leadsto \Gamma(x_1 : A; x_2 : A)
\end{align*}$$

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

It is worth noticing that the typing judgment $\sigma : \Delta_1 \leadsto \Delta_2$ is not uniquely determined from $\sigma$ and $\Delta_1$. Hence, there is not always a unique derivation tree for a given judgment. To recover unique typing, it should be sufficient to annotate all bindings with their respective types, including the $\sigma$ in the spawn prefixes.
Empty spawn. We briefly discuss a corner case: according to the typing rules for spawn bindings, we can type the empty spawn \( \rho[\emptyset] \). 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 \( \rho[\emptyset] \rightarrow P \) will cause complications because the empty spawn prefix, though operationally vacuous, can influence the typing. An example is the following application of weakening:

\[
\Gamma(\emptyset) \vdash P : x : A
\]

\[
\Gamma(\emptyset) \vdash \rho[\emptyset].P : x : \overline{A}
\]

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

The \( \pi \)BI calculus 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 \( P \rightarrow^k Q \) to mean that \( P \) reduces to \( Q \) in \( k > 1 \) consecutive steps.

Server and clients. Recall from Example 2.2 the process \( R = (vz).((z()).P | \rho[x \mapsto x_1, x_2].Q) \). 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 \( R’ \triangleq (vz).(!u(x).z’().P | \overline{u}[x_1], \overline{u}[x_2], Q) \). 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 \( \rho[x \mapsto x_1, x_2].Q[x_2/x] \)).

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 \( \pi \)DILL than in \( \pi \)BI. 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 \( z \):

\[
(vz).((z) | R) \rightarrow (vx).((P | \rho[x \mapsto x_1, x_2].Q)).
\]

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 \( P \triangleq x(y).x().z[w].(|w \mapsto y | z().\overline{s}(\)). 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: \( P’ \triangleq x \mapsto case(P, x().\rho[z \mapsto \emptyset], \overline{s}) \).
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:

$$(νz).(z(q).R | (νx).(x < \text{inl}.izzlies[u] \cdot (\bar{u}⟨⟩ | \bar{x}⟨⟩) | P'))$$

$$→^3(νz).(z(q).R | (νu).                                                                               (\bar{u}⟨⟩ | \bar{z}⟨⟩ | (w ← u) | z().\bar{s}⟨⟩)))$$

$$→^3(νu).                                                                               (\bar{u}⟨⟩ | (νz).                                                                               (R[u/q] | z().\bar{s}⟨⟩)))$$

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

$$(νz).(z(q).R | (νx).(x < \text{inr}.izzly⟨⟩ | P')) →^2(νz).                                                                               (z(q).R | p[z ← ϕ].\bar{s}⟨⟩)) → p[ϕ].\bar{s}⟨⟩$$

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 $P \triangleq (νx).                                                                               (\bar{y}⟨⟩ | \bar{z}⟨⟩) | (νz).                                                                               (x(w).x().w().\bar{z}⟨⟩ | p[z ← ϕ].\bar{s}⟨⟩)).$ From $P$, we could either reduce the spawn prefix or synchronize on $x$. If we first reduce the spawn, the spawn propagates to $x$:

$$P → (νx).                                                                               (\bar{y}⟨⟩ | \bar{z}⟨⟩) | p[ϕ].\bar{s}⟨⟩).$$

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

$$P →^2(νy).                                                                               (\bar{y}⟨⟩ | (νz).                                                                               (y().\bar{z}⟨⟩ | p[z ← ϕ].\bar{s}⟨⟩)) → (νy).                                                                               (\bar{y}⟨⟩ | p[ϕ].\bar{s}⟨⟩).$$

**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 $P \triangleq z(a).z(y).p[a ← a_1, a_2].\bar{y}[a'_1].(((a'_1 ← a_1) | \bar{y}[a'_2].((a'_2 ← a_2) | [z ← y]))$ can be typed as providing a session $A → (A → A → B) → B$ on $x$:

| $a_1 : A$ | $a'_1 ← a_1$ | $a'_1 : A$ | $a_2 : A \cdot [a'_2 ← a_2] : a'_2 : A$ | $y : A → B \cdot [z ← y] : z : B$ |
|---------|----------|---------|---------------------------------|--------------------------|
| $a_1 : A$ | $a_2 : A \cdot y : A → A → B \cdot \bar{y}[a'_1].((a'_1 ← a_1) | \bar{y}[a'_2].((a'_2 ← a_2) | [z ← y])) : z : B$ |

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 $A → (A → A → B) → B$ is not derivable: 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 $→$ 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 $\triangleright$. 

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 155. Publication date: October 2022.
On the other hand, there are types provable in DILL that are not provable in BI. A simple example is $A \o B \vdash A \rightarrow B$, converting an implication from linear to non-linear. A "number of uses" interpretation of the conversion makes sense: $A \o B$ promises to use $A$ exactly once to produce $B$; $A \rightarrow B$ declares to produce $B$ using $A$ an unspecified number of times, including exactly once. The corresponding judgment $A \o B \vdash A \rightarrow B$ is not provable in BI (and thus in $\pi$BI). Intuitively, this is because $A \rightarrow B$ allows $A$ to be obtained with resources which share their origin with the resource $A \o B$; however, $A \o B$ can only be applied to resources that do not share its own origin.

**The meaning of multiplicative and additive types.** A natural question arises: if the process interpretation of multiplicatives and additives coincides, what is the difference in the types representing behaviorally? The following example addresses the difference between linear and non-linear connectives; in Section 5 we formally elucidate this difference by giving a denotational semantics which allows tracking the origin of sessions.

**Example 2.9.** Assume an opaque base type $\mathbb{D}$ of data. The type of a stylized database could be $DB \triangleq (\mathbb{D} \rightarrow DB) \land (\mathbb{D} \land DB)$ where the first conjunct can receive some new data to overwrite the contents of the database (the 'put' operation), and the second would provide the current data stored in it (the 'get' operation). This is a recursive type, which is not currently supported by our calculus; for the purposes of this discussion, it is enough to consider some finite unfolding of the type (terminated with $1_a$).

Just by looking at the type $DB$, we can identify possible interactions with the database. A typical usage pattern of a resource $db : DB$ would be to input the 'put' and the 'get' components and weaken the one we are not intending to use in the current step. Imagine we want to put some $d : \mathbb{D}$: then we would weaken the 'get', and send $d$ over $put : (\mathbb{D} \rightarrow DB)$ to obtain a continuation of type $DB$ that represents the updated database.

A second pattern of usage afforded by $\pi$BI is to use contraction to spawn independent snapshots of the database. For example, using contraction we can obtain, from $db : DB$, a copy $db' : DB$. From then on, the two copies can be mutated independently without interference.

Now consider two different $\pi$BI processes, $P_a$ and $P_m$, with judgments:

$\begin{align*}
db_1 : DB & ; db_2 : DB \vdash P_a :: z : C \\
db_1 : DB & , db_2 : DB \vdash P_m :: z : C
\end{align*}$

$P_a$ has access to two databases that are allowed to "overlap" since they are aggregated by a ';' in contrast, $P_m$ has access to two non-overlapping databases. Here "overlapping" has a subtle meaning: it refers to the provenance of the data stored in the two databases, rather than the stored value itself.

To see the difference concretely, imagine we interact, in both cases, with $db_1$ by weakening the 'get', and with $db_2$ by weakening the 'put' (and the continuation of 'get'):

$\begin{align*}
put_1 & : D \rightarrow DB & : d : D \vdash P'_a :: z : C \\
put_1 & : D \rightarrow DB & , d : D \vdash P'_m :: z : C
\end{align*}$

Process $P'_a$ is now allowed to send $d$ on channel $put_1$, updating the database's value to $d$, thus inducing a flow of information from $db_2$ to $db_1$. This flow is however forbidden in the case of $P'_m$: the data sent through $put_1$ needs to be obtained from a resource that is separated with it by ';' as per Rule **IMPL-1**. The fact that $d$ is separated using ';', fundamentally forbids it to flow into $put_1$.

Now suppose $C = D \ast D$ and take $DB$ to be the 1-unfolding of the recursive definition. The typing of $P_m$ ensures that the two data values sent on the channel $z$ would come one from $db_1$ and the other from $db_2$: the combinations where two values taken from the same database are sent on $z$ are disallowed by typing. As we will see in Examples 5.4 and 5.5, the denotational semantics developed in Section 5 formally justifies these claims.
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 [Frumin et al. 2022] 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.** Assume $\Delta \vdash P :: x : C$. If $P \equiv Q$ or $P \rightarrow Q$, then $\Delta \vdash Q :: x : C$.

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 [Frumin et al. 2022] 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 $\text{fn}(\Sigma) = \emptyset$. Equivalently, a bunch is empty if each of its leaves is $\emptyset_m$ or $\emptyset_a$.

**Theorem 3.3 (Deadlock-freedom).** Given an empty bunch $\Sigma$, if $\Sigma \vdash P :: z : A$ with $A \in \{1_m, 1_a\}$, then either (i) $P \equiv \pi()$, or (ii) $P \equiv \rho[\emptyset].\pi()$, or (iii) there exists $S$ such that $P \rightarrow S$.

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 $(\nu x). (\nu y). (y().x()) | x().\bar{y}())$ 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 [Frumin et al. 2022]), 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 calculus is weakly normalizing, that is, for every process $P$ there exists some process $Q$ such that $P \rightarrow^* Q \rightarrow$. This is a result of independent interest, which we will use 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.
Given a process $P$, what kind of reductions can $P$ 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 $fn(R) = \{x, y\}$,

\[(\forall x). (R \mid p[x \mapsto x_1, x_2].Q) \rightarrow p[y \mapsto y_1, y_2].(\forall x_1).(R^{(1)} \mid (\forall x_2).(R^{(2)} \mid Q)). \quad (1)\]

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 $p[x \mapsto x_1, x_2]$ turned into the prefix $p[y \mapsto y_1, y_2]$ with a larger scope. As a result, the communication prefixes in $Q$ went from being guarded directly by $p[x \mapsto x_1, x_2]$, to being guarded by a prefix $p[y \mapsto y_1, y_2]$, 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 $K$, then we can use Rules RED-SPAWN-R and RED-SPAWN-L to actually propagate the spawn prefix to the top-level:

\[\mathcal{K}((\forall x). (R \mid p[x \mapsto x_1, x_2].Q)) \rightarrow \mathcal{K}[p[y \mapsto y_1, y_2].(\forall x_1).(R^{(1)} \mid (\forall x_2).(R^{(2)} \mid Q))]] \quad (2)\]

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

Following this observation, the trick is to stratify the number of prefixes at each $p$-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 $p$-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 $\mu(P) : \mathbb{N} \rightarrow \mathbb{N}$ 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. $\mu(p[\sigma].P) = \mu(P)$ for a top-level $p[\sigma]$. 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 [Frumin et al. 2022] for the full details.

Theorem 3.4. If $\Delta \vdash P :: z : A$ is a typed process, then $P$ is weakly normalizing, i.e., there exists some $Q$ such that $P \rightarrow^* Q \not\rightarrow$.

Theorem 3.4 thus captures the fact that, starting from a process $P$, different reductions may be applicable, or that there might be multiple spawn prefixes that can be brought to the top-most level.

Strictly speaking, we do not require well-typedness assumptions for establishing weak normalization; this property is enforced by the reduction semantics. This is a pleasant consequence of our design for the syntax of processes, which already incorporates some of the structure imposed by typing; this structure is then preserved via the correspondence between commuting conversions and reductions. As such, even the untyped processes are “well-scoped” in the sense that they conform to the tree-like structure typical of session-based interpretations of intuitionistic logics.

The weak normalization theorem is related to cut elimination in BI, but the two theorems are not equivalent. The main discrepancy lies in the fact that not all cut reductions in BI correspond to reductions of $\pi$BI processes; process reductions correspond to reductions of cuts which are not guarded by an input or an output prefix. Consecutively, we cannot directly adopt the usual cut elimination procedure for BI [Arisaka and Qin 2012] for the purposes of showing weak normalization.

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 $A \rightarrow B$, and $\alpha x. M$ with its corresponding function
which use implicit substitutions in $\lambda$-calculus, are translated explicitly using the corresponding left rule in combination with a cut. The weakening and contraction rules, are translated using right rules for the associated connectives. The elimination rules are translated.

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

application $M@N$ for the intuitionistic implication $A \rightarrow B$. Selected typing rules are given in the top of Figure 6; the full type system can be found in Appendix [Frumin et al. 2022].

We write $fv(M)$ 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 $\rightarrow$, 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 $z \notin fv(M)$, we inductively translate the typing derivation of $M$ to a $\pi$Bl typing derivation, denoted $\Gamma \vdash T_\pi(\Gamma \vdash M : A) : z : A$. As customary in translations of $\lambda$ into $\pi$ (cf. [Milner 1992; Sangiorgi and Walker 2003; Wedderburn 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 $\Gamma \vdash T_\pi(\Gamma \vdash M : A) : z : A$. 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 $\text{STRUCT}$ rule.

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

$$
\begin{align*}
a_2 & : A \vdash a_2 : A \\
a_1 & : A \vdash a_1 : A \\
a_2 \ ; a_1 & : A \vdash x \rightarrow A \rightarrow B \ \\
& \vdash y : A \rightarrow A \rightarrow B \ \\
& \vdash y \rightarrow a_2 : A \rightarrow B \\
a_1 & : A \vdash a_2 : A \\
a & : A \vdash y : A \rightarrow A \rightarrow B \ \\
& \vdash (y@a_2)@a_1 : B \\
a & : A \vdash ay. (y@a)@a : A \rightarrow (A \rightarrow A \rightarrow B) \rightarrow B
\end{align*}
$$

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 155. Publication date: October 2022.
The translation of $M$ into $\pi$BI is

$$
\mathcal{T}_\pi(M) = z(a).z(y). p[a \mapsto a_1, a_2]. (\forall x). ((\forall w). (\{ w \leftarrow y \} | \mathcal{T}[a'_2]. ([a'_2 \leftarrow a_2] | [x \leftarrow w]))
\mathcal{T}[a'_1]. ([a'_1 \leftarrow a_1] | [z \leftarrow x])).
$$

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 $\mathcal{T}_\pi(-)$ 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 [Frumin et al. 2022] 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 $(ax. M)$ $N$ 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 $(\forall x). (\mathcal{T}_\pi(N) | \mathcal{T}_\pi(M))$, in which one copy of $\mathcal{T}_\pi(N)$ gets connected with the body $\mathcal{T}_\pi(M)$ through the endpoint $x$. The contraction of the multiple occurrences of $x$ in $M$ is handled with a spawn prefix in $\mathcal{T}_\pi(M)$. 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 $\Delta \vdash P \rightarrow M :: z : A$, or $P \rightarrow M$ for short, if:

1. $P \equiv p[\sigma_1], \ldots, p[\sigma_s].(\forall x_a). (\mathcal{T}_\pi(N_a) | \ldots | (\forall x_1). (\mathcal{T}_\pi(N_1) | \mathcal{T}_\pi(M') \ldots))$ where for each $i \in [1, s]$, $\sigma_i = [y_1 \mapsto \emptyset; \ldots; y_m \mapsto \emptyset]$ (only weakening) or $\sigma_i = [y_1 \mapsto y'_1; \ldots; y_m \mapsto y'_m]$ (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 $\mathcal{T}_\pi(N) \rightarrow N$.

We then show the completeness result.

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

#### 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 $P \triangleq (\text{vx}).(Q | \mathcal{T}_z(M))$, suppose that the subterm $N$ has a reduction $N \mapsto N'$. The process $P$ can mimic this reduction:

$$P \rightarrow (\text{vx}).(Q | \mathcal{T}_z(M)),$$

for some $Q$. However, we do not necessarily have a corresponding reduction $M[N/x] \mapsto M[N'/x]$, 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 $\rightarrow^*$ (with reflexive, transitive closure denoted $\mapsto^*$). To be precise, let $\mathcal{C}$ be an arbitrary $\alpha\lambda$-calculus context. In addition to the reductions in Figure 6, we consider reductions under arbitrary contexts:

$$\frac{M \rightarrow M'}{\mathcal{C}[M] \rightarrow^* \mathcal{C}[M']}$$

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

Note that the premise in the theorem above permits arbitrarily many reduction steps from $P$ to $Q$ (i.e., $P \rightarrow^* Q$), ensuring 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., $P \rightarrow Q$) 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:

$$\alpha ::= x \in \text{in} \mid x \in \text{inr} \mid x \rightarrow \text{in} \mid x \rightarrow \text{inr} \mid x \mid \overline{x} \mid x() \mid \overline{x}()$$

By $\text{chan}(\alpha)$ 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 $Q \rightarrow^* Q' \downarrow_\alpha$.

![Barbs for $\pi$BI processes](image-url)
We will be mainly concerned with barbed equivalence of closed processes. A process \( P \) is closed if it is typeable as \( \Sigma \vdash P : y : B \), where \( \text{fn}(\Sigma) = \emptyset \), i.e. \( \Sigma \) is an empty bunch (cf. 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 πBI 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 \( \Delta \vdash P \approx_\gamma Q :: T \), 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 \( C \). Next, we describe a sound, more compositional approach to proving equivalence, based on a denotational semantics for πBI.

### 5.1 Denotational Semantics

Our motivation for developing a denotational semantics for πBI 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 πBI.

Intuitively, if we have a closed process of the type \( \Sigma \vdash P : x : A \ast B \), 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 \( \rho[x \mapsto x_1, x_2] \) 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 \( a_1, a_2, \ldots \), 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 (cf. 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 \( \Sigma = (x : a_1 ; y : a_2) , z : a_3 \) an atomic bunch. Since we do not add any typing rules for atomic processes, well-typed processes cannot "break down" sessions \( a_1 \) 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 \( \varphi(\text{Tag}) \)-valued set: \( \llbracket A \rrbracket : \varphi(\text{Tag}) \rightarrow \text{Set} \). We have:

\[
\begin{align*}
1_m(D) &= \{\bullet\} \\
1_a(D) &= \{\bullet\} \\
[A \lor B](D) &= [A](D) + [B](D) \\
[A \land B](D) &= [A](D) \times [B](D) \\
[A \rightarrow B](D) &= [A](D) \rightarrow [B](D) \\
[A \\ B](D) &= \Pi_{D' \land D = \emptyset} \llbracket A \rrbracket(D') \rightarrow [B](D \cup D')
\end{align*}
\]

where \( \{\bullet\} \) is the (terminal) set with one element, and \( D = D_1 \uplus D_2 \) holds if \( D_1 \cap D_2 = \emptyset \) and \( D = D_1 \cup D_2 \). The interpretation of bunches \( \llbracket A \rrbracket \) is defined analogously by treating ‘;’ and ‘,’ as ‘\( \land \)’ and ‘\( \lor \)’.
and ‘∗’, respectively. A process is interpreted as a function polymorphic in a set of tags \( D \):
\[
[\Delta \vdash P :: x : A]_D : [\Delta](D) \to [A](D).
\]
The interpretation follows the standard interpretation of BI in doubly closed categories [Pym 2002, Chapter 3.3]. Specifically, we interpret types as presheaves \( \mathbf{Set}^{\mathbf{Tag}} \), where \( \mathbf{Tag} \) is interpreted as a discrete category. The interpretation of type formers corresponds to the Cartesian closed structure and a closed monoidal structure on \( \mathbf{Set}^{\mathbf{Tag}} \). As the construction is standard, we omit the details in the interest of space.

The meaning of multiplicative and additive types, formally. As we claimed in Example 2.9, the difference between the multiplicative and additive types can be explained in terms of data flow. Equipped with the denotational semantics, we can now make this intuition formal. The idea is that applying contraction to some resource \( A \), leaves a trace in the form of the ‘∗’ in the resulting resource \( A \cdot A \). This records the fact that, although we can interact with each copy of \( A \) independently, the duplicated resources share a common provenance.

Atomic types represent the base types, whose provenance we want to track. Assigning the same tag to two atomic types indicates that they have a common provenance. For example, to inhabit \( [a_1 \cdot a_2]([t_1, t_2]) \) a term needs to decide how to split \( \{t_1, t_2\} \) into two disjoint sets, therefore assigning \( t_1 \) to \( a_1 \) and \( t_2 \) to \( a_2 \) (or vice versa), forcing the two atomic types to have different provenance (as expected for separation). The following examples show what this means for \( \pi \mathbf{BI} \) processes.

Example 5.4. Let \( \Delta \vdash P :: x : a_1 \cdot a_2 \). Then for any \( D \in \mathbf{Tag} \), and \( v \in [\Delta](D) \), \( [P]_D(v) = (t_1, t_2) \) with \( t_1 \neq t_2 \). In other words, the two sessions \( a_1 \) and \( a_2 \) that are sent over the channel \( x \) by \( P \) have a disjoint provenance. Note that this would not hold for a process typed \( \Delta \vdash Q :: x : a_1 \land a_2 \). For example, if \( Q \) is \( \rho[y \mapsto y_1, y_2]\mathbf{x}(y_1) \cdot [x \mapsto y_2] \), we have \( [y] :: \Delta \vdash Q :: x : a \land a \) \( D(t) = (t, t) \), for all \( t \in D \), i.e. the sessions that are sent over the channel \( x \) by \( Q \) do share their provenance.

Furthermore, since the denotational semantics is compositional, we can generalize the argument above. Suppose we place \( P \) in some enclosing context \( C \) such that \( y : a_1' \cdot z : a_2' \vdash C[P] :: x : a_1 \cdot a_2 \). Then \( [y] : a_1' \cdot z : a_2' \vdash C[P] :: x : a_1 \cdot a_2 \) \( [t_1, t_2] = (t_1', t_2') \) and either \( t_1 = t_1' \) and \( t_2 = t_2' \), 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 provenance as \( y \) and \( z \), or with swapped provenance. If we instead consider a program typed with additive conjunction \( y : a_1' \cdot z : a_2' \vdash C[Q] :: x : a_1 \cdot a_2 \), then it may further send over \( x \) two sessions both with the same provenance (either the one of \( y \) or of \( z \)).

Example 5.5. Recall the database type from Example 2.9. Here we let the data stored in the database be of atomic type \( a \), and we consider the 1-unfolding of the original type: \( DB_a \triangleq (a \to 1_a) \land (a \land 1_a) \). In Example 2.9 we claimed that a process with access to two databases can generate a flow of data from one to the other only if they are additively composed. Let us see how the denotational semantics makes this evident at the type level. Consider the additive case, symbolized by process \( P'_a \).
For any set of tags \( D \), we have:
\[
[\text{put}_1 : a \to 1_a; d : a \vdash P'_a :: z : C]_D : ((D \to \{\bullet\}) \times D) \to [C]_D.
\]
Therefore, the denotation of \( \text{put}_1 \), with domain \( D \), can be applied to the tag of \( d \), which is a member of \( D \). Data flow from one database to the other is allowed since the databases are already declared to have shared provenance. In contrast, for the multiplicative case represented by \( P'_m \), we have:
\[
[\text{put}_1 : a \to 1_a, d : a \vdash P'_m :: z : C]_D : ((D_1 \to \{\bullet\}) \times D_2) \to [C]_{D_1}.
\]
for sets of tags \( D_1 \) and \( D_2 \) such that \( D_1 \uplus D_2 = D \). In this case, because \( \text{put}_1 \) and \( d \) are separated by ‘∗’, the tags associated with \( a \) in these types must be disjoint. Therefore, in the case of \( P'_m \), it
is impossible to apply put (the denotation of which has domain $D_1$) to the data from the second database (which comes from $D_2$).

### 5.2 Properties

We 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 5.6.** Let $\Delta \vdash P :: z : A$ and $\Delta \vdash Q :: z : A$ be processes such that $P \equiv Q$. Then $[P] = [Q]$.

**Lemma 5.7.** Let $\Delta \vdash P :: z : A$ and $\Delta \vdash Q :: z : A$ be processes such that $P \rightarrow Q$. Then $[P] = [Q]$.

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

**Theorem 5.8.** Given two processes $\Delta \vdash P :: z : C$ and $\Delta \vdash Q :: z : C$, if $[P] = [Q]$, then $\Delta \vdash P \equiv_o Q :: z : C$.

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

**Lemma 5.9.** Suppose given a process $P$ such that $\Gamma \vdash P :: z : C$, where $P \rightarrow$ and $P$ does not have any barbs on channels from $\Gamma$. Then $P$ has a barb on the channel $z$.

**Lemma 5.10 (Observability).** Suppose $\Sigma \vdash P :: z : C$ such that $\Sigma$ is an atomic bunch. Then there exists a process $P$ such that $P \rightarrow^* Q$ where $Q \downarrow_{\alpha(z)}$.

**Proof.** By weak normalization and subject reduction (Theorems 3.1 and 3.4), and Lemma 5.9. □

We can use the observability lemma to show Theorem 5.8:

**Proof (of Theorem 5.8).** Let $C$ be a closing program context. We are to show $\Sigma \vdash C[P] \equiv_o C[Q] :: z : D$ for some context $C$.

Since the denotational semantics is compositional, we have $[C[P]] = [C[Q]]$. Furthermore, the relation $P, Q \rightarrow [C[P]] = [C[Q]]$ is an equivalence relation and is closed under reductions (Lemma 5.7). 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 $Q \rightarrow^* Q' \downarrow_{\alpha(z)}$.

For simplicity, let us consider a case where the type $D$, on which we are making observations, is of the form $A \lor B$. Then the only possible observations for $P$ and for $Q$ are $z \inl$ and $z \inr$. Suppose, without loss of generality, that $P \downarrow_{z \inl}$. Then, by Lemma 5.10, we have $Q \rightarrow^* Q' \downarrow_{\alpha(z)}$.

By Lemma 5.7, we have $[P] = [Q] = [Q']$. Because $P$ has a barb $z \inl$, the interpretation $[P]$ must be of the form $\text{inl} \circ (\ldots)$, where $\text{inl}$ is the embedding $[A] \rightarrow [A] + [B]$. It the must be the case that $[Q']$ is also of the same shape, and, hence, $Q' \downarrow_{z \inl}$.

As we have seen, the proof of Theorem 5.8 relies on Lemma 5.10, which in turn relies on weak normalization (Theorem 3.4). In extensions of $\pi$BI for which weak normalization does not hold, the proof strategy would need to appeal to observability based on other techniques, such as logical relations. We elaborate on this in Section 6.

*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 $\equiv_o$ from Section 4 decomposes as a translation $\mathcal{T}_\omega(\cdot)$ and an observational equivalence $\approx_o$.

**Theorem 5.11.** If $\Delta \vdash P = M :: z : A$, then $[P] = [\mathcal{T}_\omega(M)]$, and, consequently, $P \approx_o \mathcal{T}_\omega(M)$.
Proof. Essentially, we need to show that for any $\Gamma(x : A) \vdash M : B$ and $\Delta \vdash N : A$, we have

$$[T_x(M[N/x])] = [(\forall x). (T_x(N) \mid T_x(M))].$$

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 $T_x(\_)$ itself to establish a simulation; instead we had to take a coarser relation $\triangleright$. Theorem 5.11 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 $\lambda_{sep}$ 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.

Observational equivalence. Observational equivalence compares the behaviour of two processes in every (well-typed) program context. This universal quantification makes direct proofs of equivalence very hard and non-compositional. Observational equivalence is therefore usually established using more compositional methods that do not involve reasoning about a program in a context. Examples of these methods are bisimulations and logical relations; in the session-typed setting, such methods have been addressed in [Kouzapas et al. 2011] and [Atkey 2017; Caires et al. 2013; Derakhshan et al. 2021; Pérez et al. 2014], respectively. In Section 5 we followed an approach based on denotational semantics, exploiting a canonical construction. Our denotational semantics also serves to elucidate the difference between the additive and multiplicative types. Our proof of adequacy of the denotational semantics for proving observational equivalence (Theorem 5.8) relies on weak normalization (Theorem 3.4). In extensions of \( \pi BI \) for which weak normalization does not hold, our proof strategy would need to be revised. Recent work on denotational semantics and logical relations for session-typed languages [Derakhshan et al. 2021; Kavanagh 2022] could provide the basis for handling such extensions.

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 \( !A \cong A * \cdots * A \). 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.

ACKNOWLEDGMENTS

The authors would like to thank Revantha Ramanayake for providing comments on an early version of the paper and the anonymous referees for their valuable comments and helpful suggestions. The authors also would like to thank Vasilios Andrikopoulos for helpful discussions.

This work was supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software); and by a European Research Council (ERC) Consolidator Grant for the project “PERSIST” under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).
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