On Polymorphic Sessions and Functions

A Tale of Two (Fully Abstract) Encodings

BERNARDO TONINHO, NOVA-LINCS and NOVA School of Science and Technology, Portugal
NOBUKO YOSHIDA, Imperial College London, United Kingdom

This work exploits the logical foundation of session types to determine what kind of type discipline for the π-calculus can exactly capture, and is captured by, λ-calculus behaviours. Leveraging the proof theoretic content of the soundness and completeness of sequent calculus and natural deduction presentations of linear logic, we develop the first *mutually inverse* and *fully abstract* processes-as-functions and functions-as-processes encodings between a polymorphic session π-calculus and a linear formulation of System F. We are then able to derive results of the session calculus from the theory of the λ-calculus: (1) we obtain a characterisation of inductive and coinductive session types via their algebraic representations in System F; and (2) we extend our results to account for *value* and *process* passing, entailing strong normalisation.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; **Process calculi**; *Linear logic*; • **Software and its engineering** → *Message passing*; *Concurrent programming languages*; *Concurrent programming structures*.

Additional Key Words and Phrases: Session Types, π-calculus, System F, Linear Logic, Full Abstraction

ACM Reference Format:
Bernardo Toninho and Nobuko Yoshida. 2021. On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings. *ACM Trans. Program. Lang. Syst.* 1, 1 (March 2021), 55 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The π-calculus is an analytical tool for understanding [interactive] systems – Robin Milner [41]

Encodability is the main traditional method to compare and examine process calculi and their operators with respect to their expressive power. There are in fact an enormous number of process calculi for expressing non-determinism, parallelism, distribution, locality, real-time, stochastic phenomena, etc, and each of these aspects can be described in different ways. Encodings not only allow a comparison of the expressive power of languages but also formalise similarities and differences between the considered calculi. Thus, they provide a basis for design and implementations of concurrent language primitives and operators into real systems and programming languages [49, 52]. One of the first examples of this is an input-guarded choice encoding in the π-calculus [44], which provided a library in the Pict Programming Language [57].

Dating back to Milner's seminal work [42], encodings of λ-calculus into π-calculus are, in particular, seen as essential benchmarks to examine expressiveness of various extensions of the π-calculus.

Authors' addresses: Bernardo Toninho, NOVA-LINCS and NOVA School of Science and Technology, Department of Informatics, Portugal, btoninho@fct.unl.pt; Nobuko Yoshida, Imperial College London, Department of Computing, United Kingdom, nobuko.yoshida@imperial.ac.uk.

0164-0925/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Milner's original motivation was to demonstrate the power of link mobility by decomposing higher-order computations into pure name passing. Another goal was to analyse functional behaviours in a broad computational universe of concurrency and non-determinism. While *operationally* correct encodings of many higher-order constructs exist, it is challenging to obtain encodings that are precise with respect to behavioural equivalence: the semantic distance between the λ-calculus and the π-calculus typically requires either restricting process behaviours [64] (e.g. via typed equivalences [8]) or enriching the λ-calculus with constants that allow for a suitable characterisation of the term equivalence induced by the behavioural equivalence on processes [62].

Pierce and Sangiorgi [56], exploring the fact that types for π-calculi limit the valid contexts in which processes may interact, observed the semantic consequences of typed equivalences by showing that the observational congruence induced by IO-subtyping can prove the *semantic* correctness of Milner's encoding [55], which was impossible in the untyped setting. Following these developments, many works on typed π-calculi have investigated the correctness of Milner's encodings in order to examine the power of proposed typing systems.

Encodings in π-calculi also gave rise to new typing disciplines: *Session types* [28, 30], a typing system that is able to ensure deadlock-freedom for communication protocols between two or more parties [31], were originally motivated "from process encodings of various data structures in an asynchronous version of the π-calculus" [29]. Following this original motivation, session types have been integrated into mainstream programming languages [1, 21]. A popular technique is to use "encodings" of session types into linear or functional types to correctly implement *structured communications* in programming languages such as Haskell [46], OCaml [32, 34, 48] and Scala [67, 68] (see Section 6).

Recently, a propositions-as-types correspondence between linear logic and session types [12, 13, 76] has produced several new developments and logically-motivated techniques [11, 37, 70, 76] to augment both the theory and practice of session-based message-passing concurrency. Notably, parametric session polymorphism [11] (in the sense of Reynolds [59]) has been proposed and a corresponding abstraction theorem has been shown.

Our work expands upon the proof theoretic consequences of this propositions-as-types correspondence to address the problem of how to *exactly* match the behaviours induced by session π-calculus encodings of the λ-calculus with those of the λ-calculus. We develop *mutually inverse* and *fully abstract* encodings (up to typed observational congruences) between a polymorphic session-typed π-calculus and the polymorphic λ-calculus. The encodings arise from the proof theoretic content of the equivalence between sequent calculus (i.e. the session calculus) and natural deduction (i.e. the λ-calculus) for *second-order* intuitionistic linear logic, greatly generalising those for the propositional setting [70]. While fully abstract encodings between λ-calculi and π-calculi have been proposed (e.g. [8, 62]), our work is the first to consider a two-way, *both* mutually inverse *and* fully abstract embedding between the two calculi by crucially exploiting the linear logic-based session discipline. This also sheds some definitive light on the nature of concurrency in the (logical) session calculi, which exhibit "don't care" forms of non-determinism (e.g. processes may race on stateless replicated servers) rather than "don't know" non-determinism (which requires less harmonious logical features [3]).

In the spirit of Gentzen [22], who established soundness and completeness of his sequent calculus and natural deduction in order to use the former as a way to study the latter (i.e., to show consistency and normalisation of natural deduction through cut elimination in the sequent calculus), we use our encodings as a tool to study non-trivial properties of the session calculus, deriving them from results in the λ-calculus: We show the existence of inductive and coinductive sessions in the polymorphic session calculus by considering the representation of initial F-algebras and final F-coalgebras [40] in the polymorphic λ-calculus [2, 27] (in a linear setting [10]). By appealing to full abstraction, we are able to derive processes that satisfy the necessary algebraic properties and thus form adequate *uniform* representations of inductive and coinductive session types. The derived algebraic properties enable us to reason about standard data structure examples, providing a logical justification to typed variations of the representations in [43].

We systematically extend our results to a session calculus with λ-term and process passing [71], inspired by Benton's LNL [6]. By showing that our encodings naturally adapt to this setting, we prove that it is possible to encode higher-order process passing in the first-order session calculus fully abstractly, providing a typed and proof-theoretically justified re-envisioning of Sangiorgi's encodings of higher-order π-calculus [65]. In addition, the encoding instantly provides a strong normalisation property of the higher-order session calculus.

Contributions and Outline. Contributions of our article are as follows:

- **Section 3.1** develops a functions-as-processes encoding of a linear formulation of System F, Linear-F, using a logically motivated polymorphic session π-calculus, Polyπ, and shows that the encoding is operationally sound and complete.
- **Section 3.2** develops a processes-as-functions encoding of Polyπ into Linear-F, arising from the completeness of the sequent calculus wrt natural deduction, also operationally sound and complete.
- **Section 3.3** studies the relationship between the two encodings, establishing they are *mutually inverse* and *fully abstract* wrt typed congruence, the first two-way embedding satisfying *both* properties.
- **Section 4** develops a *faithful* representation of inductive and coinductive session types in Polyπ via the encoding of initial and final (co)algebras in the polymorphic λ-calculus, which is driven through our encodings to produce processes satisfying the necessary algebraic properties. We demonstrate a use of these algebraic properties via examples.
- **Sections 5 and 5.2** study term-passing and process-passing session calculi, extending our encodings to provide embeddings into the first-order session calculus. As a consequence, we obtain a proof-theoretically, type-driven reinvisioning of Sangiorgi's encodings of higher-order processes into first-order processes. We show that the full abstraction and mutual inversion results are smoothly extended to these calculi and derive strong normalisation of the higher-order session calculus from the encoding.

In order to introduce our encodings, we first overview the logically motivated polymorphic session calculus Polyπ, its typing system and behavioural equivalence (Section 2). We discuss related work in Section 6 and conclude with future work in Section 7. The appendix includes detailed proofs and additional lemmas.

Outline. This article revises and extends an earlier version of this work [73] with additional materials and full proofs. § 2 was extended to include all the necessary formal definitions for the development of the coming sections, namely the definitions of structural and extended structural congruence, typed barbed congruence and logical equivalence. We further include the complete set of typing rules of the system and extended discussion on their relationship with the literature on linear logic. We further include a more detailed analysis of logical equivalence. Section 3 now details the operational semantics of Linear-F. Section 3.2 includes the encoding from session π-calculus typing derivations to Linear-F typing derivations explicitly. We have also included additional discussion throughout the section on the relationship with various proof theoretic considerations and extended the examples, as well as additional discussion on the nature of the encodings with respect to the operational semantics of Linear-F and potential extensions to effects and non-divergence. The proofs of the main results of the section, namely of full abstraction (Theorems 3.15

and 3.16) are included in the main article. Proofs of the results in the remainder of the section can be found in detail in the appendix. Section 4 has been extended with additional discussion, explanations and proofs. Section 5 has generally been extended with additional results and proofs. Section 5.2 now includes the development of the strong normalisation result (Theorem 5.24) for the higher-order process passing calculus via a modification of the encoding presented previously in the section, which also includes the reestablishment of the properties of operational correspondence, and the inverse theorem for the reformulated encoding. Finally, Section 6 has been enhanced with additional discussion of related work, including works that were published after the conference version of this work [73].

2 POLYMORPHIC SESSION π-CALCULUS

This section summarises the polymorphic session π-calculus [11], dubbed Polyπ, arising as a process assignment to second-order linear logic [23], its typing system and behavioural equivalences.

2.1 Processes and Typing

Syntax. Given an infinite set of names x, y, z, u, v, w, the grammar of processes P, Q, R and session types A, B, C is defined by:

$$\begin{array}{lcllllll} P, Q, R & ::= & x\langle y\rangle.P & \mid\ x(y).P & \mid\ P \mid Q & \mid\ (\nu y)P & \mid\ [x \leftrightarrow y] & \mid\ \mathbf{0} \\ & \mid & x\langle A\rangle.P & \mid\ x(Y).P & \mid\ x.\mathsf{inl}; P & \mid\ x.\mathsf{inr}; P & \mid\ x.\mathsf{case}(P, Q) & \mid\ !x(y).P \\ A, B & ::= & \multicolumn{6}{l}{\mathbf{1} \mid A \multimap B \mid A \otimes B \mid A \,\&\, B \mid A \oplus B \mid\ !A \mid \forall X.A \mid \exists X.A \mid X} \end{array}$$

$x\langle y\rangle.P$ denotes the output of channel y on x with continuation process P; $x(y).P$ denotes an input along x, bound to y in P; $P \mid Q$ denotes parallel composition; $(\nu y)P$ denotes the restriction of name y to the scope of P; $\mathbf{0}$ denotes the inactive process; $[x \leftrightarrow y]$ denotes the linking of the two channels x and y (implemented as renaming); $x\langle A\rangle.P$ and $x(Y).P$ denote the sending and receiving of a *type* A along x bound to Y in P of the receiver process; $x.\mathsf{inl}; P$ and $x.\mathsf{inr}; P$ denote the emission of a selection between the left or right branch of a receiver $x.\mathsf{case}(P, Q)$ process; $!x(y).P$ denotes an input-guarded replication that spawns replicas upon receiving an input along x. We often abbreviate $(\nu y)x\langle y\rangle.P$ to $\overline{x}\langle y\rangle.P$ and omit trailing $\mathbf{0}$ processes. By convention, we range over linear channels with x, y, z and shared channels with u, v, w.

The syntax of session types is that of (intuitionistic) linear logic propositions which are assigned to channels according to their usages in processes: $\mathbf{1}$ denotes the type of a channel along which no further behaviour occurs; $A \multimap B$ denotes a session that waits to receive a channel of type A and will then proceed as a session of type B; dually, $A \otimes B$ denotes a session that sends a channel of type A and continues as B; $A \,\&\, B$ denotes a session that offers a choice between proceeding as behaviours A or B; $A \oplus B$ denotes a session that internally chooses to continue as either A or B, signalling appropriately to the communicating partner; $!A$ denotes a session offering an unbounded (but finite) number of behaviours of type A; $\forall X.A$ denotes a polymorphic session that receives a type B and behaves uniformly as $A\{B/X\}$; dually, $\exists X.A$ denotes an existentially typed session, which emits a type B and behaves as $A\{B/X\}$.

Operational Semantics. The operational semantics of our calculus is presented as a standard labelled transition system (Fig. 1) in the style of the *early* system for the π-calculus [65].

In the remainder of this work we write $\equiv$ for a standard π-calculus structural congruence (Def. 2.1) extended with the clause $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$. In order to streamline the presentation of observational equivalence [11, 50], we write $\equiv_!$ (Def. 2.2) for structural congruence extended with the so-called sharpened replication axioms [65], which capture basic equivalences of replicated processes (and are present in the proof dynamics of the exponential of linear logic).

$$\text{(out)}\; x\langle y\rangle.P \xrightarrow{\overline{x\langle y\rangle}} P \qquad \text{(in)}\; x(y).P \xrightarrow{x(z)} P\{^{z}/_{y}\} \qquad \text{(outT)}\; x\langle A\rangle.P \xrightarrow{\overline{x\langle A\rangle}} P \qquad \text{(inT)}\; x(Y).P \xrightarrow{x(B)} P\{^{B}/_{Y}\}$$

$$\text{(rout)}\; x.\mathsf{inr};P \xrightarrow{\overline{x.\mathsf{inr}}} P \qquad \text{(lout)}\; x.\mathsf{inl};P \xrightarrow{\overline{x.\mathsf{inl}}} P \qquad \text{(id)}\; (\nu x)([x \leftrightarrow y] \mid P) \xrightarrow{\tau} P\{y/x\} \qquad \text{(open)}\; \frac{P \xrightarrow{\overline{x\langle y\rangle}} Q}{(\nu y)P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} Q}$$

$$\text{(rin)}\; x.\mathsf{case}(P,Q) \xrightarrow{x.\mathsf{inr}} Q \qquad \text{(lin)}\; x.\mathsf{case}(P,Q) \xrightarrow{x.\mathsf{inl}} P \qquad \text{(rep)}\; !x(y).P \xrightarrow{x(z)} P\{^{z}/_{y}\} \mid !x(y).P$$

$$\text{(close)}\; \frac{P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')} \qquad \text{(par)}\; \frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \qquad \text{(com)}\; \frac{P \xrightarrow{\overline{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{(res)}\; \frac{P \xrightarrow{\alpha} Q}{(\nu y)P \xrightarrow{\alpha} (\nu y)Q}$$

Fig. 1. Labelled Transition System.

Definition 2.1 (Structural congruence). ($P \equiv Q$), is the least congruence relation generated by the following laws:

$$P \mid \mathbf{0} \equiv P \quad P \equiv_\alpha Q \Rightarrow P \equiv Q \quad P \mid Q \equiv Q \mid P \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \quad x \notin fn(P) \Rightarrow P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) \quad (\nu x)\mathbf{0} \equiv \mathbf{0} \quad [x \leftrightarrow y] \equiv [y \leftrightarrow x]$$

Definition 2.2 (Extended Structural Congruence). We write $\equiv_!$ for the least congruence relation on processes which results from extending structural congruence $\equiv$ with the following axioms:

(1) $(\nu u)(!u(z).P \mid (\nu y)(Q \mid R)) \equiv_! (\nu y)((\nu u)(!u(z).P \mid Q) \mid (\nu u)(!u(z).P \mid R))$
(2) $(\nu u)(!u(y).P \mid (\nu v)(!v(z).Q \mid R)) \equiv_! (\nu v)((!v(z).(\nu u)(!u(y).P \mid Q)) \mid (\nu u)(!u(y).P \mid R))$
(3) $(\nu u)(!u(y).Q \mid P) \equiv_! P$ if $u \notin fn(P)$

Axioms (1) and (2) above represent principles for the distribution of shared servers among processes, while (3) formalises the garbage collection of shared servers which cannot be invoked by any process. The axioms embody distributivity, contraction and weakening of shared resources and are sound wrt (typed) observational equivalence [50].

A transition $P \xrightarrow{\alpha} Q$ denotes that P may evolve to Q by performing the action represented by label α. An action α ($\overline{\alpha}$) requires a matching $\overline{\alpha}$ (α) in the environment to enable progress. Labels of our transition semantics include the silent internal action τ, output and bound output actions ($\overline{x\langle y\rangle}$ and $\overline{(\nu z)x\langle z\rangle}$); input action $x(y)$; labels pertaining to the binary choice actions ($x.\mathsf{inl}$, $\overline{x.\mathsf{inl}}$, $x.\mathsf{inr}$, and $\overline{x.\mathsf{inr}}$); and labels describing output and input actions of types ($\overline{x\langle A\rangle}$ and $x(A)$).

Definition 2.3 (Labelled Transition System). The labelled transition relation is defined by the rules in Fig. 1, subject to the side conditions: in rule (res), we require $y \notin fn(\alpha)$; in rule (par), we require $bn(\alpha) \cap fn(R) = \emptyset$; in rule (close), we require $y \notin fn(Q)$. We omit the symmetric versions of (par), (com), (id), (close) and closure under α-conversion.

We write $\rho_1\rho_2$ for the composition of relations ρ_1, ρ_2. We write $\rightarrow$ to stand for $\xrightarrow{\tau}\equiv$. Weak transitions are defined as usual: we write $\Longrightarrow$ for the reflexive, transitive closure of $\rightarrow$ and $\rightarrow^+$ for the transitive closure of $\rightarrow$. Given $\alpha \neq \tau$, notation $\stackrel{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ and $\stackrel{\tau}{\Longrightarrow}$ stands for $\Longrightarrow$.

Typing System. The typing rules of Polyπ are given in Fig. 2, following [11]. The rules define the judgment $\Omega; \Gamma; \Delta \vdash P :: z{:}A$, denoting that process P offers a session of type A along channel z, using the *linear* sessions in Δ, (potentially) using the unrestricted or *shared* sessions in Γ, with

$$(1\mathsf{R})\ \frac{}{\Omega;\Gamma;\cdot\vdash \mathbf{0} :: z{:}\mathbf{1}} \qquad (1\mathsf{L})\ \frac{\Omega;\Gamma;\Delta\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}\mathbf{1}\vdash P :: z{:}C}$$

$$(\multimap\mathsf{R})\ \frac{\Omega;\Gamma;\Delta, x{:}A\vdash P :: z{:}B}{\Omega;\Gamma;\Delta\vdash z(x).P :: z{:}A\multimap B} \qquad (\otimes\mathsf{R})\ \frac{\Omega;\Gamma;\Delta_1\vdash P :: y{:}A \quad \Omega;\Gamma;\Delta_2\vdash Q :: z{:}B}{\Omega;\Gamma;\Delta_1,\Delta_2\vdash (\nu y)z\langle y\rangle.(P\mid Q) :: z{:}A\otimes B}$$

$$(\multimap\mathsf{L})\ \frac{\Omega;\Gamma;\Delta_1\vdash P :: y{:}A \quad \Omega;\Gamma;\Delta_2, x{:}B\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2, x{:}A\multimap B\vdash (\nu y)x\langle y\rangle.(P\mid Q) :: z{:}C} \qquad (\otimes\mathsf{L})\ \frac{\Omega;\Gamma;\Delta, y{:}A, x{:}B\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}A\otimes B\vdash x(y).P :: z{:}C}$$

$$(\&\mathsf{R})\ \frac{\Omega;\Gamma;\Delta\vdash P :: z{:}A \quad \Omega;\Gamma;\Delta\vdash Q :: z{:}B}{\Omega;\Gamma;\Delta\vdash z.\mathsf{case}(P,Q) :: z{:}A\,\&\,B} \qquad (\&\mathsf{L}_1)\ \frac{\Omega;\Gamma;\Delta, x{:}A\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}A\,\&\,B\vdash x.\mathsf{inl};P :: z{:}C}$$

$$(\&\mathsf{L}_2)\ \frac{\Omega;\Gamma;\Delta, x{:}B\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}A\,\&\,B\vdash x.\mathsf{inr};P :: z{:}C} \qquad (\oplus\mathsf{R}_1)\ \frac{\Omega;\Gamma;\Delta\vdash P :: z{:}A}{\Omega;\Gamma;\Delta\vdash z.\mathsf{inl};P :: z{:}A\oplus B}$$

$$(\oplus\mathsf{R}_2)\ \frac{\Omega;\Gamma;\Delta\vdash P :: z{:}B}{\Omega;\Gamma;\Delta\vdash z.\mathsf{inr};P :: z{:}A\oplus B} \qquad (\oplus\mathsf{L})\ \frac{\Omega;\Gamma;\Delta, x{:}A\vdash P :: z{:}C \quad \Omega;\Gamma;\Delta, x{:}B\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta, x{:}A\oplus B\vdash x.\mathsf{case}(P,Q) :: z{:}C}$$

$$(!\mathsf{R})\ \frac{\Omega;\Gamma;\cdot\vdash P :: x{:}A}{\Omega;\Gamma;\cdot\vdash !z(x).P :: z{:}!A} \qquad (!\mathsf{L})\ \frac{\Omega;\Gamma, u{:}A;\Delta\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}!A\vdash P\{x/u\} :: z{:}C}$$

$$(\mathsf{copy})\ \frac{\Omega;\Gamma, u{:}A;\Delta, y{:}A\vdash P :: z{:}C}{\Omega;\Gamma, u{:}A;\Delta\vdash (\nu y)u\langle y\rangle.P :: z{:}C}$$

$$(\forall\mathsf{R})\ \frac{\Omega, X;\Gamma;\Delta\vdash P :: z{:}A}{\Omega;\Gamma;\Delta\vdash z(X).P :: z{:}\forall X.A} \qquad (\forall\mathsf{L})\ \frac{\Omega\vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta, x{:}A\{B/X\}\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}\forall X.A\vdash x\langle B\rangle.P :: z{:}C}$$

$$(\exists\mathsf{R})\ \frac{\Omega\vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta\vdash P :: z{:}A\{B/X\}}{\Omega;\Gamma;\Delta\vdash z\langle B\rangle.P :: z{:}\exists X.A} \qquad (\exists\mathsf{L})\ \frac{\Omega, X;\Gamma;\Delta, x{:}A\vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}\exists X.A\vdash x(X).P :: z{:}C}$$

$$(\mathsf{id})\ \frac{}{\Omega;\Gamma;x{:}A\vdash [x\leftrightarrow z] :: z{:}A} \qquad (\mathsf{cut})\ \frac{\Omega;\Gamma;\Delta_1\vdash P :: x{:}A \quad \Omega;\Gamma;\Delta_2, x{:}A\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2\vdash (\nu x)(P\mid Q) :: z{:}C}$$

$$(\mathsf{cut}^!)\ \frac{\Omega;\Gamma;\cdot\vdash P :: x{:}A \quad \Omega;\Gamma, u{:}A;\Delta\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta\vdash (\nu u)(!u(x).P\mid Q) :: z{:}C}$$

Fig. 2. Typing Rules

polymorphic type variables maintained in Ω. We use a well-formedness judgment $\Omega\vdash A\,\mathsf{type}$ which states that A is well-formed wrt the type variable environment Ω (i.e. $fv(A)\subseteq\Omega$). We often write T for the right-hand side typing $z{:}A$, $\cdot$ for the empty context and Δ,Δ' for the union of contexts Δ and Δ', only defined when Δ and Δ' are disjoint. We write $\cdot\vdash P :: T$ for $\cdot;\cdot;\cdot\vdash P :: T$.

Moreover, typing treats processes quotiented by structural congruence – given a well-typed process $\Omega;\Gamma;\Delta\vdash P :: T$, subject reduction ensures that for all possible reductions $P\xrightarrow{\tau}P'$, there exists a process Q where $P'\equiv Q$ such that $\Omega;\Gamma;\Delta\vdash Q :: T$. Related properties hold wrt general transitions $P\xrightarrow{\alpha}P'$. We refer the reader to [12, 13] for additional details on this matter.

As in [12, 13, 50, 76], the typing discipline enforces that channel outputs always have as object a *fresh* name, in the style of the internal mobility π-calculus [63]. We clarify a few of the key rules: Rule id types a linear forwarding between the sole ambient *linear* session $x{:}A$ and the offered session at channel z with the same type (the use of a non-empty Γ context embodies weakening of persistent resources). Rule $\forall$R defines the meaning of (impredicative) universal quantification over session types, stating that a session of type $\forall X.A$ inputs a type and then behaves uniformly as A; dually, to use such a session (rule $\forall$L), a process must output a type B which then warrants the use of the session as type $A\{B/X\}$. Rule $\multimap$R captures session input, where a session of type $A \multimap B$ expects to receive a session of type A which will then be used to produce a session of type B. Dually, session output (rule $\otimes$R) is achieved by producing a fresh session of type A (that uses a disjoint set of sessions to those of the continuation) and outputting the fresh session along z, which is then a session of type B. Rule !R types a process offering a session of type $!A$ along channel z, consisting of a replicated input along z which may be triggered an arbitrary (but finite) number of times. To preserve linearity, the replicated process cannot use any linear sessions. We note that the !R rule is often called the *promotion* rule in linear logic literature, whereas rule !L formalises the idea that a channel $u{:}A$ in the persistent context Γ is the same as a channel $x{:}!A$ in the linear context Δ. The use of a persistent session is captured by the copy rule: To use a persistent session u of type A, a process must output along u a fresh linear name y, triggering the replication and warranting the *linear* use of y as a session of type A. Proof-theoretically, copy corresponds to an instance of *dereliction* followed by *contraction*. Linear and persistent session composition is captured by rules cut and cut$^!$, respectively. The former enables a process that offers a session $x{:}A$ (using linear sessions in Δ_1) to be composed with a process that *uses* that session (amongst others in Δ_2) to offer $z{:}C$. The latter allows for a process that uses no linear sessions to be replicated and thus composed with processes that use the offered session in an unrestricted fashion. As shown in [11], typing entails Subject Reduction, Global Progress, and Termination.

The key properties of the typing system follow. For any P, we define *live*(P) iff $P \equiv (\nu\tilde{n})(\pi.Q \mid R)$, for some set of names $\tilde{n}$, process R, and *non-replicated* guarded process $\pi.Q$. We write $P \Downarrow$ if there is no infinite reduction sequence starting from P.

Theorem 2.4 (Properties of Well-Typed Processes [11]).

Subject Reduction *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $\Omega; \Gamma; \Delta \vdash Q :: z{:}A$.
Global Progress *If* $\vdash P :: z{:}\mathbf{1}$ *and* *live*(P), *there exists* Q *such that* $P \longrightarrow Q$.
Termination/Strong Normalisation *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then* $P \Downarrow$.

Observational Equivalences. We briefly summarise the typed congruence and logical equivalence with polymorphism, giving rise to a suitable notion of relational parametricity in the sense of Reynolds [59], defined as a contextual logical relation on typed processes [11]. The logical relation is reminiscent of a typed bisimulation. However, extra care is needed to ensure well-foundedness due to impredicative type instantiation. As a consequence, the logical relation allows us to reason about process equivalences where type variables are not instantiated with *the same*, but rather *related* types.

Typed Barbed Congruence ($\cong$). We use the typed contextual congruence from [11], which preserves *observable* actions, called barbs. In untyped process settings, barbed congruence is typically defined as the largest equivalence relation on processes, closed under all possible process contexts and internal actions, that preserves some basic notion of *observable*, called a barb. In our setting, following [11], we consider a typed variant of barbed congruence in which the notion of context is *typed*. Thus, typed barbed congruence is the largest equivalence relation on typed processes that is type-respecting, τ-closed, barb-preserving and contextual (for a suitable notion of typed context). We make these four notions precise. Thus, a relation is *contextual* if it is closed under any *typed* process context. A typed process context consists of a process with a typed hole (these can be mechanically derived from the typing rules by exhaustively considering all possibilities for typed holes). We omit the full details of defining typed contexts and refer the reader to the work of [11] for the full development.

Definition 2.5 (Type-respecting Relations [11]). A type-respecting relation over processes, written $\{\mathcal{R}_S\}_S$ is defined as a family of relations over processes indexed by typing S (i.e., S lists the left-hand context and right-hand typing information for processes in the relation). We often write $\mathcal{R}$ to refer to the whole family, and write $\Omega; \Gamma; \Delta \vdash P \mathcal{R} Q :: T$ to denote $\Omega; \Gamma; \Delta \vdash P, Q :: T$ and $(P, Q) \in \mathcal{R}_{\Omega;\Gamma;\Delta \vdash T}$.

We say that a type-respecting relation is an equivalence if it satisfies the usual properties of reflexivity, transitivity and symmetry. In the remainder of this development we often omit "type-respecting".

Definition 2.6 (τ-closed [11]). Relation $\mathcal{R}$ is *τ-closed* if $\Omega; \Gamma; \Delta \vdash P \mathcal{R} Q :: T$ and $P \longrightarrow P'$ imply there exists a Q' such that $Q \Longrightarrow Q'$ and $\Omega; \Gamma; \Delta \vdash P' \mathcal{R} Q' :: T$.

Our definition of basic observable on processes, or *barb*, is given below.

Definition 2.7 (Barbs [11]). Let $O_x = \{\overline{x}, x, \overline{x.\mathsf{inl}}, \overline{x.\mathsf{inr}}, x.\mathsf{inl}, x.\mathsf{inr}\}$ be the set of *basic observables* under name x. Given a well-typed process P, we write:

(i) $\mathsf{barb}(P, \overline{x})$, if $P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P'$;
(ii) $\mathsf{barb}(P, \overline{x})$, if $P \xrightarrow{\overline{x\langle A\rangle}} P'$, for some A, P';
(iii) $\mathsf{barb}(P, x)$, if $P \xrightarrow{x(A)} P'$, for some A, P';
(iv) $\mathsf{barb}(P, x)$, if $P \xrightarrow{x(y)} P'$, for some y, P';
(v) $\mathsf{barb}(P, \alpha)$, if $P \xrightarrow{\alpha} P'$, for some P' and $\alpha \in O_x \setminus \{x, \overline{x}\}$.

Given some $o \in O_x$, we write $\mathsf{wbarb}(P, o)$ if there exists a P' such that $P \Longrightarrow P'$ and $\mathsf{barb}(P', o)$ holds.

Definition 2.8 (Barb preserving relation). Relation $\mathcal{R}$ is a *barb preserving* if, for every name x, $\Omega; \Gamma; \Delta \vdash P \mathcal{R} Q :: T$ and $\mathsf{barb}(P, o)$ imply $\mathsf{wbarb}(Q, o)$, for any $o \in O_x$.

Definition 2.9 (Contextuality). A relation $\mathcal{R}$ is contextual if $\Omega; \Gamma; \Delta \vdash P \mathcal{R} Q :: T$ implies $\Omega; \Gamma; \Delta' \vdash C[P] \,\mathcal{R}\, C[Q] :: T'$, for every Δ' T' and typed context C.

Definition 2.10 (Barbed Congruence). Barbed congruence, noted $\cong$, is the largest equivalence on well-typed processes symmetric type-respecting relation that is τ-closed, barb preserving, and contextual.

Logical Equivalence ($\approx_{\mathsf{L}}$). The definition of logical equivalence is no more than a typed contextual bisimulation with the following intuitive reading: given two open processes P and Q (i.e. processes with non-empty left-hand side typings), we define their equivalence by inductively closing out the context, composing with equivalent processes offering appropriately typed sessions. When processes are closed, we have a single distinguished session channel along which we can perform observations, and proceed inductively on the structure of the offered session type. We can then show that such an equivalence satisfies the necessary fundamental properties (Theorem 2.13).

The logical relation is defined using the candidates technique of Girard [24]. In this setting, an *equivalence candidate* is a relation on typed processes satisfying basic closure conditions: an equivalence candidate must be compatible with barbed congruence and closed under forward and converse reduction.

Definition 2.11 (Equivalence Candidate). An *equivalence candidate* $\mathcal{R}$ at $z{:}A$ and $z{:}B$, noted $\mathcal{R} :: z{:}A{\Leftrightarrow}B$, is a binary relation on processes such that, for every $(P, Q) \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$ both $\cdot \vdash P :: z{:}A$ and $\cdot \vdash Q :: z{:}B$ hold, together with the following (we often write $(P, Q) \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$ as $P\,\mathcal{R}\,Q :: z{:}A{\Leftrightarrow}B$):

(1) If $(P, Q) \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$, $\cdot \vdash P \cong P' :: z{:}A$, and $\cdot \vdash Q \cong Q' :: z{:}B$ then $(P', Q') \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$.
(2) If $(P, Q) \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$ then, for all P_0 such that $\cdot \vdash P_0 :: z{:}A$ and $P_0 \Longrightarrow P$, we have $(P_0, Q) \in \mathcal{R} :: z{:}A{\Leftrightarrow}B$. Symmetrically for Q.

To define the logical relation we rely on some auxiliary notation, pertaining to the treatment of type variables arising due to impredicative polymorphism. We write $\omega : \Omega$ to denote a mapping ω that assigns a closed type to the type variables in Ω. We write $\omega(X)$ for the type mapped by ω to variable X. Given two mappings $\omega : \Omega$ and $\omega' : \Omega$, we define an equivalence candidate assignment η between ω and ω' as a mapping of equivalence candidate $\eta(X) :: -{:}\omega(X){\Leftrightarrow}\omega'(X)$ to the type variables in Ω, where the particular choice of a distinguished right-hand side channel is *delayed* (i.e. to be instantiated later on). We write $\eta(X)(z)$ for the instantiation of the (delayed) candidate with the name z. We write $\eta : \omega{\Leftrightarrow}\omega'$ to denote that η is a candidate assignment between ω and ω'; and $\hat{\omega}(P)$ to denote the application of mapping ω to P.

We define a sequent-indexed family of process relations, that is, a set of pairs of processes (P, Q), written $\Gamma; \Delta \vdash P \approx_{\mathsf{L}} Q :: T[\eta : \omega{\Leftrightarrow}\omega']$, satisfying some conditions, typed under $\Omega; \Gamma; \Delta \vdash T$, with $\omega : \Omega$, $\omega' : \Omega$ and $\eta : \omega{\Leftrightarrow}\omega'$. Logical equivalence is defined inductively on the size of the typing contexts and then on the structure of the right-hand side type.

Definition 2.12 (Logical Equivalence). **(Base Case)** Given a type A and mappings ω, ω', η, we define *logical equivalence*, noted $P \approx_{\mathsf{L}} Q :: z{:}A[\eta : \omega{\Leftrightarrow}\omega']$, as the smallest symmetric binary relation containing all pairs of processes (P, Q) such that (i) $\cdot \vdash \hat{\omega}(P) :: z{:}\hat{\omega}(A)$; (ii) $\cdot \vdash \hat{\omega}'(Q) :: z{:}\hat{\omega}'(A)$; and

(iii) satisfies the conditions given below we write $P \not\rightarrow$ to denote that P cannot reduce):

$$
\begin{array}{rcl}
P \approx_{\mathsf{L}} Q :: z{:}X[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & (P, Q) \in \eta(X)(z) \\
P \approx_{\mathsf{L}} Q :: z{:}\mathbf{1}[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', Q'.\ (P \Longrightarrow P' \wedge P' \not\rightarrow \wedge Q \Longrightarrow Q' \wedge Q' \not\rightarrow) \Rightarrow \\
 & & (P' \equiv_{!} \mathbf{0} \wedge Q' \equiv_{!} \mathbf{0}) \\
P \approx_{\mathsf{L}} Q :: z{:}A \multimap B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', y.\ (P \stackrel{z(y)}{\rightarrow} P') \Rightarrow \exists Q'. Q \stackrel{z(y)}{\Longrightarrow} Q' \ s.t. \\
 & & \forall R_1, R_2.\ R_1 \approx_{\mathsf{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \\
 & & (\nu y)(P' \mid R_1) \approx_{\mathsf{L}} (\nu y)(Q' \mid R_2) :: z{:}B[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathsf{L}} Q :: z{:}A \otimes B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', y.\ (P \stackrel{\overline{(\nu y)z\langle y \rangle}}{\rightarrow} P') \Rightarrow \exists Q'. Q \stackrel{\overline{(\nu y)z\langle y \rangle}}{\Longrightarrow} Q' \ s.t. \\
 & & \exists P_1, P_2, Q_1, Q_2. P' \equiv_{!} P_1 \mid P_2 \wedge Q' \equiv_{!} Q_1 \mid Q_2 \\
 & & P_1 \approx_{\mathsf{L}} Q_1 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \wedge P_2 \approx_{\mathsf{L}} Q_2 :: z{:}B[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathsf{L}} Q :: z{:}!A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P'.\ (P \stackrel{z(y)}{\rightarrow} P') \Rightarrow \exists Q'. Q \stackrel{z(y)}{\Longrightarrow} Q' \wedge \\
 & & \quad P' \approx_{\mathsf{L}} Q' :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathsf{L}} Q :: z{:}A \,\&\, B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \\
(\forall P'. (P \stackrel{z.\mathsf{inl}}{\rightarrow} P') & \Rightarrow & \exists Q'. (Q \stackrel{z.\mathsf{inl}}{\Longrightarrow} Q' \wedge P' \approx_{\mathsf{L}} Q' :: z{:}A[\eta : \omega \Leftrightarrow \omega'])) \wedge \\
(\forall P'. (P \stackrel{z.\mathsf{inr}}{\rightarrow} P') & \Rightarrow & \exists Q'. (Q \stackrel{z.\mathsf{inr}}{\Longrightarrow} Q' \wedge P' \approx_{\mathsf{L}} Q' :: z{:}B[\eta : \omega \Leftrightarrow \omega'])) \\
P \approx_{\mathsf{L}} Q :: z{:}A \oplus B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \\
(\forall P'. (P \stackrel{\overline{z.\mathsf{inl}}}{\rightarrow} P') & \Rightarrow & \exists Q'. (Q \stackrel{\overline{z.\mathsf{inl}}}{\Longrightarrow} Q' \wedge P' \approx_{\mathsf{L}} Q' :: z{:}A[\eta : \omega \Leftrightarrow \omega'])) \wedge \\
(\forall P'. (P \stackrel{\overline{z.\mathsf{inr}}}{\rightarrow} P') & \Rightarrow & \exists Q'. (Q \stackrel{\overline{z.\mathsf{inr}}}{\Longrightarrow} Q' \wedge P' \approx_{\mathsf{L}} Q' :: z{:}B[\eta : \omega \Leftrightarrow \omega'])) \\
P \approx_{\mathsf{L}} Q :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall B_1, B_2, P', \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \stackrel{z(B_1)}{\rightarrow} P') \ implies \\
\exists Q'. Q \stackrel{z(B_2)}{\Longrightarrow} Q',\ P' & \approx_{\mathsf{L}} & Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]] \\
P \approx_{\mathsf{L}} Q :: z{:}\exists X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \exists B_1, B_2, \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \stackrel{\overline{z\langle B_1 \rangle}}{\rightarrow} P') \ implies \\
\exists Q'. Q \stackrel{\overline{z\langle B_2 \rangle}}{\Longrightarrow} Q',\ P' & \approx_{\mathsf{L}} & Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]]
\end{array}
$$

(Inductive Case) Let Γ, Δ be non empty. Given $\Omega; \Gamma; \Delta \vdash P :: T$ and $\Omega; \Gamma; \Delta \vdash Q :: T$, the binary relation on processes $\Gamma; \Delta \vdash P \approx_{\mathsf{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$ (with $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$) is inductively defined as:

$$
\begin{array}{l}
\Gamma; \Delta, y : A \vdash P \approx_{\mathsf{L}} Q :: T[\eta : \omega \Leftrightarrow \omega'] \quad \text{iff} \quad \forall R_1, R_2.\ \text{s.t. } R_1 \approx_{\mathsf{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'], \\
\qquad\qquad \Gamma; \Delta \vdash (\nu y)(\hat{\omega}(P) \mid \hat{\omega}(R_1)) \approx_{\mathsf{L}} (\nu y)(\hat{\omega}'(Q) \mid \hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega'] \\
\Gamma, u : A; \Delta \vdash P \approx_{\mathsf{L}} Q :: T[\eta : \omega \Leftrightarrow \omega'] \quad \text{iff} \quad \forall R_1, R_2.\ \text{s.t. } R_1 \approx_{\mathsf{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'], \\
\qquad \Gamma; \Delta \vdash (\nu u)(\hat{\omega}(P) \mid !u(y).\hat{\omega}(R_1)) \approx_{\mathsf{L}} (\nu u)(\hat{\omega}'(Q) \mid !u(y).\hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega']
\end{array}
$$

For the sake of readability we often omit the $\eta : \omega \Leftrightarrow \omega'$ portion of $\approx_{\mathsf{L}}$, which is henceforth implicitly universally quantified. Thus, we write $\Omega; \Gamma; \Delta \vdash P \approx_{\mathsf{L}} Q :: z{:}A$ (or $P \approx_{\mathsf{L}} Q$) iff the two given processes are logically equivalent for all consistent instantiations of its type variables.

It is instructive to inspect the clause for type input ($\forall X.A$): the two processes must be able to match inputs of any pair of *related* types (i.e. types related by a candidate), such that the continuations are related at the open type A with the appropriate type variable instantiations, following Girard [24]. The power of this style of logical relation arises from a combination of the extensional flavour of the equivalence and the fact that polymorphic equivalences do not require the same type to be instantiated in both processes, but rather that the types are *related* (via a suitable equivalence candidate relation).

Theorem 2.13 (Properties of Logical Equivalence [11]).

Parametricity: *If* $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ *then, for all* $\omega, \omega' : \Omega$ *and* $\eta : \omega{\Leftrightarrow}\omega'$, *we have* $\Gamma;\Delta \vdash \hat{\omega}(P) \approx_{\mathsf{L}} \hat{\omega}'(P) :: z{:}A[\eta : \omega{\Leftrightarrow}\omega']$.
Soundness: *If* $\Omega;\Gamma;\Delta \vdash P \approx_{\mathsf{L}} Q :: z{:}A$ *then* $C[P] \cong C[Q] :: z{:}A$, *for any closing* $C[-]$.
Completeness: *If* $\Omega;\Gamma;\Delta \vdash P \cong Q :: z{:}A$ *then* $\Omega;\Gamma;\Delta \vdash P \approx_{\mathsf{L}} Q :: z{:}A$.

The contextual nature of logical equivalence (and thus of typed barbed congruence) admits what may at first seem as exotic equivalences from a concurrency perspective. For instance, the following *can* be a valid equivalence: $x(a).(\nu b)y\langle b\rangle.(P_1 \mid P_2) \approx_{\mathsf{L}} (\nu b)y\langle b\rangle.(P_1 \mid x(a).P_2)$. To argue why such prefix commutations are reasonable, we first consider a possible typing for such processes:

$$\dfrac{\dfrac{\cdot;\cdot;\cdot \vdash P_1 :: b:C \quad \cdot;\cdot;a{:}A, x{:}B \vdash P_2 :: y{:}D}{\cdot;\cdot;a{:}A, x{:}B \vdash (\nu b)y\langle b\rangle.(P_1 \mid P_2) :: y{:}C \otimes D}\ (\otimes\mathsf{R})}{\cdot;\cdot;x{:}A \otimes B \vdash x(a).(\nu b)y\langle b\rangle.(P_1 \mid P_2) :: y{:}C \otimes D}\ (\otimes\mathsf{L})$$

$$\dfrac{\cdot;\cdot;\cdot \vdash P_1 :: b:C \quad \dfrac{\cdot;\cdot;a{:}A, x{:}B \vdash P_2 :: y{:}D}{\cdot;\cdot;x{:}A \otimes B \vdash x(a).P :: y{:}D}\ (\otimes\mathsf{L})}{\cdot;\cdot;x{:}A \otimes B \vdash (\nu b)y\langle b\rangle.(P_1 \mid x(a).P_2) :: y{:}C \otimes D}\ (\otimes\mathsf{R})$$

To type the first process we first apply rule $\otimes$L, receiving on x and then rule $\otimes$R to send on y accordingly. To type the second process, we apply the same rules in reverse order. Why is it then reasonable to equate the two processes through logical equivalence? Both processes are typed in a context that must provide a session $x{:}A \otimes B$ so that the processes may offer $y{:}C \otimes D$. Let us posit a process $Q :: x{:}A \otimes B$, we can compose Q with the given processes via the cut rule to then have $(\nu x)(Q \mid x(a).(\nu b)y\langle b\rangle.(P_1 \mid P_2))$ and $(\nu x)(Q \mid (\nu b)y\langle b\rangle.(P_1 \mid x(a).P_2))$, respectively, both offering $y{:}C \otimes D$ in the empty context. Now the contextual nature of the equivalence becomes clear: since both processes are typed in a context requiring $x{:}A \otimes B$, they must be reasoned about as if their contextual requirements are satisfied. In this setting, the channel x is now hidden by the ν-binder and therefore no actions on x are visible, only those on y (the right-hand side typing). Thus, it is *impossible* for any well-typed process (and any well-typed context) to distinguish between the two processes, and so the equivalence is justified.

We further note that if $P_1 \equiv \mathbf{0}$ and $C = \mathbf{1}$, we can specialize the equivalence to the seemingly more exotic $x(a).(\nu b)y\langle b\rangle.P_2 \equiv (\nu b)y\langle b\rangle.x(a).P_2$, or, if $C = D = \mathbf{1}$ and $P_1 \equiv \mathbf{0}$, we can even derive $x(a).(\nu b)y\langle b\rangle.P_2 \equiv (\nu b)y\langle b\rangle.\mathbf{0} \mid x(a).P_2$. Neither of these are derivable in the general case, albeit all are perfectly justified given the typed *and* contextual nature of logical equivalence (and barbed congruence). A more complete discussion of commuting conversions and their interpretation as behavioural equivalences can be found in [11, 50, 51].

3 TO LINEAR-F AND BACK

We now develop our mutually inverse and fully abstract encodings between Polyπ and a linear polymorphic λ-calculus [79] that we dub Linear-F. We first introduce the syntax and typing of the linear λ-calculus and then proceed to detail our encodings and their properties (we omit typing ascriptions from the existential polymorphism constructs for readability).

Definition 3.1 (Linear-F). The syntax of terms M, N and types A, B of Linear-F is given below.

$$\begin{array}{rcl} M, N & ::= & \lambda x{:}A.M \mid M\,N \mid \langle M \otimes N\rangle \mid \mathsf{let}\ x \otimes y = M\ \mathsf{in}\ N \mid\ !M \mid \mathsf{let}\ !u = M\ \mathsf{in}\ N \mid \Lambda X.M \\ & \mid & M[A] \mid \mathsf{pack}\ A\ \mathsf{with}\ M \mid \mathsf{let}\ (X, y) = M\ \mathsf{in}\ N \mid \mathsf{let}\ \mathbf{1} = M\ \mathsf{in}\ N \mid \langle\rangle \mid \mathsf{T} \mid \mathsf{F} \\ A, B & ::= & A \multimap B \mid A \otimes B \mid\ !A \mid \forall X.A \mid \exists X.A \mid X \mid \mathbf{1} \mid \mathbf{2} \end{array}$$

$$(\textsc{var})\ \frac{}{\Omega;\Gamma;x{:}A \vdash x{:}A} \qquad (\multimap I)\ \frac{\Omega;\Gamma;\Delta,x{:}A \vdash M : B}{\Omega;\Gamma;\Delta \vdash \lambda x{:}A.M : A \multimap B} \qquad (\multimap E)\ \frac{\Omega;\Gamma;\Delta \vdash M : A \multimap B \quad \Omega;\Gamma;\Delta' \vdash N : A}{\Omega;\Gamma;\Delta,\Delta' \vdash M\,N : B}$$

$$(\otimes I)\ \frac{\Omega;\Gamma;\Delta \vdash M : A \quad \Omega;\Gamma;\Delta' \vdash N : B}{\Omega;\Gamma;\Delta,\Delta' \vdash \langle M \otimes N\rangle : A \otimes B} \qquad (\otimes E)\ \frac{\Omega;\Gamma;\Delta \vdash M : A \otimes B \quad \Omega;\Gamma;\Delta',x{:}A,y{:}B \vdash N : B'}{\Omega;\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ x \otimes y = M\ \mathsf{in}\ N : B'}$$

$$(!I)\ \frac{\Omega;\Gamma;\cdot \vdash M : A}{\Omega;\Gamma;\cdot \vdash !M : !A} \qquad (!E)\ \frac{\Omega;\Gamma;\Delta \vdash M : !A \quad \Omega;\Gamma,u{:}A;\Delta' \vdash N : B}{\Omega;\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ !u = M\ \mathsf{in}\ N : B} \qquad (\textsc{uvar})\ \frac{}{\Omega;\Gamma,u{:}A;\cdot \vdash u{:}A}$$

$$(\forall I)\ \frac{\Omega,X;\Gamma;\Delta \vdash M : A}{\Omega;\Gamma;\Delta \vdash \Lambda X.M : \forall X.A} \qquad (\forall E)\ \frac{\Omega \vdash A\ \mathsf{type} \quad \Omega;\Gamma;\Delta \vdash M : \forall X.B}{\Omega;\Gamma;\Delta \vdash M[A] : B\{A/X\}}$$

$$(\exists I)\ \frac{\Omega \vdash A\ \mathsf{type} \quad \Omega;\Gamma;\Delta \vdash M : B\{A/X\}}{\Omega;\Gamma;\Delta \vdash \mathsf{pack}\ A\ \mathsf{with}\ M : \exists X.B} \qquad (\exists E)\ \frac{\Omega;\Gamma;\Delta \vdash M : \exists X.A \quad \Omega,X;\Gamma;\Delta',y{:}A \vdash N : B \quad \Omega \vdash B\ \mathsf{type}}{\Omega;\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ (X,y) = M\ \mathsf{in}\ N : B}$$

$$(\mathbf{1}I)\ \frac{}{\Omega;\Gamma;\cdot \vdash \langle\rangle : \mathbf{1}} \qquad (\mathbf{1}E)\ \frac{\Omega;\Gamma;\Delta \vdash M : \mathbf{1} \quad \Omega;\Gamma;\Delta' \vdash N : C}{\Omega;\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ \mathbf{1} = M\ \mathsf{in}\ N : C} \qquad (\mathbf{2}I_1)\ \frac{}{\Omega;\Gamma;\cdot \vdash \mathsf{T} : \mathbf{2}} \qquad (\mathbf{2}I_2)\ \frac{}{\Omega;\Gamma;\cdot \vdash \mathsf{F} : \mathbf{2}}$$

Fig. 3. Linear-F Typing Rules

The syntax of types is that of the multiplicative and exponential fragments of second-order intuitionistic linear logic. The term assignment is mostly standard: $\lambda x{:}A.M$ denotes linear λ-abstractions; $M\,N$ denotes the application; $\langle M \otimes N\rangle$ denotes the multiplicative pairing of M and N, as reflected in its elimination form $\mathsf{let}\ x \otimes y = M\ \mathsf{in}\ N$ which simultaneously deconstructs the pair M, binding its first and second projection to x and y in N, respectively; $!M$ denotes a term M that does not use any linear variables and so may be used an arbitrary number of times; $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ binds the underlying exponential term of M as u in N; $\Lambda X.M$ is the type abstraction former; $M[A]$ stands for type application; $\mathsf{pack}\ A\ \mathsf{with}\ M$ is the existential type introduction form, where M is a term where the existentially typed variable is instantiated with A; $\mathsf{let}\ (X,y) = M\ \mathsf{in}\ N$ unpacks an existential package M, binding the representation type to X and the underlying term to y in N; the multiplicative unit $\mathbf{1}$ has as introduction form the nullary pair $\langle\rangle$ and is eliminated by the construct $\mathsf{let}\ \mathbf{1} = M\ \mathsf{in}\ N$, where M is a term of type $\mathbf{1}$. Booleans (type $\mathbf{2}$ with values T and F) are the basic observable.

The typing judgment in Linear-F is given as $\Omega;\Gamma;\Delta \vdash M : A$, following the DILL formulation of linear logic [5], stating that term M has type A in a linear context Δ (i.e. bindings for linear variables $x{:}B$), intuitionistic context Γ (i.e. binding for intuitionistic variables $u{:}B$) and type variable context Ω. The typing rules are given in Figure 3.

The operational semantics of the calculus are the expected call-by-name semantics [39, 79], given in Figure 4. For conciseness we use a evaluation context to codify the various congruence rules, where $E[M]$ stands for the instantiation of the single hole $\bullet$ in context E with the term M. We write $\Downarrow$ for the usual evaluation relation.

We write $\cong$ for the largest typed congruence that is consistent with the observables of type $\mathbf{2}$ (i.e. a so-called Morris-style equivalence as in [8]).

$$\frac{}{(\lambda x{:}A.M)\,N \longrightarrow M\{N/x\}} \qquad \frac{}{\mathsf{let}\ !u = !M\ \mathsf{in}\ N \longrightarrow N\{M/u\}}$$

$$\frac{}{(\Lambda X.M)[A] \longrightarrow M\{A/X\}} \qquad \frac{}{\mathsf{let}\ x \otimes y = \langle M_1 \otimes M_2\rangle\ \mathsf{in}\ N \longrightarrow N\{M_1/x, M_2/y\}}$$

$$\frac{}{\mathsf{let}\ (X, y) = \mathsf{pack}\ A\ \mathsf{with}\ M\ \mathsf{in}\ N \longrightarrow N\{A/X, M/y\}} \qquad \frac{}{\mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ M \longrightarrow M}$$

$$\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

$$\begin{array}{lcl} E & ::= & \bullet \mid E\,M \mid \mathsf{let}\ \mathbf{1} = E\ \mathsf{in}\ M \mid \mathsf{let}\ \mathbf{1} = M\ \mathsf{in}\ E \mid \mathsf{let}\ !u = M\ \mathsf{in}\ E \mid \mathsf{let}\ !u = E\ \mathsf{in}\ M \\ & \mid & \mathsf{let}\ x \otimes y = E\ \mathsf{in}\ M \mid \langle E \otimes M\rangle \mid \langle M \otimes E\rangle \end{array}$$

Fig. 4. Operational Semantics of Linear-F

3.1 Encoding Linear-F into Session π-Calculus

We define a translation from Linear-F to Polyπ generalising the one from [70], accounting for polymorphism and multiplicative pairs. We translate typing derivations of λ-terms to those of π-calculus terms (we omit the full typing derivation for the sake of readability).

Proof theoretically, the λ-calculus corresponds to a proof term assignment for natural deduction presentations of logic, whereas the session π-calculus from § 2 corresponds to a proof term assignment for sequent calculus. Thus, we obtain a translation from λ-calculus to the session π-calculus by considering the proof theoretic content of the constructive proof of soundness of the sequent calculus wrt natural deduction. Following Gentzen [22], the translation from natural deduction to sequent calculus maps introduction rules to the corresponding right rules and elimination rules to a combination of the corresponding left rule, cut and/or identity.

Since typing in the session calculus identifies a distinguished channel along which a process offers a session, the translation of λ-terms is parameterised by a "result" channel along which the behaviour of the λ-term is implemented. Given a λ-term M, the process $[\![M]\!]_z$ encodes the behaviour of M along the session channel z. We enforce that the type $\mathbf{2}$ of booleans and its two constructors are consistently translated to their polymorphic Church encodings before applying the translation to Polyπ. Thus, type $\mathbf{2}$ is first translated to $\forall X.!X \multimap !X \multimap X$, the value T to $\Lambda X.\lambda u{:}!X.\lambda v{:}!X.\mathsf{let}\ !x = u\ \mathsf{in}\ \mathsf{let}\ !y = v\ \mathsf{in}\ x$ and the value F to $\Lambda X.\lambda u{:}!X.\lambda v{:}!X.\mathsf{let}\ !x = u\ \mathsf{in}\ \mathsf{let}\ !y = v\ \mathsf{in}\ y$. Such representations of the booleans are adequate up to parametricity [10] and suitable for our purposes of relating the session calculus (which has no primitive notion of value or result type) with the λ-calculus precisely due to the tight correspondence between the two calculi.

Definition 3.2 (From Linear-F to Polyπ). $[\![\Omega]\!]; [\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![M]\!]_z :: z{:}A$ denotes the translation of contexts, types and terms from Linear-F to the polymorphic session calculus. The translations on contexts and types are the identity function. Booleans and their values are first translated to their (typed) Church encodings, that is, type $\mathbf{2}$ is translated to type $\forall X.!X \multimap !X \multimap X$, the value T to $\Lambda X.\lambda u{:}!X.\lambda v{:}!X.\mathsf{let}\ !x = u\ \mathsf{in}\ \mathsf{let}\ !y = v\ \mathsf{in}\ x$ and value F to $\Lambda X.\lambda u{:}!X.\lambda v{:}!X.\mathsf{let}\ !x = u\ \mathsf{in}\ \mathsf{let}\ !y = v\ \mathsf{in}\ y$, as specified above. The translation on λ-terms is given below:

$$
\begin{array}{lllllll}
[\![x]\!]_z & \triangleq & [x \leftrightarrow z] & [\![M\,N]\!]_z & \triangleq & (\nu x)([\![M]\!]_x \mid (\nu y)x\langle y\rangle.([\![N]\!]_y \mid [x \leftrightarrow z])) \\
[\![u]\!]_z & \triangleq & (\nu x)u\langle x\rangle.[x \leftrightarrow z] & [\![\mathsf{let}\,!u = M\,\mathsf{in}\,N]\!]_z & \triangleq & (\nu x)([\![M]\!]_x \mid [\![N]\!]_z\{x/u\}) \\
[\![\lambda x{:}A.M]\!]_z & \triangleq & z(x).[\![M]\!]_z & [\![\langle M \otimes N\rangle]\!]_z & \triangleq & (\nu y)z\langle y\rangle.([\![M]\!]_y \mid [\![N]\!]_z) \\
[\![!M]\!]_z & \triangleq & !z(x).[\![M]\!]_x & [\![\mathsf{let}\,x \otimes y = M\,\mathsf{in}\,N]\!]_z & \triangleq & (\nu y)([\![M]\!]_y \mid y(x).[\![N]\!]_z) \\
[\![\Lambda X.M]\!]_z & \triangleq & z(X).[\![M]\!]_z & [\![M[A]]\!]_z & \triangleq & (\nu x)([\![M]\!]_x \mid x\langle A\rangle.[x \leftrightarrow z]) \\
[\![\mathsf{pack}\,A\,\mathsf{with}\,M]\!]_z & \triangleq & z\langle A\rangle.[\![M]\!]_z & [\![\mathsf{let}\,(X,y) = M\,\mathsf{in}\,N]\!]_z & \triangleq & (\nu y)([\![M]\!]_y \mid y(X).[\![N]\!]_z) \\
[\![\langle\rangle]\!]_z & \triangleq & \mathbf{0} & [\![\mathsf{let}\,\mathbf{1} = M\,\mathsf{in}\,N]\!]_z & \triangleq & (\nu x)([\![M]\!]_x \mid [\![N]\!]_z)
\end{array}
$$

To translate a (linear) λ-abstraction $\lambda x{:}A.M$, which corresponds to the proof term for the introduction rule for $\multimap$, we map it to the corresponding $\multimap$R rule, thus obtaining a process $z(x).[\![M]\!]_z$ that inputs along the result channel z a channel x which will be used in $[\![M]\!]_z$ to access the function argument. To encode the application $M\,N$, we compose (i.e. cut) $[\![M]\!]_x$, where x is a fresh name, with a process that provides the (encoded) function argument by outputting along x a channel y which offers the behaviour of $[\![N]\!]_y$. After the output is performed, the type of x is now that of the function's codomain and thus we conclude by forwarding (i.e. the id rule) between x and the result channel z.

The encoding for polymorphism follows a similar pattern: To encode the abstraction $\Lambda X.M$, we receive along the result channel a type that is bound to X and proceed inductively. To encode type application $M[A]$ we encode the abstraction M in parallel with a process that sends A to it, and forwards accordingly. Finally, the encoding of the existential package $\mathsf{pack}\,A\,\mathsf{with}\,M$ maps to an output of the type A followed by the behaviour $[\![M]\!]_z$, with the encoding of the elimination form $\mathsf{let}\,(X,y) = M\,\mathsf{in}\,N$ composing the translation of the term of existential type M with a process performing the appropriate type input and proceeding as $[\![N]\!]_z$.

Computation in the λ-calculus entails substitution of variables with terms whereas communication in the π-calculus substitutes names for names. Thus, we observe that the encoding of $M\{N/x\}$ is identified with $(\nu x)([\![M]\!]_z \mid [\![N]\!]_x)$. Similarly, the encoding of $M\{N/u\}$ corresponds to $(\nu u)(!u(x).[\![N]\!]_x \mid [\![M]\!]_z)$.

Example 3.3 (Encoding of Linear-F). Consider the following λ-term corresponding to a polymorphic pairing function (recall that we write $\overline{z}\langle w\rangle.P$ for $(\nu w)z\langle w\rangle.P$):

$$M \triangleq \Lambda X.\Lambda Y.\lambda x{:}X.\lambda y{:}Y.\langle x \otimes y\rangle \quad \text{and} \quad N \triangleq ((M[A][B]\,M_1)\,M_2)$$

Then we have, with $\tilde{x} = x_1x_2x_3x_4$:

$$
\begin{array}{lll}
[\![N]\!]_z & \equiv & (\nu\tilde{x})(\;[\![M]\!]_{x_1} \mid x_1\langle A\rangle.[x_1 \leftrightarrow x_2] \mid x_2\langle B\rangle.[x_2 \leftrightarrow x_3] \mid \\
& & \quad \overline{x_3}\langle x\rangle.([\![M_1]\!]_x \mid [x_3 \leftrightarrow x_4]) \mid \overline{x_4}\langle y\rangle.([\![M_2]\!]_y \mid [x_4 \leftrightarrow z])) \\
& \equiv & (\nu\tilde{x})(\;x_1(X).x_1(Y).x_1(x).x_1(y).\overline{x_1}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow x_1]) \mid x_1\langle A\rangle.[x_1 \leftrightarrow x_2] \mid \\
& & \quad x_2\langle B\rangle.[x_2 \leftrightarrow x_3] \mid \overline{x_3}\langle x\rangle.([\![M_1]\!]_x \mid [x_3 \leftrightarrow x_4]) \mid \overline{x_4}\langle y\rangle.([\![M_2]\!]_y \mid [x_4 \leftrightarrow z]))
\end{array}
$$

We can observe that $N \rightarrow^+ (((\lambda x{:}A.\lambda y{:}B.\langle x \otimes y\rangle)\,M_1)\,M_2) \rightarrow^+ \langle M_1 \otimes M_2\rangle$. At the process level, each reduction corresponding to the redex of type application is simulated by two reductions, obtaining:

$$
\begin{array}{lll}
[\![N]\!]_z & \rightarrow^+ & (\nu x_3, x_4)(\;x_3(x).x_3(y).\overline{x_3}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow x_3]) \mid \\
& & \quad \overline{x_3}\langle x\rangle.([\![M_1]\!]_x \mid [x_3 \leftrightarrow x_4]) \mid \overline{x_4}\langle y\rangle.([\![M_2]\!]_y \mid [x_4 \leftrightarrow z])) = P
\end{array}
$$

The reductions corresponding to the β-redexes clarify the way in which the encoding represents substitution of terms for variables via fine-grained name passing. Consider $[\![\langle M_1 \otimes M_2\rangle]\!]_z \triangleq \overline{z}\langle w\rangle.([\![M_1]\!]_w \mid [\![M_2]\!]_z)$ and

$$P \rightarrow^+ (\nu x, y)([\![M_1]\!]_x \mid [\![M_2]\!]_y \mid \overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]))$$

The encoding of the pairing of M_1 and M_2 outputs a fresh name w which will denote the behaviour of (the encoding of) M_1, and then the behaviour of the encoding of M_2 is offered on z. The reduct of P outputs a fresh name w which is then identified with x and thus denotes the behaviour of $[\![M_1]\!]_w$. The channel z is identified with y and thus denotes the behaviour of $[\![M_2]\!]_z$, making the two processes listed above equivalent. This informal reasoning exposes the insights that justify the operational correspondence of the encoding. Proof-theoretically, these equivalences simply map to commuting conversions which push the processes $[\![M_1]\!]_x$ and $[\![M_2]\!]_z$ under the output on z.

We note that in Theorem 3.5 (and in the subsequent development) we distinguish between the soundness and completeness directions of operational correspondence (c.f. [25]).

Lemma 3.4 (Compositionality).

(1) *Let* $\Omega; \Gamma; \Delta_1, x{:}A \vdash M : B$ *and* $\Omega; \Gamma; \Delta_2 \vdash N : A$. *We have that* $\Omega; \Gamma; \Delta_1, \Delta_2 \vdash [\![M\{N/x\}]\!]_z \approx_{\mathsf{L}} (\nu x)([\![M]\!]_z \mid [\![N]\!]_x) :: z{:}B$.
(2) *Let* $\Omega; \Gamma, u{:}A; \Delta \vdash M : B$ *and* $\Omega; \Gamma; \cdot \vdash N : A$. *we have that* $\Omega; \Gamma; \Delta \vdash [\![M\{N/u\}]\!]_z \approx_{\mathsf{L}} (\nu u)([\![M]\!]_z \mid !u(x).[\![N]\!]_x) :: z{:}B$.

Proof. By induction on the structure of M, exploiting the fact that commuting conversions and $\equiv_!$ are sound $\approx_{\mathsf{L}}$ equivalences. See Lemma 5.2 for further details. □

Theorem 3.5 (Operational Correspondence). *Let* $\Omega; \Gamma; \Delta \vdash M : A$.

Completeness: *If* $M \to N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $[\![N]\!]_z \approx_{\mathsf{L}} P$
Soundness: *If* $[\![M]\!]_z \to P$ *then* $M \to^+ N$ *and* $[\![N]\!]_z \approx_{\mathsf{L}} P$

3.2 Encoding Session π-calculus to Linear-F

Just as the proof theoretic content of the soundness of sequent calculus wrt natural deduction induces a translation from λ-terms to session-typed processes, the *completeness* of the sequent calculus wrt natural deduction induces a translation from the session calculus to the λ-calculus. For conciseness, we omit the additive types $\oplus$ and & from the translation, which can be straightforwardly considered by adding the corresponding additive pairs and sums to Linear-F. This mapping identifies sequent calculus right rules with the introduction rules of natural deduction and left rules with elimination rules combined with (type-preserving) substitution. Crucially, the mapping is defined on *typing derivations*, enabling us to consistently identify when a process uses a session (i.e. left rules) or, dually, when a process offers a session (i.e. right rules). The encoding makes use of the two admissible substitution principles denoted by the following rules:

$$\text{(subst)}\ \frac{\Omega; \Gamma; \Delta_1, x{:}B \vdash M : A \quad \Omega; \Gamma; \Delta_2 \vdash N : B}{\Omega; \Gamma; \Delta_1, \Delta_2 \vdash M\{N/x\} : A} \qquad \text{(subst}^!\text{)}\ \frac{\Omega; \Gamma, u{:}B; \Delta \vdash M : A \quad \Omega; \Gamma; \cdot \vdash N : B}{\Omega; \Gamma; \Delta \vdash M\{N/u\} : A}$$

Definition 3.6 (From Polyπ to Linear-F). We write $(\!|\Omega|\!); (\!|\Gamma|\!); (\!|\Delta|\!) \vdash (\!|P|\!) : A$ for the translation from typing derivations in Polyπ to derivations in Linear-F. The translations on types and contexts are the identity function. The translation on processes is given below, where the leftmost column indicates the typing rule at the root of the derivation (Figures 5 and 6 list the translation on typing derivations, where we write $\llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta\vdash z:A}$ to denote the translation of $\Omega;\Gamma;\Delta \vdash P :: z{:}A$).

(id)	$\llparenthesis [x \leftrightarrow y] \rrparenthesis$	$\triangleq$	x	(copy)	$\llparenthesis (\nu x)u\langle x\rangle.P \rrparenthesis$	$\triangleq$	$\llparenthesis P \rrparenthesis\{u/x\}$
(**1**R)	$\llparenthesis \mathbf{0} \rrparenthesis$	$\triangleq$	$\langle\rangle$	(**1**L)	$\llparenthesis P \rrparenthesis$	$\triangleq$	$\mathsf{let}\ \mathbf{1} = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$
($\multimap$R)	$\llparenthesis z(x).P \rrparenthesis$	$\triangleq$	$\lambda x{:}A.\llparenthesis P \rrparenthesis$	($\multimap$L)	$\llparenthesis (\nu y)x\langle y\rangle.(P \mid Q) \rrparenthesis$	$\triangleq$	$\llparenthesis Q \rrparenthesis\{(x\ \llparenthesis P \rrparenthesis)/x\}$
($\otimes$R)	$\llparenthesis (\nu x)z\langle x\rangle.(P \mid Q) \rrparenthesis$	$\triangleq$	$\langle \llparenthesis P \rrparenthesis \otimes \llparenthesis Q \rrparenthesis \rangle$	($\otimes$L)	$\llparenthesis x(y).P \rrparenthesis$	$\triangleq$	$\mathsf{let}\ x \otimes y = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$
(!R)	$\llparenthesis !z(x).P \rrparenthesis$	$\triangleq$	$!\llparenthesis P \rrparenthesis$	(!L)	$\llparenthesis P\{u/x\} \rrparenthesis$	$\triangleq$	$\mathsf{let}\ !u = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$
($\forall$R)	$\llparenthesis z(X).P \rrparenthesis$	$\triangleq$	$\Lambda X.\llparenthesis P \rrparenthesis$	($\forall$L)	$\llparenthesis x\langle B\rangle.P \rrparenthesis$	$\triangleq$	$\llparenthesis P \rrparenthesis\{(x[B])/x\}$
($\exists$R)	$\llparenthesis z\langle B\rangle.P \rrparenthesis$	$\triangleq$	$\mathsf{pack}\ B\ \mathsf{with}\ \llparenthesis P \rrparenthesis$	($\exists$L)	$\llparenthesis x(Y).P \rrparenthesis$	$\triangleq$	$\mathsf{let}\ (Y,x) = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$
(cut)	$\llparenthesis (\nu x)(P \mid Q) \rrparenthesis$	$\triangleq$	$\llparenthesis Q \rrparenthesis\{\llparenthesis P \rrparenthesis/x\}$	($\text{cut}^!$)	$\llparenthesis (\nu u)(!u(x).P \mid Q) \rrparenthesis$	$\triangleq$	$\llparenthesis Q \rrparenthesis\{\llparenthesis P \rrparenthesis/u\}$

For instance, the encoding of a process $z(x).P :: z{:}A \multimap B$, typed by rule $\multimap$R, results in the corresponding $\multimap I$ introduction rule in the λ-calculus and thus is $\lambda x{:}A.\llparenthesis P \rrparenthesis$. To encode the process $(\nu y)x\langle y\rangle.(P \mid Q)$, typed by rule $\multimap$L, we make use of substitution: Given that the sub-process Q is typed as $\Omega;\Gamma;\Delta', x{:}B \vdash Q :: z{:}C$, the encoding of the full process is given by $\llparenthesis Q \rrparenthesis\{(x\ \llparenthesis P \rrparenthesis)/x\}$. The term $x\ \llparenthesis P \rrparenthesis$ consists of the application of x (of function type) to the argument $\llparenthesis P \rrparenthesis$, thus ensuring that the term resulting from the substitution is of the appropriate type. We note that, for instance, the encoding of rule $\otimes$L does not need to appeal to substitution – the λ-calculus let style rules can be mapped directly. Similarly, rule $\forall$R is mapped to type abstraction, whereas rule $\forall$L which types a process of the form $x\langle B\rangle.P$ maps to a substitution of the type application $x[B]$ for x in $\llparenthesis P \rrparenthesis$. The encoding of existentials is simpler due to the let-style elimination. We also highlight the encoding of the cut rule which embodies parallel composition of two processes sharing a linear name, which clarifies the use/offer duality of the intuitionistic calculus – the process that offers P is encoded and substituted into the encoded user Q.

Theorem 3.7. *If* $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ *then* $\llparenthesis \Omega \rrparenthesis; \llparenthesis \Gamma \rrparenthesis; \llparenthesis \Delta \rrparenthesis \vdash \llparenthesis P \rrparenthesis : A$.

Proof. Straightforward induction. The proof follows from the typing derivations of Figures 5 and 6. ☐

Example 3.8 (Encoding of Polyπ). Consider the following processes

$$P \triangleq z(X).z(Y).z(x).z(y).\overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) \quad Q \triangleq z\langle \mathbf{1}\rangle.z\langle \mathbf{1}\rangle.\overline{z}\langle x\rangle.\overline{z}\langle y\rangle.z(w).[w \leftrightarrow r]$$

with $\vdash P :: z{:}\forall X.\forall Y.X \multimap Y \multimap X \otimes Y$ and $z{:}\forall X.\forall Y.X \multimap Y \multimap X \otimes Y \vdash Q :: r{:}\mathbf{1}$, derivable as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X,Y;\cdot;x{:}X \vdash [x \leftrightarrow w] :: w{:}X} \quad \dfrac{}{X,Y;\cdot;y{:}Y \vdash [y \leftrightarrow z] :: z{:}Y}}{X,Y;\cdot;x{:}X,y{:}Y \vdash \overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z{:}X \otimes Y}}{X,Y;\cdot;x{:}X \vdash z(y).\overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z{:}Y \multimap X \otimes Y}}{X,Y;\cdot;\cdot \vdash z(x).z(y).\overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z{:}X \multimap Y \multimap X \otimes Y}}{X;\cdot;\cdot \vdash z(Y).z(x).z(y).\overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z{:}\forall Y.X \multimap Y \multimap X \otimes Y}}{\cdot;\cdot;\cdot \vdash z(X).z(Y).z(x).z(y).\overline{z}\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z{:}\forall X.\forall Y.X \multimap Y \multimap X \otimes Y}$$

The derivation (read bottom-up) consists of two applications of rule $\forall$R, two instances of rule $\multimap$R and one instance of rule $\otimes$R followed by two uses of the identity rule.

$$\left\llparenthesis \frac{}{\Omega;\Gamma;\cdot \vdash \mathbf{0} :: z{:}\mathbf{1}}(\mathbf{1}\mathsf{R}) \right\rrparenthesis \triangleq \frac{}{\Omega;\Gamma;\cdot \vdash \langle\rangle : \mathbf{1}}(\mathbf{1}I)$$

$$\left\llparenthesis \frac{\Omega;\Gamma;\Delta \vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}\mathbf{1} \vdash P :: z{:}C}(\mathbf{1}\mathsf{L}) \right\rrparenthesis \triangleq \frac{\Omega;\Gamma;x{:}\mathbf{1} \vdash x : \mathbf{1} \quad \Omega;\Gamma;\Delta \vdash \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta\vdash z:C} : C}{\Omega;\Gamma;\Delta, x{:}\mathbf{1} \vdash \mathsf{let}\ \mathbf{1} = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta\vdash z:C} : C}(\mathbf{1}E)$$

$$\left\llparenthesis \frac{}{\Omega;\Gamma;x{:}A \vdash [x \leftrightarrow z] :: z{:}A}(\textsc{id}) \right\rrparenthesis \triangleq \frac{}{\Omega;\Gamma;x{:}A \vdash x{:}A}(\textsc{var})$$

$$\left\llparenthesis \frac{\Omega;\Gamma;\cdot \vdash P :: x{:}A}{\Omega;\Gamma;\cdot \vdash !z(x).P :: z{:}!A}(!\mathsf{R}) \right\rrparenthesis \triangleq \frac{\Omega;\Gamma;\cdot \vdash \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\cdot\vdash x:A} : A}{\Omega;\Gamma;\cdot \vdash !\llparenthesis P \rrparenthesis_{\Omega;\Gamma;\cdot\vdash z:!A} : !A}(!I)$$

$$\left\llparenthesis (\multimap\mathsf{R})\ \frac{\Omega;\Gamma;\Delta, x{:}A \vdash P :: z{:}B}{\Omega;\Gamma;\Delta \vdash z(x).P :: z{:}A \multimap B} \right\rrparenthesis \triangleq (\multimap I)\ \frac{\Omega;\Gamma;\Delta, x{:}A \vdash \llparenthesis P \rrparenthesis_{\Omega,\Gamma;\Delta,x:A\vdash z:B} : B}{\Omega;\Gamma;\Delta \vdash \lambda x{:}A.\llparenthesis P \rrparenthesis_{\Omega,\Gamma;\Delta,x:A\vdash z:B} : A \multimap B}$$

$$\left\llparenthesis \frac{\Omega;\Gamma;\Delta_1 \vdash P :: y{:}A \quad \Omega;\Gamma;\Delta_2, x{:}B \vdash Q :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2, x{:}A \multimap B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: z{:}C}(\multimap\mathsf{L}) \right\rrparenthesis \triangleq$$

$$\frac{\Omega;\Gamma;\Delta_2, x{:}B \vdash \llparenthesis Q \rrparenthesis_{\Omega;\Gamma;\Delta_2,x:B\vdash z:C} : C \qquad \dfrac{\Omega;\Gamma;x{:}A \multimap B \vdash x{:}A \multimap B \quad \Omega;\Gamma;\Delta_1 \vdash \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash y:A} : A}{\Omega;\Gamma;\Delta_1, x{:}A \multimap B \vdash x\ \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash y:A} : B}(\multimap E)}{\Omega;\Gamma;\Delta_1,\Delta_2, x{:}A \multimap B \vdash \llparenthesis Q \rrparenthesis_{\Omega;\Gamma;\Delta_2,x:B\vdash z:C}\{(x\ \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash y:A})/x\} : C}(\textsc{subst})$$

$$\left\llparenthesis \frac{\Omega;\Gamma;\Delta_1 \vdash P :: x{:}A \quad \Omega;\Gamma;\Delta_2 \vdash Q :: z{:}B}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)z\langle x\rangle.(P \mid Q) :: z{:}A \otimes B}(\otimes\mathsf{R}) \right\rrparenthesis \triangleq \frac{\Omega;\Gamma;\Delta_1 \vdash \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash x:A} : A \quad \Omega;\Gamma;\Delta_2 \vdash \llparenthesis Q \rrparenthesis_{\Omega;\Gamma;\Delta_2\vdash z:B} : B}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash \langle \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash x:A} \otimes \llparenthesis Q \rrparenthesis_{\Omega;\Gamma;\Delta_2\vdash z:B} \rangle : A \otimes B}(\otimes I)$$

$$\left\llparenthesis \frac{\Omega;\Gamma;\Delta, y{:}A.x{:}B \vdash P :: z{:}C}{\Omega;\Gamma;\Delta, x{:}A \otimes B \vdash x(y).P :: z{:}C}(\otimes\mathsf{L}) \right\rrparenthesis \triangleq \frac{\Omega;\Gamma;x{:}A \otimes B \vdash x : A \otimes B \quad \Omega;\Gamma;\Delta, y{:}A, x{:}B \vdash \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta,y:A.x:B\vdash z:C} : C}{\Omega;\Gamma;\Delta, x{:}A \otimes B \vdash \mathsf{let}\ x \otimes y = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis_{\Omega;\Gamma;\Delta,y:A.x:B\vdash z:C} : C}(\otimes E)$$

Fig. 5. Translation on Typing Derivations from Polyπ to Linear-F (Part 1)

$$\left\llparenthesis (!\mathsf{L})\ \frac{\Omega;\Gamma,u{:}A;\Delta\vdash P :: z{:}C}{\Omega;\Gamma;\Delta,x{:}!A\vdash P\{x/u\} :: z{:}C}\right\rrparenthesis \triangleq (!E)\ \frac{\Omega;\Gamma;x{:}!A\vdash x : !A \quad \Omega;\Gamma,u{:}A;\Delta\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma,u:A;\Delta\vdash z:C} : C}{\Omega;\Gamma;\Delta,x{:}!A\vdash \mathsf{let}\ !u = x\ \mathsf{in}\ \llparenthesis P\rrparenthesis_{\Omega;\Gamma,u:A;\Delta\vdash z:C} : C}$$

$$\left\llparenthesis (\mathsf{copy})\ \frac{\Omega;\Gamma,u{:}A;\Delta,x{:}A\vdash P :: z{:}C}{\Omega;\Gamma,u{:}A;\Delta\vdash (\nu x)u\langle x\rangle.P :: z{:}C}\right\rrparenthesis \triangleq$$

$$(\textsc{subst})\ \frac{\Omega;\Gamma,u{:}A;\Delta,x{:}A\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma,u:A;\Delta,x:A\vdash z:C} : C \quad \Omega;\Gamma,u{:}A;\cdot\vdash u{:}A}{\Omega;\Gamma,u{:}A;\Delta\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma,u:A;\Delta,x:A\vdash z:C}\{u/x\} : C}$$

$$\left\llparenthesis (\forall\mathsf{R})\ \frac{\Omega,X;\Gamma;\Delta\vdash P :: z{:}A}{\Omega;\Gamma;\Delta\vdash z(X).P :: z{:}\forall X.A}\right\rrparenthesis \triangleq (\forall I)\ \frac{\Omega,X;\Gamma;\Delta\vdash \llparenthesis P\rrparenthesis_{\Omega,X;\Gamma;\Delta\vdash z:A} : A}{\Omega;\Gamma;\Delta\vdash \Lambda X.\llparenthesis P\rrparenthesis_{\Omega,X;\Gamma;\Delta\vdash z:A} : \forall X.A}$$

$$\left\llparenthesis (\forall\mathsf{L})\ \frac{\Omega\vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta,x{:}A\{B/X\}\vdash P :: z{:}C}{\Omega;\Gamma;\Delta,x{:}\forall X.A\vdash x\langle B\rangle.P :: z{:}C}\right\rrparenthesis \triangleq$$

$$(\textsc{subst})\ \frac{\Omega;\Gamma;\Delta,x{:}A\{B/X\}\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta,x:A\{B/X\}\vdash z:C} : C \quad (\forall E)\ \dfrac{\Omega;\Gamma,x{:}\forall X.A\vdash x{:}\forall X.A \quad \Omega\vdash B\,\mathsf{type}}{\Omega;\Gamma;x{:}\forall X.A\vdash x[B] : A\{B/X\}}}{\Omega;\Gamma;\Delta,x{:}\forall X.A\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta,x:A\{B/X\}\vdash z:C}\{(x[B]/x)\} : C}$$

$$\left\llparenthesis (\exists\mathsf{R})\ \frac{\Omega\vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta\vdash P :: z{:}A\{B/X\}}{\Omega;\Gamma;\Delta\vdash z\langle B\rangle.P :: z{:}\exists X.A}\right\rrparenthesis \triangleq (\exists I)\ \frac{\Omega\vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta\vdash z:A\{B/X\}} : A\{B/X\}}{\Omega;\Gamma;\Delta\vdash \mathsf{pack}\ B\ \mathsf{with}\ \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta\vdash z:A\{B/X\}} : \exists X.A}$$

$$\left\llparenthesis (\exists\mathsf{L})\ \frac{\Omega,Y;\Gamma;\Delta,x{:}A\vdash P :: z{:}C}{\Omega;\Gamma;\Delta,x{:}\exists X.A\vdash x(Y).P :: z{:}C}\right\rrparenthesis \triangleq$$

$$(\exists E)\ \frac{\Omega;\Gamma;x{:}\exists Y.A\vdash x{:}\exists Y.A \quad \Omega,Y;\Gamma;\Delta,x{:}A\vdash \llparenthesis P\rrparenthesis_{\Omega,Y;\Gamma;\Delta,x:A\vdash z:C} : C}{\Omega;\Gamma;\Delta,x{:}\exists Y.A\vdash \mathsf{let}\ (Y,x) = x\ \mathsf{in}\ \llparenthesis P\rrparenthesis_{\Omega,Y;\Gamma;\Delta,x:A\vdash z:C} : C}$$

$$\left\llparenthesis (\mathsf{cut})\ \frac{\Omega;\Gamma;\Delta_1\vdash P :: x{:}A \quad \Omega;\Gamma;\Delta_2,x{:}A\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2\vdash (\nu x)(P\mid Q) :: z{:}C}\right\rrparenthesis \triangleq$$

$$(\textsc{subst})\ \frac{\Omega;\Gamma;\Delta_2,x{:}A\vdash \llparenthesis Q\rrparenthesis_{\Omega;\Gamma;\Delta_2,x:A\vdash z:C} : C \quad \Omega;\Gamma;\Delta_1\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash x:A} : A}{\Omega;\Gamma;\Delta_1,\Delta_2\vdash \llparenthesis Q\rrparenthesis_{\Omega;\Gamma;\Delta_2,x:A\vdash z:C}\{\llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash x:A}/x\} : C}$$

$$\left\llparenthesis (\mathsf{cut}^!)\ \frac{\Omega;\Gamma;\cdot\vdash P :: x{:}A \quad \Omega;\Gamma,u{:}A;\Delta\vdash Q :: z{:}C}{\Omega;\Gamma;\Delta\vdash (\nu u)(!u(x).P\mid Q) :: z{:}C}\right\rrparenthesis \triangleq (\textsc{subst}^!)\ \frac{\Omega;\Gamma,u{:}A;\Delta\vdash \llparenthesis Q\rrparenthesis_{\Omega;\Gamma,u:A;\Delta\vdash z:C} : C \quad \Omega;\Gamma;\cdot\vdash \llparenthesis P\rrparenthesis_{\Omega;\Gamma;\Delta_1\vdash x:A} : A}{\Omega;\Gamma;\Delta\vdash \llparenthesis Q\rrparenthesis_{\Omega;\Gamma,u:A;\Delta\vdash z:C}\{\llparenthesis P\rrparenthesis_{\Omega;\Gamma;\cdot\vdash x:A}/u\}}$$

Fig. 6. Translation on Typing Derivations from Polyπ to Linear-F (Part 2)

$$\cfrac{\cdot \vdash \mathbf{1}\ \mathsf{type} \qquad \cfrac{\cdot \vdash \mathbf{1}\ \mathsf{type} \qquad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\cdot;\cdot;w{:}\mathbf{1} \vdash [w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;w{:}\mathbf{1}, z{:}\mathbf{1} \vdash [w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;z{:}\mathbf{1}\otimes\mathbf{1} \vdash z(w).[w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;z{:}\mathbf{1} \multimap \mathbf{1}\otimes\mathbf{1} \vdash \overline{z}\langle y\rangle.z(w).[w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;z{:}\mathbf{1} \multimap \mathbf{1} \multimap \mathbf{1}\otimes\mathbf{1} \vdash \overline{z}\langle x\rangle.\overline{z}\langle y\rangle.z(w).[w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;z{:}\forall Y.\mathbf{1} \multimap Y \multimap \mathbf{1}\otimes Y \vdash z\langle\mathbf{1}\rangle.\overline{z}\langle x\rangle.\overline{z}\langle y\rangle.z(w).[w \leftrightarrow r] :: r{:}\mathbf{1}}}{\cdot;\cdot;z{:}\forall X.\forall Y.X \multimap Y \multimap X\otimes Y \vdash z\langle\mathbf{1}\rangle.z\langle\mathbf{1}\rangle.\overline{z}\langle x\rangle.\overline{z}\langle y\rangle.z(w).[w \leftrightarrow r] :: r{:}\mathbf{1}}$$

The typing derivation for Q above is dual to that of P: two instances of $\forall$L, followed by two instances of $\multimap$L, followed by an instance of $\otimes$L, $\mathbf{1}$L and the identity rule.

Then: $(\!|P|\!) = \Lambda X.\Lambda Y.\lambda x{:}X.\lambda y{:}Y.\langle x \otimes y\rangle \qquad (\!|Q|\!) = \mathsf{let}\ x \otimes y = z[\mathbf{1}][\mathbf{1}]\ \langle\rangle\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathbf{1} = y\ \mathsf{in}\ x$

$(\!|(\nu z)(P \mid Q)|\!) = \mathsf{let}\ x \otimes y = (\Lambda X.\Lambda Y.\lambda x{:}X.\lambda y{:}Y.\langle x \otimes y\rangle)[\mathbf{1}][\mathbf{1}]\ \langle\rangle\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathbf{1} = y\ \mathsf{in}\ x$

By the behaviour of $(\nu z)(P \mid Q)$, which consists of a sequence of cuts, and its encoding, we have that $(\!|(\nu z)(P \mid Q)|\!) \to^+ \langle\rangle$ and $(\nu z)(P \mid Q) \to^+ \mathbf{0} = (\!|\langle\rangle|\!)$.

The reader may at this point be wondering what reasonable properties can a translation from (typed) π-calculus processes to polymorphic λ-terms have, given that the π-calculus exhibits non-determinism that is absent from the λ-calculus. However, as is made clear by our developments in Section 3.3, our type-preserving translation from Polyπ to Linear-F is only possible precisely because the session discipline effectively erases all forms of non-determinism (in the sense of non-confluent computations) from the π-calculus. While the operational semantics of Polyπ processes does contain forms of non-determinism (sometimes dubbed *don't care* non-determinism, as opposed to *don't know* non-determinism), the session typing ensures nonetheless confluence and strong normalisation [51], as is the case with parallel reduction in typed λ-calculus.

Note that typing of Polyπ is implicitly modulo structural equivalence, as in previous work [12, 13].

In general, the translation of Def. 3.6 can introduce some distance between the *immediate* operational behaviour of a process and its corresponding λ-term, insofar as the translations of cuts (and left rules to non let-form elimination rules) make use of substitutions that can take place deep within the resulting term. Consider the process at the root of the following typing judgment $\Delta_1, \Delta_2, \Delta_3 \vdash (\nu x)(x(y).P_1 \mid (\nu y)x\langle y\rangle.(P_2 \mid w(z).\mathbf{0})) :: w{:}\mathbf{1} \multimap \mathbf{1}$, derivable through a cut on session x between instances of $\multimap$R and $\multimap$L, where the continuation process $w(z).\mathbf{0}$ offers a session $w{:}\mathbf{1} \multimap \mathbf{1}$ (and so must use rule $\mathbf{1}$L on x). We have that: $(\nu x)(x(y).P_1 \mid (\nu y)x\langle y\rangle.(P_2 \mid w(z).\mathbf{0})) \to (\nu x, y)(P_1 \mid P_2 \mid w(z).\mathbf{0})$. However, the translation of the process above results in the term $\lambda z{:}\mathbf{1}.\mathsf{let}\ \mathbf{1} = ((\lambda y{:}A.(\!|P_1|\!))\ (\!|P_2|\!))\ \mathsf{in}\ \mathsf{let}\ \mathbf{1} = z\ \mathsf{in}\ \langle\rangle$, where the redex that corresponds to the process reduction is present but hidden under the binder for z (corresponding to the input along w).

In this sense, the encoding of parallel composition through a (meta-level) substitution can indeed hide some of the computational behaviour of a process under a binder in the corresponding λ-term, (albeit the encoding $(\!|(\nu x, y)(P_1 \mid P_2 \mid w(z).\mathbf{0})|\!)$ is β-equivalent to the λ-term above). This is justified proof theoretically by the commuting conversions of sequent calculus and therefore by contextual equivalence. An alternative would be to consider a let-binder in the λ-calculus that would act as the translation target of all substitution-style rules (the cuts, copy, $\multimap$L and $\forall$L rules). In this alternate formulation, the process above would be translated as $\mathsf{let}\ x = \lambda y{:}A.(\!|P_1|\!)\ \mathsf{in}\ \mathsf{let}\ x' = x\ (\!|P_2|\!)\ \mathsf{in}\ \mathsf{let}\ \mathbf{1} =$

x' in $\lambda z{:}\mathbf{1}.\mathsf{let}\ \mathbf{1} = z\ \mathsf{in}\ \langle\rangle$, which mirrors the process reduction order more explicitly, at the cost of an extra-logical construct in the λ-calculus.

Thus, to establish a more precise form of operational completeness, without adding extra-logical constructs to the λ-calculus, we consider full β-reduction, denoted by $\rightarrow_\beta$, i.e. enabling β-reductions under binders (such an extension is easily obtained by including evaluation context clauses under all binding sites in the language). We note that, as argued above, operational correspondence does not *require* full β-reduction, but the results can be established more naturally and precisely (i.e., without an appeal to contextual equivalence and/or by adding extra-logical features to the λ-calculus).

Theorem 3.9 (Operational Completeness). *Let $\Omega; \Gamma; \Delta \vdash P :: z{:}A$. If $P \rightarrow Q$ then $(\!|P|\!) \rightarrow_\beta^* (\!|Q|\!)$.*

In order to study the soundness direction it is instructive to consider typed process $x{:}\mathbf{1} \multimap \mathbf{1} \vdash \overline{x}\langle y\rangle.(\nu z)(z(w).\mathbf{0} \mid \overline{z}\langle w\rangle.\mathbf{0}) :: v{:}\mathbf{1}$ and its translation:

$$\begin{aligned}(\!|\overline{x}\langle y\rangle.(\nu z)(z(w).\mathbf{0} \mid \overline{z}\langle w\rangle.\mathbf{0})|\!) &= (\!|(\nu z)(z(w).\mathbf{0} \mid \overline{z}\langle w\rangle.\mathbf{0})|\!)\{(x\ \langle\rangle)/x\} \\ &= \mathsf{let}\ \mathbf{1} = (\lambda w{:}\mathbf{1}.\mathsf{let}\ \mathbf{1} = w\ \mathsf{in}\ \langle\rangle)\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathbf{1} = x\ \langle\rangle\ \mathsf{in}\ \langle\rangle\end{aligned}$$

The process above cannot reduce due to the output prefix on x, which cannot synchronise with a corresponding input action since there is no provider for x (i.e. the channel is in the left-hand side context). However, its encoding can exhibit the β-redex corresponding to the synchronisation along z, hidden by the prefix on x. The corresponding reductions hidden under prefixes in the encoding can be *soundly* exposed in the session calculus by appealing to the commuting conversions of linear logic (e.g. in the process above, the instance of rule $\multimap\mathsf{L}$ corresponding to the output on x can be commuted with the cut on z).

As shown in [50], commuting conversions are sound wrt observational equivalence, and thus we formulate operational soundness through a notion of *extended* process reduction, which extends process reduction with the reductions that are induced by commuting conversions. Such a relation was also used for similar purposes in [8] and in [37], in a classical linear logic setting. For conciseness, we define extended reduction as a relation on *typed* processes modulo $\equiv$.

Definition 3.10 (Extended Reduction [8]). We define $\mapsto$ as the type preserving relations on typed processes modulo $\equiv$ generated by:

(1) $C[(\nu y)x\langle y\rangle.P] \mid x(y).Q \mapsto C[(\nu y)(P \mid Q)]$;
(2) $C[(\nu y)x\langle y\rangle.P] \mid !x(y).Q \mapsto C[(\nu y)(P \mid Q)] \mid !x(y).Q$; and (3) $(\nu x)(!x(y).Q) \mapsto \mathbf{0}$

where C is a (typed) process context which does not capture the bound name y.

We highlight that clause (3) above is exactly the reduction of a cut between promotion and weakening in linear logic.

Theorem 3.11 (Operational Soundness). *Let $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ and $(\!|P|\!) \rightarrow M$, there exists Q such that $P \mapsto^* Q$ and $(\!|Q|\!) =_\alpha M$.*

Before addressing the more semantic properties that are detailed in the following sections, it is important to consider the general landscape of our encodings: Both Polyπ and Linear-F are extremely proof-theoretically well-behaved, satisfying confluence and strong normalization. In this sense, our encodings are greatly simplified and inherit significant intrinsic correctness from typing alone, seeing as the main differences between the two calculi lie in those between natural deduction and sequent calculi style systems themselves. This is made manifest in our encodings by the accounting of commutting conversions via behavioural equivalence or full β-reduction (alternatively, as discussed above, by considering an extension of the λ-calculus with a general let-binder).

Any extensions of either system that would weaken their proof-theoretic robustness, e.g. divergence or other forms of effects, would require careful revision of the encodings and their operational properties. In terms of divergence, a revision of the encoding along the lines detailed above with a let-binder (and the appropriate recursive constructs) would likely suffice. To consider more general effects, a framework along the lines of the work [47] would need to be considered, likely foregoing the logical correspondence. In such a setting, operational correctness can be reestablished although the status of the semantic properties of Section 3.3 (and subsequent sections) is unclear.

3.3 Inversion and Full Abstraction

Having established the operational preciseness of the encodings to-and-from Polyπ and Linear-F, we establish our main results for the encodings. Specifically, we show that the encodings are mutually inverse up-to behavioural equivalence (with *fullness* as its corollary), which then enables us to establish *full abstraction* for *both* encodings.

Theorem 3.12 (Inverse).

- *If* $\Omega; \Gamma; \Delta \vdash M : A$ *then* $\Omega; \Gamma; \Delta \vdash (\!|[\![M]\!]_z|\!) \cong M : A$
- *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then* $\Omega; \Gamma; \Delta \vdash [\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} P :: z{:}A$

Corollary 3.13 (Fullness).

- *Given* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$*, there exists* M *such that* $\Omega; \Gamma; \Delta \vdash M : A$ *and* $\Omega; \Gamma; \Delta \vdash [\![M]\!]_z \approx_{\mathsf{L}} P :: z{:}A$.
- *Given* $\Omega; \Gamma; \Delta \vdash M : A$*, there exists* P *such that* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *and* $\Omega; \Gamma; \Delta \vdash (\!|P|\!) \cong M : A$.

We now state our full abstraction results. Given two Linear-F terms of the same type, equivalence in the image of the $[\![-]\!]_z$ translation can be used as a proof technique for contextual equivalence in Linear-F. This is called the *soundness* direction of full abstraction in the literature [26] and proved by showing the relation generated by $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$ forms $\cong$; we then establish the *completeness* direction by contradiction, using fullness (see Appendix A.2).

Lemma 3.14. *Let* $\cdot \vdash M : \mathbf{2}$. $M \Downarrow \mathsf{T}$ *iff* $[\![M]\!]_z \approx_{\mathsf{L}} [\![\mathsf{T}]\!]_z :: z{:}[\![\mathbf{2}]\!]$

Proof. By operational correspondence. ∎

Theorem 3.15 (Full Abstraction). $\Omega; \Gamma; \Delta \vdash M \cong N : A$ *iff* $\Omega; \Gamma; \Delta \vdash [\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z :: z{:}A$.

Proof. (**Soundness**, $\Leftarrow$) Since $\cong$ is the largest consistent congruence compatible with the booleans, let $M \mathcal{R} N$ iff $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$. We show that $\mathcal{R}$ is one such relation.

(1) (Congruence) Since $\approx_{\mathsf{L}}$ is a congruence, $\mathcal{R}$ is a congruence.
(2) (Reduction-closed) Let $M \longrightarrow M'$ and $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$. Then we have by operational correspondence (Theorem 3.5) that $[\![M]\!]_z \longrightarrow^* P$ such that $P \approx_{\mathsf{L}} [\![M']\!]_z$ hence $[\![M']\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$, thus $\mathcal{R}$ is reduction closed.
(3) (Compatible with the booleans) Follows from Lemma 3.14.

(**Completeness**, $\Rightarrow$) Assume to the contrary that $M \cong N : A$ and $[\![M]\!]_z \not\approx_{\mathsf{L}} [\![N]\!]_z :: z{:}A$.

This means we can find a distinguishing context R such that $(\nu z, \tilde{x})([\![M]\!]_z \mid R) \approx_{\mathsf{L}} [\![\mathsf{T}]\!]_y :: y{:}[\![\mathbf{2}]\!]$ and $(\nu z, \tilde{x})([\![N]\!]_z \mid R) \approx_{\mathsf{L}} [\![\mathsf{F}]\!]_y :: y{:}[\![\mathbf{2}]\!]$. By Fullness (Theorem 3.13), we have that there exists some L such that $[\![L]\!]_y \approx_{\mathsf{L}} R$, thus: $(\nu z, \tilde{x})([\![M]\!]_z \mid [\![L]\!]_y) \approx_{\mathsf{L}} [\![\mathsf{T}]\!]_y :: y{:}[\![\mathbf{2}]\!]$ and $(\nu z, \tilde{x})([\![N]\!]_z \mid [\![L]\!]_y) \approx_{\mathsf{L}} [\![\mathsf{F}]\!]_y :: y{:}[\![\mathbf{2}]\!]$. By Theorem 3.15 (Soundness), we have that $L[M] \cong \mathsf{T}$ and $L[N] \cong \mathsf{F}$ and thus $L[M] \not\cong L[N]$ which contradicts $M \cong N : A$. ∎

We can straightforwardly combine the above full abstraction with Theorem 3.12 to obtain full abstraction of the $(\!|-|\!)$ translation.

Theorem 3.16 (Full Abstraction). $\Omega; \Gamma; \Delta \vdash P \approx_{\mathsf{L}} Q :: z{:}A$ *iff* $\Omega; \Gamma; \Delta \vdash (\!|P|\!) \cong (\!|Q|\!) : A$.

Proof. (**Soundness**, $\Leftarrow$) Let $M = (\!|P|\!)$ and $N = (\!|Q|\!)$. By Theorem 3.15 (Completeness) we have $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$. Thus by Theorem 3.12 we have: $[\![M]\!]_z = [\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} P$ and $[\![N]\!]_z = [\![(\!|Q|\!)]\!]_z \approx_{\mathsf{L}} Q$. By compatibility with observational equivalence we have $P \approx_{\mathsf{L}} Q :: z{:}A$.
(**Completeness**, $\Rightarrow$) From $P \approx_{\mathsf{L}} Q :: z{:}A$, Theorem 3.12 and compatibility with observational equivalence we have $[\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} [\![(\!|Q|\!)]\!]_z :: z{:}A$. Let $(\!|P|\!) = M$ and $(\!|Q|\!) = N$. We have by Theorem 3.15 (Soundness) that $M \cong N : A$ and thus $(\!|P|\!) \approx_{\mathsf{L}} (\!|Q|\!) : A$. □

4 INDUCTIVE AND COINDUCTIVE SESSION TYPES

In this section we study inductive and coinductive sessions, arising through encodings of initial F-algebras and final F-coalgebras in the polymorphic λ-calculus.

The study of polymorphism in the λ-calculus [2, 10, 27, 58] has shown that parametric polymorphism is expressive enough to encode both inductive and coinductive types in a precise way, through a faithful representation of initial and final (co)algebras [40], without extending the language of terms nor the semantics of the calculus, giving a logical justification to the Church encodings of inductive datatypes such as lists and natural numbers.

The polymorphic session typing framework of the previous sections allows us to express fairly intricate communication behaviours, being able to specify generic protocols through both existential and universal polymorphism (i.e. protocols that are parametric in their sub-protocols). However, it is often the case that protocols are expressed in terms of recursive behaviours (e.g., a client iterates over a buy list with a server, a server that repeats a sequence of interactions with a client an arbitrary number of times until the client chooses to terminate, etc) which are seemingly unavailable in the framework of Section 2. The introduction of recursive behaviours in the logical-based session typing framework has been addressed through the introduction of explicit inductive and coinductive session types [37, 72] and the corresponding process constructs, preserving the good properties of the framework such as strong normalisation and absence of deadlocks.

However, the study of polymorphism in the λ-calculus [2, 10, 27, 58] has shown that parametric polymorphism is expressive enough to encode both inductive and coinductive types in a precise way, through a faithful representation of initial and final (co)algebras [40], without extending the language of terms nor the semantics of the calculus.

Given the logical foundation of the polymorphic session calculus it is natural to wonder if such a result holds for inductive and coinductive sessions. In this section we answer this question *positively* by using our fully abstract encodings of (linear) polymorphic λ-calculus to show that session polymorphism is expressive enough to encode inductive and coinductive sessions, "importing" the results for the λ-calculus through the encodings. The development of this section is a particular instance of the benefits of our encodings which enable us to import non-trivial results from the λ-calculus to our process setting for free. We first provide a brief recap of the representation of inductive and coinductive types using polymorphism in System F.

Inductive and Coinductive Types in System F. Exploring an algebraic interpretation of polymorphism where types are interpreted as functors, it can be shown that given a type F with a free variable X that occurs only positively (i.e., occurrences of X are on the left-hand side of an even number of function arrows), the polymorphic type $\forall X.((F(X) \to X) \to X)$ forms an initial F-algebra [2, 60] (we write $F(X)$ to denote that X may occur in F). This enables the representation of *inductively* defined structures using an algebraic or categorical justification. For instance, the natural numbers can be seen as the initial F-algebra of $F(X) = \mathbf{1} + X$ (where $\mathbf{1}$ is the unit type and + is the coproduct), and are thus *already present* in System F, in a precise sense, as the type $\forall X.((\mathbf{1} + X) \to X) \to X$ (noting that both $\mathbf{1}$ and + can also be encoded in System F). A similar

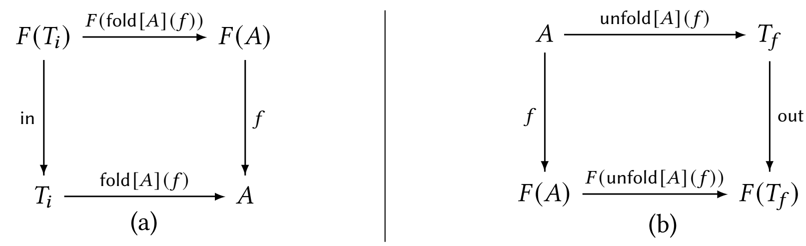

Fig. 7. Diagrams for Initial F-algebras and Final F-coalgebras

story can be told for *coinductively* defined structures, which correspond to final F-coalgebras and are representable with the polymorphic type $\exists X.(X \to F(X)) \times X$, where $\times$ is a product type. In the remainder of this section we assume the positivity requirement on F mentioned above.

While the complete formal development of the representation of inductive and coinductive types in System F would lead us too far astray, we summarise here the key concepts as they apply to the λ-calculus (the interested reader can refer to [27] for the full categorical details).

To show that the polymorphic type $T_i \triangleq \forall X.((F(X) \to X) \to X)$ is an initial F-algebra, one exhibits a pair of λ-terms, often dubbed fold and in, such that the diagram in Fig. 7(a) commutes (for any A, where $F(f)$, where f is a λ-term, denotes the functorial action of F applied to f), and, crucially, that fold is *unique*. When these conditions hold, we are justified in saying that T_i is a least fixed point of F. Through a fairly simple calculation, we have that:

$$\begin{array}{rcl} \mathsf{fold} & \triangleq & \Lambda X.\lambda f{:}F(X) \to X.\lambda t{:}T_i.t[X](f) \\ \mathsf{in} & \triangleq & \lambda x{:}F(T_i).\Lambda X.\lambda f{:}F(X) \to X.f\,(F(\mathsf{fold}[X](x))(x)) \end{array}$$

satisfy the necessary equalities. To show uniqueness one appeals to *parametricity*, which allows us to prove that any function of the appropriate type is equivalent to fold. This property is often dubbed initiality or universality.

The construction of final F-coalgebras and their justification as *greatest* fixed points is dual. Assuming products in the calculus and taking $T_f \triangleq \exists X.(X \to F(X)) \times X$, we produce the λ-terms

$$\begin{array}{rcl} \mathsf{unfold} & \triangleq & \Lambda X.\lambda f{:}X \to F(X).\lambda x{:}T_f.\mathsf{pack}\,X\,\mathsf{with}\,(f,x) \\ \mathsf{out} & \triangleq & \lambda t : T_f.\mathsf{let}\,(X,(f,x)) = t\,\mathsf{in}\,F(\mathsf{unfold}[X](f))\,(f(x)) \end{array}$$

such that the diagram in Fig. 7(b) commutes and unfold is unique (again, up to parametricity). While the argument above applies to System F, a similar development can be made in Linear-F [10] by considering $T_i \triangleq \forall X.!(F(X) \multimap X) \multimap X$ and $T_f \triangleq \exists X.!(X \multimap F(X)) \otimes X$. Reusing the same names for the sake of conciseness, the associated *linear* λ-terms are:

$$\begin{array}{rcl} \mathsf{fold} & \triangleq & \Lambda X.\lambda u{:}!(F(X) \multimap X).\lambda y{:}T_i.(y[X]\,u) : \forall X.!(F(X) \multimap X) \multimap T_i \multimap X \\ \mathsf{in} & \triangleq & \lambda x{:}F(T_i).\Lambda X.\lambda y{:}!(F(X) \multimap X).\mathsf{let}\,!u = y\,\mathsf{in}\,k\,(F\,(\mathsf{fold}[X](!u))(x)) : F(T_i) \multimap T_i \\ \mathsf{unfold} & \triangleq & \Lambda X.\lambda u{:}!(X \multimap F(X)).\lambda x{:}X.\mathsf{pack}\,X\,\mathsf{with}\,\langle u \otimes x\rangle : \forall X.!(X \multimap F(X)) \multimap X \multimap T_f \\ \mathsf{out} & \triangleq & \lambda t : T_f.\mathsf{let}\,(X,(u,x)) = t\,\mathsf{in}\,\mathsf{let}\,!f = u\,\mathsf{in}\,F(\mathsf{unfold}[X](!f))\,(f(x)) : T_f \multimap F(T_f) \end{array}$$

Inductive and Coinductive Sessions for Free. As a consequence of full abstraction we may appeal to the $[\![-]\!]_z$ encoding to derive representations of fold and unfold that satisfy the necessary algebraic properties. The derived processes are (recall that we write $\overline{x}\langle y\rangle.P$ for $(\nu y)x\langle y\rangle.P$):

$$\begin{array}{rcl} [\![\mathsf{fold}]\!]_z & \triangleq & z(X).z(u).z(y).(\nu w)((\nu x)([y \leftrightarrow x] \mid x\langle X\rangle.[x \leftrightarrow w]) \mid \overline{w}\langle v\rangle.([u \leftrightarrow v] \mid [w \leftrightarrow z])) \\ [\![\mathsf{unfold}]\!]_z & \triangleq & z(X).z(u).z(x).z\langle X\rangle.\overline{z}\langle y\rangle.([u \leftrightarrow y] \mid [x \leftrightarrow z]) \end{array}$$

We can then show universality of the two constructions. We write $P^u_{x,y}$ to single out that x and y and u are free in P and $P^v_{z,w}$ to denote the result of employing capture-avoiding substitution on P, substituting x, y, u by z, w, v, respectively. Let:

$$\begin{array}{rcl} \mathsf{foldP}(A)^u_{y_1,y_2} & \triangleq & (\nu x)(\llbracket \mathsf{fold} \rrbracket_x \mid x\langle A\rangle.\overline{x}\langle v\rangle.(\overline{u}\langle y\rangle.[y \leftrightarrow v] \mid \overline{x}\langle z\rangle.([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2]))) \\ \mathsf{unfoldP}(A)^u_{y_1,y_2} & \triangleq & (\nu x)(\llbracket \mathsf{unfold} \rrbracket_x \mid x\langle A\rangle.\overline{x}\langle v\rangle.(\overline{u}\langle y\rangle.[y \leftrightarrow v] \mid \overline{x}\langle z\rangle.([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2]))) \end{array}$$

where $\mathsf{foldP}(A)^u_{y_1,y_2}$ corresponds to the application of fold to an F-algebra A with the associated morphism $F(A) \multimap A$ available on the shared channel u, consuming an ambient session $y_1{:}T_i$ and offering $y_2{:}A$. Similarly, $\mathsf{unfoldP}(A)^u_{y_1,y_2}$ corresponds to the application of unfold to an F-coalgebra A with the associated morphism $A \multimap F(A)$ available on the shared channel u, consuming an ambient session $y_1{:}A$ and offering $y_2{:}T_f$.

Theorem 4.1 (Universality of foldP). *Let Q be a well-typed process such that*

$$X; u{:}F(X) \multimap X; y_1{:}T_i \vdash Q :: y_2{:}X$$

for some functor F and channels y_1, y_2. We have that:

$$X; u{:}F(X) \multimap X; y_1{:}T_i \vdash Q \approx_{\mathsf{L}} \mathsf{foldP}(X)^u_{y_1,y_2} :: y_2{:}X$$

Proof. By universality of fold we have that $\mathsf{fold}[X](u) \cong M$ where $u :!(F(X) \multimap X)$, for any M of the appropriate type. In particular we have that $\mathsf{fold}[X](u) \cong (\!|\mathsf{foldP}(X)_{y_1,y_2}|\!)$. By full abstraction (Theorem 3.15) and transitivity we have that $\llbracket \mathsf{fold}[X](u) \rrbracket_{y_2} \approx_{\mathsf{L}} \llbracket (\!|\mathsf{foldP}(X)^u_{y_1,y_2}|\!) \rrbracket_{y_2} \approx_{\mathsf{L}} \llbracket M \rrbracket_{y_2}$. By the inverse theorem (Theorem 3.12) it follows that $\mathsf{foldP}(X)^u_{y_1,y_2} \approx_{\mathsf{L}} \llbracket M \rrbracket_{y_2}$. Since the reasoning holds for any such M we can conclude by Fullness of the encoding (Corollary 3.13). □

Theorem 4.2 (Universality of unfoldP). *Let Q be a well-typed process A an F-coalgebra such that:*

$$\cdot; \cdot; y_1{:}A \vdash Q :: y_2{:}T_f$$

we have that

$$\cdot; u{:}A \multimap F(A); y_1{:}A \vdash Q \approx_{\mathsf{L}} \mathsf{unfoldP}(A)^u_{y_1,y_2} :: y_2 :: T_f$$

Proof. By universality of unfold we have that $\mathsf{unfold}[A](u) \cong M$ where $u{:}!(A \multimap F(A))$, for any M of the appropriate type. We thus have that $\mathsf{unfold}[A](u) \cong (\!|\mathsf{unfoldP}(A)^u_{y_1,y_2}|\!)$, since $(\!|\mathsf{unfoldP}(A)^u_{y_1,y_2}|\!)$ is one such M. By full abstraction (Theorem 3.15) and transitivity we have that $\llbracket \mathsf{unfold}[A](u) \rrbracket_{y_2} \approx_{\mathsf{L}} \llbracket (\!|\mathsf{unfoldP}(A)^u_{y_1,y_2}|\!) \rrbracket_{y_2} \approx_{\mathsf{L}} \llbracket M \rrbracket_{y_2}$. By the inverse theorem (Theorem 3.12) it then follows that $\mathsf{unfoldP}(A)^u_{y_1,y_2} \approx_{\mathsf{L}} \llbracket M \rrbracket_{y_2}$. Since the reasoning holds for any such M we can conclude by Fullness of the encoding (Corollary 3.13). □

Example 4.3 (Natural Numbers). We show how to represent the natural numbers as an inductive session type using $F(X) = \mathbf{1} \oplus X$, making use of in:

$$\mathsf{zero}_x \triangleq (\nu z)(z.\mathsf{inl}; \mathbf{0} \mid \llbracket \mathsf{in}(z) \rrbracket_x) \quad \mathsf{succ}_{y,x} \triangleq (\nu s)(s.\mathsf{inr}; [y \leftrightarrow s] \mid \llbracket \mathsf{in}(s) \rrbracket_x)$$

with $\mathsf{Nat} \triangleq \forall X.!((\mathbf{1} \oplus X) \multimap X) \multimap X$ where $\vdash \mathsf{zero}_x :: x{:}\mathsf{Nat}$ and $y{:}\mathsf{Nat} \vdash \mathsf{succ}_{y,x} :: x{:}\mathsf{Nat}$ encode the representation of 0 and successor, respectively. The natural 1 would thus be represented by $\mathsf{one}_x \triangleq (\nu y)(\mathsf{zero}_y \mid \mathsf{succ}_{y,x})$. The behaviour of type Nat can be seen as a that of a sequence of internal choices of arbitrary (but finite) length. We can then observe that the foldP process acts as a recursor. For instance consider:

$$\mathsf{stepDec}_d \triangleq d(n).n.\mathsf{case}(\mathsf{zero}_d, [n \leftrightarrow d]) \quad \mathsf{dec}_{x,z} \triangleq (\nu u)(!u(d).\mathsf{stepDec}_d \mid \mathsf{foldP}(\mathsf{Nat})^u_{x,z})$$

with $\mathsf{stepDec}_d :: d{:}(\mathbf{1} \oplus \mathsf{Nat}) \multimap \mathsf{Nat}$ and $x{:}\mathsf{Nat} \vdash \mathsf{dec}_{x,z} :: z{:}\mathsf{Nat}$, where dec decrements a given natural number session on channel x. We have that:

$$(\nu x)(\mathsf{one}_x \mid \mathsf{dec}_{x,z}) \equiv (\nu x, y, u)(\mathsf{zero}_y \mid \mathsf{succ}_{y,x}!u(d).\mathsf{stepDec}_d \mid \mathsf{foldP}(\mathsf{Nat})^u_{x,z}) \approx_{\mathsf{L}} \mathsf{zero}_z$$

We note that the resulting encoding is reminiscent of the encoding of lists of [43] (where zero is the empty list and succ the cons cell). The main differences in the encodings arise due to our primitive notions of labels and forwarding, as well as due to the generic nature of in and fold.

Example 4.4 (Streams). We build on Example 4.3 by representing *streams* of natural numbers as a coinductive session type. We encode infinite streams of naturals with $F(X) = \mathsf{Nat} \otimes X$. Thus: $\mathsf{NatStream} \triangleq \exists X.!(X \multimap (\mathsf{Nat} \otimes X)) \otimes X$. The behaviour of a session of type NatStream amounts to an infinite sequence of outputs of channels of type Nat. Such an encoding enables us to construct the stream of all naturals nats (and the stream of all non-zero naturals oneNats):

$$\begin{array}{lcl}
\mathsf{genHdNext}_z & \triangleq & z(n).\overline{z}\langle y\rangle.(\overline{n}\langle n'\rangle.[n' \leftrightarrow y] \mid !z(w).\overline{n}\langle n'\rangle.\mathsf{succ}_{n',w}) \\
\mathsf{nats}_y & \triangleq & (\nu x, u)(\mathsf{zero}_x \mid !u(z).\mathsf{genHdNext}_z \mid \mathsf{unfoldP}(!\mathsf{Nat})^u_{x,y}) \\
\mathsf{oneNats}_y & \triangleq & (\nu x, u)(\mathsf{one}_x \mid !u(z).\mathsf{genHdNext}_z \mid \mathsf{unfoldP}(!\mathsf{Nat})^u_{x,y})
\end{array}$$

with $\mathsf{genHdNext}_z :: z{:}!\mathsf{Nat} \multimap \mathsf{Nat}\otimes!\mathsf{Nat}$ and both nats_y and $\mathsf{oneNats} :: y{:}\mathsf{NatStream}$. $\mathsf{genHdNext}_z$ consists of a helper that generates the current head of a stream and the next element. As expected, the following process implements a session that "unrolls" the stream once, providing the head of the stream and then behaving as the rest of the stream (recall that $\mathsf{out} : T_f \multimap F(T_f)$).

$$(\nu x)(\mathsf{nats}_x \mid [\![\mathsf{out}(x)]\!]_y) :: y{:}\mathsf{Nat} \otimes \mathsf{NatStream}$$

We note a peculiarity of the interaction of linearity with the stream encoding: a process that begins to deconstruct a stream has no way of "bottoming out" and stopping. One cannot, for instance, extract the first element of a stream of naturals and stop unrolling the stream in a well-typed way. We can, however, easily encode a "terminating" stream of all natural numbers via $F(X) = (\mathsf{Nat}\otimes!X)$ by replacing the $\mathsf{genHdNext}_z$ with the generator given as:

$$\mathsf{genHdNextTer}_z \quad \triangleq \quad z(n).\overline{z}\langle y\rangle.(\overline{n}\langle n'\rangle.[n' \leftrightarrow y] \mid !z(w).!w(w').\overline{n}\langle n'\rangle.\mathsf{succ}_{n',w'})$$

It is then easy to see that a usage of $[\![\mathsf{out}(x)]\!]_y$ results in a session of type $\mathsf{Nat}\otimes!\mathsf{NatStream}$, enabling us to discard the stream as needed. One can replay this argument with the operator $F(X) = (!\mathsf{Nat} \otimes X)$ to enable discarding of stream elements. Assuming such modifications, we can then show:

$$(\nu y)((\nu x)(\mathsf{nats}_x \mid [\![\mathsf{out}(x)]\!]_y) \mid y(n).[y \leftrightarrow z]) \approx_{\mathsf{L}} \mathsf{oneNats}_z :: z{:}\mathsf{NatStream}$$

5 COMMUNICATING VALUES

We now study encodings for an extension of the core session calculus with term passing (i.e., sending and receiving typed λ-terms). The core calculus drops polymorphism from Polyπ.

Using the development of term passing (Section 5.1) as a stepping stone, we generalise the encodings to a *higher-order* session calculus (Section 5.2), where processes can send, receive and execute other processes. To obtain such a calculus process passing, you extend the term-passing fragment with a monadic embedding of processes [71]. Proof theoretically, this calculus is inspired by Benton's LNL [6]. We show full abstraction and mutual inversion theorems for the encodings from higher-order to first-order. As a consequence, we can straightforwardly derive a strong normalisation property for the higher-order process-passing calculus.

5.1 Session Processes with Term Passing – Sess$\pi\lambda$

We consider a session calculus extended with a data layer obtained from a λ-calculus (whose terms are ranged over by M, N and types by τ, σ). We dub this calculus Sess$\pi\lambda$.

$$
\begin{array}{lcl@{\qquad}lcl}
P, Q & ::= & \cdots \mid x\langle M\rangle.P \mid x(y).P & A, B & ::= & \cdots \mid \tau \wedge A \mid \tau \supset A \\
M, N & ::= & \lambda x{:}\tau.M \mid M\,N \mid x & \tau, \sigma & ::= & \cdots \mid \tau \to \sigma
\end{array}
$$

Without loss of generality, we consider the data layer to be simply-typed, with a call-by-name semantics, satisfying the usual type safety properties. The typing judgment for this calculus is $\Psi \vdash M : \tau$. We omit session polymorphism for the sake of conciseness, restricting processes to communication of data and (session) channels. The typing judgment for processes is thus modified to $\Psi; \Gamma; \Delta \vdash P :: z{:}A$, where Ψ is an intuitionistic context that accounts for variables in the data layer. The rules for the relevant process constructs are (all other rules simply propagate the Ψ context from conclusion to premises):

$$
\frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta \vdash P :: z{:}A}{\Psi; \Gamma; \Delta \vdash z\langle M\rangle.P :: z{:}\tau \wedge A}\ (\wedge\mathsf{R})
\qquad
\frac{\Psi, y{:}\tau; \Gamma; \Delta, x{:}A \vdash Q :: z{:}C}{\Psi; \Gamma; \Delta, x{:}\tau \wedge A \vdash x(y).Q :: z{:}C}\ (\wedge\mathsf{L})
$$

$$
\frac{\Psi, x{:}\tau; \Gamma; \Delta \vdash P :: z{:}A}{\Psi; \Gamma; \Delta \vdash z(x).P :: z{:}\tau \supset A}\ (\supset\mathsf{R})
\qquad
\frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, x{:}A \vdash Q :: z{:}C}{\Psi; \Gamma; \Delta, x{:}\tau \supset A \vdash x\langle M\rangle.Q :: z{:}C}\ (\supset\mathsf{L})
$$

With the reduction rule given by:[1] $x\langle M\rangle.P \mid x(y).Q \to P \mid Q\{M/y\}$. With a simple extension to our encodings we may eliminate the data layer by encoding the data objects as processes, showing that from an expressiveness point of view, data communication is orthogonal to the framework. We note that the data language we are considering is *not* linear, and the usage discipline of data in processes is itself also not linear. For instance, the following is a valid typing derivation:

$$
\cfrac{
  \cfrac{
    \cfrac{}{x{:}\tau \vdash x : \tau}
    \quad
    \cfrac{
      \cfrac{\cfrac{}{x{:}\tau, y{:}\sigma \vdash x{:}\tau}}{x{:}\tau \vdash \lambda y{:}\sigma.x : \sigma \to \tau}
      \quad
      \cfrac{}{x{:}\tau; \cdot; \cdot \vdash \mathbf{0} :: z{:}\mathbf{1}}\ \mathsf{1R}
    }{x{:}\tau; \cdot; \cdot \vdash z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} :: z{:}(\sigma \to \tau) \wedge \mathbf{1}}\ \wedge\mathsf{R}
  }{x{:}\tau; \cdot; \cdot \vdash z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} :: z{:}\tau \wedge ((\sigma \to \tau) \wedge \mathbf{1})}\ \wedge\mathsf{R}
}{\cdot; \cdot; \cdot \vdash z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} :: z{:}\tau \supset (\tau \wedge ((\sigma \to \tau) \wedge \mathbf{1}))}\ \supset\mathsf{R}
\qquad (1)
$$

The process at the root of the typing derivation above receives a data element of type τ bound to x and uses it in the two subsequent outputs. The first is a simple forwarding of the received term, whereas the second is that of a non-linear function that discards its argument and returns x.

To First-Order Processes. We now introduce our encoding from Sess$\pi\lambda$ to Sessπ (the core calculus without value passing) via an encoding from Linλ (the simply-typed linear lambda-calculus) to Sessπ. The encodings are defined inductively on session types, processes, types and λ-terms (we omit the purely inductive cases on session types and processes for conciseness).

The encoding on processes $[\![-]\!]$ from Sess$\pi\lambda$ to Sessπ, is defined on *typing derivations*, where we indicate the typing rule at the root of the typing derivation. The encoding $[\![-]\!]_z$, from Linλ to Sessπ, follows the same pattern of Section 3.1.

$$
[\![\tau \wedge A]\!] \triangleq ![\![\tau]\!] \otimes [\![A]\!] \qquad [\![\tau \supset A]\!] \triangleq ![\![\tau]\!] \multimap [\![A]\!] \qquad [\![\tau \to \sigma]\!] \triangleq ![\![\tau]\!] \multimap [\![\sigma]\!]
$$

$$
\begin{array}{lrcl@{\qquad}lrcl}
(\wedge\mathsf{R}) & [\![z\langle M\rangle.P]\!] & \triangleq & \overline{z}\langle x\rangle.(!x(y).[\![M]\!]_y \mid [\![P]\!]) & (\wedge\mathsf{L}) & [\![x(y).P]\!] & \triangleq & x(y).[\![P]\!] \\
(\supset\mathsf{R}) & [\![z(x).P]\!] & \triangleq & z(x).[\![P]\!] & (\supset\mathsf{L}) & [\![x\langle M\rangle.P]\!] & \triangleq & \overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P]\!])
\end{array}
$$

$$
\begin{array}{c}
[\![x]\!]_z \triangleq \overline{x}\langle y\rangle.[y \leftrightarrow z] \qquad [\![\lambda x{:}\tau.M]\!]_z \triangleq z(x).[\![M]\!]_z \\
[\![M\,N]\!]_z \triangleq (\nu y)([\![M]\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![N]\!]_w \mid [y \leftrightarrow z]))
\end{array}
$$

[1]For simplicity, in this section, we define the process semantics through a reduction relation.

The encoding addresses the non-linear usage of data elements in processes by encoding the types $\tau \wedge A$ and $\tau \supset A$ as $!\llbracket\tau\rrbracket \otimes \llbracket A\rrbracket$ and $!\llbracket\tau\rrbracket \multimap \llbracket A\rrbracket$, respectively. Thus, sending and receiving of data is codified as the sending and receiving of channels of type !, which therefore can be used non-linearly. Moreover, since data terms are themselves non-linear, the $\tau \to \sigma$ type is encoded as $!\llbracket\tau\rrbracket \multimap \llbracket\sigma\rrbracket$, following Girard's embedding of intuitionistic logic in linear logic [23].

At the level of processes, offering a session of type $\tau \wedge A$ (i.e. a process of the form $z\langle M\rangle.P$) is encoded according to the translation of the type: we first send a *fresh* name x which will be used to access the encoding of the term M. Since M can be used an arbitrary number of times by the receiver, we guard the encoding of M with a replicated input, proceeding with the encoding of P accordingly. Using a session of type $\tau \supset A$ follows the same principle. The input cases (and the rest of the process constructs) are completely homomorphic.

The encoding of λ-terms follows Girard's decomposition of the intuitionistic function space [70]. The λ-abstraction is translated as input. Since variables in a λ-abstraction may be used non-linearly, the case for variables and application is slightly more intricate: to encode the application $M\,N$ we compose M in parallel with a process that will send the "reference" to the function argument N which will be encoded using replication, in order to handle the potential for 0 or more usages of variables in a function body. Respectively, a variable is encoded by performing an output to trigger the replication and forwarding accordingly. Without loss of generality, we assume variable names and their corresponding replicated counterparts match, which can be achieved through α-conversion before applying the translation. We exemplify our encoding as follows:

$$\begin{aligned}\llbracket z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0}\rrbracket &= z(x).\overline{z}\langle w\rangle.(!w(u).\llbracket x\rrbracket_u \mid \overline{z}\langle v\rangle.(!v(i).\llbracket\lambda y{:}\sigma.x\rrbracket_i \mid \mathbf{0}))\\ &= z(x).\overline{z}\langle w\rangle.(!w(u).\overline{x}\langle y\rangle.[y \leftrightarrow u] \mid \overline{z}\langle v\rangle.(!v(i).i(y).\overline{x}\langle t\rangle.[t \leftrightarrow i] \mid \mathbf{0}))\end{aligned}$$

Properties of the Encoding. We discuss the correctness of our encoding. We can straightforwardly establish that the encoding preserves typing.

Lemma 5.1 (Type Soundness of $\llbracket-\rrbracket_z$ Encoding).

(1) *If* $\Psi \vdash M : \tau$ *then* $\llbracket\Psi\rrbracket; \cdot \vdash \llbracket M\rrbracket_z :: z{:}\llbracket\tau\rrbracket$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $\llbracket\Psi\rrbracket, \llbracket\Gamma\rrbracket; \llbracket\Delta\rrbracket \vdash \llbracket P\rrbracket :: z{:}\llbracket A\rrbracket$

Proof. Straightforward induction on the given typing derivations. □

To show that our encoding is operationally sound and complete, we capture the interaction between substitution on λ-terms and the encoding into processes through logical equivalence. Consider the following reduction of a process:

$$\begin{aligned}&(\nu z)(z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} \mid z\langle\lambda w{:}\tau_0.w\rangle.P)\\ &\quad\to (\nu z)(z\langle\lambda w{:}\tau_0.w\rangle.z\langle(\lambda y{:}\sigma.\lambda w{:}\tau_0.w)\rangle.\mathbf{0} \mid P)\end{aligned} \tag{2}$$

Given that substitution in the target session π-calculus amounts to renaming, whereas in the λ-calculus we replace a variable for a term, the relationship between the encoding of a substitution $M\{N/x\}$ and the encodings of M and N corresponds to the composition of the encoding of M with that of N, but where the encoding of N is guarded by a replication, codifying a form of explicit non-linear substitution. We note the contrast with the notions of compositionality for the linear setting (Lemma 3.4), where we separate shared variable usage, which requires replication, from linear variable usage, which does not.

Lemma 5.2 (Compositionality). *Let* $\Psi, x{:}\tau \vdash M : \sigma$ *and* $\Psi \vdash N : \tau$. *We have that* $\llbracket M\{N/x\}\rrbracket_z \approx_{\mathsf{L}} (\nu x)(\llbracket M\rrbracket_z \mid !x(y).\llbracket N\rrbracket_y)$

Proof. See Appendix A.3.1. □

Revisiting the process to the left of the arrow in Equation 2 we have:

$$\begin{array}{l}[\![(\nu z)(z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} \mid z\langle\lambda w{:}\tau_0.w\rangle.P)]\!] \\ = (\nu z)([\![z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0}]\!]_z \mid \overline{z}\langle x\rangle.(!x(b).[\![\lambda w{:}\tau_0.w]\!]_b \mid [\![P]\!])) \\ \longrightarrow (\nu z,x)(\overline{z}\langle w\rangle.(!w(u).\overline{x}\langle y\rangle.[y\leftrightarrow u] \mid \overline{z}\langle v\rangle.(!v(i).[\![\lambda y{:}\sigma.x]\!]_i \mid \mathbf{0}) \mid !x(b).[\![\lambda w{:}\tau_0.w]\!]_b \mid [\![P]\!]))\end{array}$$

whereas the process to the right of the arrow is encoded as:

$$\begin{array}{l}[\![(\nu z)(z\langle\lambda w{:}\tau_0.w\rangle.z\langle(\lambda y{:}\sigma.\lambda w{:}\tau_0.w)\rangle.\mathbf{0} \mid P)]\!] \\ = (\nu z)(\overline{z}\langle w\rangle.(!w(u).[\![\lambda w{:}\tau_0.w]\!]_u \mid \overline{z}\langle v\rangle.(!v(i).[\![\lambda y{:}\sigma.\lambda w{:}\tau_0.w]\!]_i \mid [\![P]\!])))\end{array}$$

While the reduction of the encoded process and the encoding of the reduct differ syntactically, they are observationally equivalent – the latter inlines the replicated process behaviour that is accessible in the former on x. Having characterised substitution, we can establish operational soundness and completeness for the encoding (see Appendix A.3.1 for proofs of Theorems 5.3 and 5.4 below).

Theorem 5.3 (Operational Soundness – $[\![-]\!]_z$).
(1) *If* $\Psi \vdash M : \tau$ *and* $[\![M]\!]_z \longrightarrow Q$ *then* $M \longrightarrow^+ N$ *such that* $[\![N]\!]_z \approx_{\mathsf{L}} Q$
(2) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $[\![P]\!] \longrightarrow Q$ *then* $P \longrightarrow^+ P'$ *such that* $[\![P']\!] \approx_{\mathsf{L}} Q$

Theorem 5.4 (Operational Completeness – $[\![-]\!]_z$).
(1) *If* $\Psi \vdash M : \tau$ *and* $M \longrightarrow N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $P \approx_{\mathsf{L}} [\![N]\!]_z$
(2) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $[\![P]\!] \longrightarrow^+ R$ *with* $R \approx_{\mathsf{L}} [\![Q]\!]$

The process equivalence in Theorems 5.3 and 5.4 above need not be extended to account for data (although it would be relatively simple to do so), since the processes in the image of the encoding are fully erased of any data elements.

Back to λ-Terms. We extend our encoding of processes to λ-terms to Sess$\pi\lambda$. Our extended translation maps Sess$\pi\lambda$ processes to Linλ-terms, with the session type $\tau \wedge A$ interpreted as a pair type where the first component is replicated. Dually, $\tau \supset A$ is interpreted as a function type where the domain type is replicated. The remaining session constructs are translated as in Section 3.2. By a slight abuse of notation, the translation $(\!|-|\!)$ is overloaded, taking Sess$\pi\lambda$ processes and types to Linλ-terms and types, respectively, but also translating the simply-typed λ-calculus fragment of Sess$\pi\lambda$ to Linλ.

$$(\!|\tau \wedge A|\!) \triangleq !(\!|\tau|\!) \otimes (\!|A|\!) \qquad (\!|\tau \supset A|\!) \triangleq !(\!|\tau|\!) \multimap (\!|A|\!) \qquad (\!|\tau \rightarrow \sigma|\!) \triangleq !(\!|\tau|\!) \multimap (\!|\sigma|\!)$$

$$\begin{array}{llcl llcl}(\wedge\mathsf{L}) & (\!|x(y).P|\!) & \triangleq & \mathsf{let}\ y\otimes x = x\ \mathsf{in}\ \mathsf{let}\ !y = y\ \mathsf{in}\ (\!|P|\!) & (\wedge\mathsf{R}) & (\!|z\langle M\rangle.P|\!) & \triangleq & \langle !(\!|M|\!) \otimes (\!|P|\!)\rangle \\ (\supset\mathsf{R}) & (\!|x(y).P|\!) & \triangleq & \lambda x{:}!(\!|\tau|\!).\mathsf{let}\ !x = x\ \mathsf{in}\ (\!|P|\!) & (\supset\mathsf{L}) & (\!|x\langle M\rangle.P|\!) & \triangleq & (\!|P|\!)\{(x\,!(\!|M|\!))/x\}\end{array}$$

$$(\!|\lambda x{:}\tau.M|\!) \triangleq \lambda x{:}!(\!|\tau|\!).\mathsf{let}\ !x = x\ \mathsf{in}\ (\!|M|\!) \qquad (\!|M\,N|\!) \triangleq (\!|M|\!)\,!(\!|N|\!) \qquad (\!|x|\!) \triangleq x$$

The treatment of non-linear components of processes is identical to our previous encoding: non-linear functions $\tau \rightarrow \sigma$ are translated to linear functions of type $!\tau \multimap \sigma$; a process offering a session of type $\tau \wedge A$ (i.e. a process of the form $z\langle M\rangle.P$, typed by rule $\wedge$R) is translated to a pair where the first component is the encoding of M prefixed with ! so that it may be used non-linearly, and the second is the encoding of P. Non-linear variables are handled at the respective binding sites: a process using a session of type $\tau \wedge A$ is encoded using the elimination form for the pair and the elimination form for the exponential; similarly, a process offering a session of type $\tau \supset A$ is encoded as a λ-abstraction where the bound variable is of type $!(\!|\tau|\!)$. Thus, we use the elimination form for the exponential, ensuring that the typing is correct. We illustrate our encoding:

$$\begin{array}{lcl}(\!|z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0}|\!) & = & \lambda x{:}!(\!|\tau|\!).\mathsf{let}\ !x = x\ \mathsf{in}\ \langle !x \otimes \langle !(\!|\lambda y{:}\sigma.x|\!) \otimes \langle\rangle\rangle\rangle \\ & = & \lambda x{:}!(\!|\tau|\!).\mathsf{let}\ !x = x\ \mathsf{in}\ \langle !x \otimes \langle !(\lambda y{:}!(\!|\sigma|\!).\mathsf{let}\ !y = y\ \mathsf{in}\ x) \otimes \langle\rangle\rangle\rangle\end{array}$$

Properties of the Encoding. Unsurprisingly due to the logical correspondence between natural deduction and sequent calculus presentations of logic, our encoding satisfies both type soundness and operational correspondence (c.f. Theorems 3.7, 3.9, and 3.11).

Lemma 5.5 (Type Soundness of $(\!|-|\!)$ Encoding).

(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $(\!|\Psi|\!), (\!|\Gamma|\!); (\!|\Delta|\!) \vdash (\!|P|\!) : (\!|A|\!)$

(2) *If* $\Psi \vdash M : \tau$ *then* $(\!|\Psi|\!); \cdot \vdash (\!|M|\!) : (\!|\tau|\!)$

Proof. Straightforward induction on the given typing derivation. □

As before, we establish operational soundness and completeness of the encoding by appealing to a notion of compositionality wrt substitution.

Lemma 5.6 (Compositionality).

(1) *If* $\Psi, x{:}\tau; \Gamma; \Delta \vdash P :: z{:}B$ *and and* $\Psi \vdash M : \tau$ *then* $(\!|P\{M/x\}|\!) =_\alpha (\!|P|\!)\{(\!|M|\!)/x\}$

(2) *If* $\Psi, x{:}\tau \vdash M : \sigma$ *and* $\Psi \vdash N : \tau$ *then* $(\!|M\{N/x\}|\!) =_\alpha (\!|M|\!)\{(\!|N|\!)/x\}$

Proof. By induction on the structure of the given process and term with free variable x. □

Mirroring the development of Section 3.2, we make use of extended reduction $\mapsto$ for processes and full β-reduction $\rightarrow_\beta$ for λ-terms (see Appendix A.3.2 for proofs of Theorems 5.7 and 5.8).

Theorem 5.7 (Operational Soundness – $(\!|-|\!)$).

(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $(\!|P|\!) \rightarrow M$ *then* $P \mapsto^* Q$ *such that* $M =_\alpha (\!|Q|\!)$

(2) *If* $\Psi \vdash M : \tau$ *and* $(\!|M|\!) \rightarrow N$ *then* $M \rightarrow_\beta^+ M'$ *such that* $N =_\alpha (\!|M'|\!)$

Theorem 5.8 (Operational Completeness – $(\!|-|\!)$).

(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \rightarrow Q$ *then* $(\!|P|\!) \rightarrow_\beta^* (\!|Q|\!)$

(2) *If* $\Psi \vdash M : \tau$ *and* $M \rightarrow N$ *then* $(\!|M|\!) \rightarrow^+ (\!|N|\!)$.

Relating the Two Encodings. We prove the two encodings are mutually inverse and preserve the full abstraction properties (we write $=_\beta$ and $=_{\beta\eta}$ for β- and $\beta\eta$-equivalence, respectively).

Theorem 5.9 (Inverse). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} [\![P]\!]$. *If* $\Psi \vdash M : \tau$ *then* $(\!|[\![M]\!]_z|\!) =_\beta (\!|M|\!)$.

Proof. We prove the two statements separately in Appendix A.3.3 (Theorems A.3 and A.4, respectively). □

The equivalences above are formulated between the composition of the encodings applied to P (resp. M) and the process (resp. λ-term) *after* applying the translation embedding the non-linear components into their linear counterparts. This formulation matches more closely that of § 3.3, which applies to linear calculi for which the *target* languages of this section are a strict subset (and avoids the formalisation of process equivalence with terms). We also note that in this setting, observational equivalence and $\beta\eta$-equivalence coincide [5, 45]. Moreover, the extensional flavour of $\approx_{\mathsf{L}}$ includes η-like principles at the process level.

Lemma 5.10. *Let* $\cdot \vdash M : \tau$ *and* $\cdot \vdash V : \tau$ *with* $V \not\rightarrow$. $[\![M]\!]_z \approx_{\mathsf{L}} [\![V]\!]_z$ *iff* $(\!|M|\!) \rightarrow_{\beta\eta}^* (\!|V|\!)$

Theorem 5.11 (Full Abstraction).
Let:

(a) $\cdot \vdash M : \tau$ *and* $\cdot \vdash N : \tau$;

(b) $\cdot \vdash P :: z{:}A$ *and* $\cdot \vdash Q :: z{:}A$.

We have that $(\!|M|\!) =_{\beta\eta} (\!|N|\!)$ *iff* $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$ *and* $[\![P]\!] \approx_{\mathsf{L}} [\![Q]\!]$ *iff* $(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$.

Proof. Following the development of previous sections, we prove the two statements separately in Theorems A.5 and A.6, respectively, in Appendix A.3.3. The proof of Theorem A.5 relies on Lemma 5.10. □

We establish full abstraction for the encoding of λ-terms into processes (Theorem 5.11 (1)) in two steps: The completeness direction (i.e. from left-to-right) follows from operational completeness and strong normalisation of the λ-calculus. The soundness direction uses operational soundness. The proof of Theorem 5.11(2) uses the same strategy of Theorem 3.16, appealing to the inverse theorems.

5.2 Higher-Order Session Processes – Sess$\pi\lambda^+$

We extend the value-passing framework of the previous section, accounting for process-passing (i.e. the higher-order) in a session-typed setting. As shown in previous work [71], we achieve this by adding to the data layer a *contextual monad* that encapsulates (open) session-typed processes as data values, with a corresponding elimination form in the process layer. We dub this calculus Sess$\pi\lambda^+$.

$$\begin{array}{lcl@{\qquad}lcl} P, Q & ::= & \cdots \mid x \leftarrow M \leftarrow \overline{y_i}; Q & M.N & ::= & \cdots \mid \{x \leftarrow P \leftarrow \overline{y_i{:}A_i}\} \\ \tau, \sigma & ::= & \cdots \mid \{\overline{x_j{:}A_j} \vdash z{:}A\} \end{array}$$

The type $\{\overline{x_j{:}A_j} \vdash z{:}A\}$ is the type of a term which encapsulates an open process that uses the linear channels $\overline{x_j{:}A_j}$ and offers A along channel z. This formulation has the added benefit of formalising the integration of session-typed processes in a functional language and forms the basis for the concurrent programming language SILL [53, 71]. The typing rules for the new constructs are (for simplicity we assume no shared channels in process monads):

$$\frac{\Psi; \cdot; \overline{x_i{:}A_i} \vdash P :: z{:}A}{\Psi \vdash \{z \leftarrow P \leftarrow \overline{x_i{:}A_i}\} : \{\overline{x_i{:}A_i} \vdash z{:}A\}}\ \{\}I$$

$$\frac{\Psi \vdash M : \{\overline{x_i{:}A_i} \vdash x{:}A\} \quad \Delta_1 = \overline{y_i{:}A_i} \quad \Psi; \Gamma; \Delta_2, x{:}A \vdash Q :: z{:}C}{\Psi; \Gamma; \Delta_1, \Delta_2 \vdash x \leftarrow M \leftarrow \overline{y_i}; Q :: z{:}C}\ \{\}E$$

Rule $\{\}I$ embeds processes in the term language by essentially quoting an open process that is well-typed according to the type specification in the monadic type. Dually, rule $\{\}E$ allows for processes to use monadic values through composition that *consumes* some of the ambient channels in order to provide the monadic term with the necessary context (according to its type). These constructs are discussed in substantial detail in [71]. The reduction semantics of the process construct is given by (we tacitly assume that the names $\overline{y}$ and c do not occur in P and omit the congruence case):

$$(c \leftarrow \{z \leftarrow P \leftarrow \overline{x_i{:}A_i}\} \leftarrow \overline{y_i}; Q) \rightarrow (\nu c)(P\{\overline{y}/\overline{x_i}\{c/z\}\} \mid Q)$$

The semantics allows for the underlying monadic term M to evaluate to a (quoted) process P. The process P is then executed in parallel with the continuation Q, sharing the linear channel c for subsequent interactions. We illustrate the higher-order extension with following typed process (we write $\{x \leftarrow P\}$ when P does not depend on any linear channels and assume $\vdash Q :: d{:}\mathsf{Nat} \wedge \mathbf{1}$):

$$P \triangleq (\nu c)(c\langle\{d \leftarrow Q\}\rangle.c(x).\mathbf{0} \mid c(y).d \leftarrow y; d(n).c\langle n\rangle.\mathbf{0}) \tag{3}$$

Process P above gives an abstract view of a communication idiom where a process (the left-hand side of the parallel composition) sends another process Q which potentially encapsulates some complex computation. The receiver then *spawns* the execution of the received process and inputs from it a result value that is sent back to the original sender. An execution of P is given by:

$$\begin{array}{rcl} P \longrightarrow (\nu c)(c(x).\mathbf{0} \mid d \leftarrow \{d \leftarrow Q\}; d(n).c\langle n\rangle.\mathbf{0}) & \longrightarrow & (\nu c)(c(x).\mathbf{0} \mid (\nu d)(Q \mid d(n).c\langle n\rangle.\mathbf{0})) \\ & \longrightarrow^{+} & (\nu c)(c(x).\mathbf{0} \mid c\langle 42\rangle.\mathbf{0}) \longrightarrow \mathbf{0} \end{array}$$

Given the seminal work of Sangiorgi [65], such a representation naturally begs the question of whether or not we can develop a *typed* encoding of higher-order processes into the first-order setting. Indeed, we can achieve such an encoding with a fairly simple extension of the encoding of § 5 to Sess$\pi\lambda^{+}$ by observing that monadic values are processes that need to be potentially provided with extra sessions in order to be executed correctly. For instance, a term of type $\{x{:}A \vdash y{:}B\}$ denotes a process that given a session x of type A will then offer $y{:}B$. Exploiting this observation we encode this type as the session $A \multimap B$, ensuring subsequent usages of such a term are consistent with this interpretation.

$$\begin{array}{rcl} [\![\{\overline{x_j{:}A_j} \vdash z{:}A\}]\!] & \triangleq & \overline{[\![A_j]\!]} \multimap [\![A]\!] \\ [\![\{x \leftarrow P \leftarrow \overline{y_i}\}]\!]_z & \triangleq & z(y_0).\ldots.z(y_n).[\![P\{z/x\}]\!] \quad (z \notin fn(P)) \\ [\![x \leftarrow M \leftarrow \overline{y_i}; Q]\!] & \triangleq & (\nu x)([\![M]\!]_x \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid [\![Q]\!])\ldots)) \end{array}$$

To encode the monadic type $\{\overline{x_j{:}A_j} \vdash z{:}A\}$, denoting the type of process P that is typed by $\overline{x_j{:}A_j} \vdash P :: z{:}A$, we require that the session in the image of the translation specifies a sequence of channel inputs with behaviours $\overline{A_j}$ that make up the linear context. After the contextual aspects of the type are encoded, the session will then offer the (encoded) behaviour of A. Thus, the encoding of the monadic type is $[\![A_0]\!] \multimap \ldots \multimap [\![A_n]\!] \multimap [\![A]\!]$, which we write as $\overline{[\![A_j]\!]} \multimap [\![A]\!]$. The encoding of monadic expressions adheres to this behaviour, first performing the necessary sequence of inputs and then proceeding inductively. Finally, the encoding of the elimination form for monadic expressions behaves dually, composing the encoding of the monadic expression with a sequence of outputs that instantiate the consumed names accordingly (via forwarding). The encoding of process P from Equation 3 is thus:

$$\begin{array}{l} [\![P]\!] = (\nu c)([\![c\langle\{d \leftarrow Q\}\rangle.c(x).\mathbf{0}]\!] \mid [\![c(y).d \leftarrow y; d(n).c\langle n\rangle.\mathbf{0}]\!]) \\ = (\nu c)(\overline{c}\langle w\rangle.(!w(d).[\![Q]\!] \mid c(x).\mathbf{0})c(y).(\nu d)(\overline{y}\langle b\rangle.[b \leftrightarrow d] \mid d(n).\overline{c}\langle m\rangle.(\overline{n}\langle e\rangle.[e \leftrightarrow m] \mid \mathbf{0}))) \end{array}$$

Properties of the Encoding. As in our previous development, we can show that our encoding for Sess$\pi\lambda^{+}$ is type sound and satisfies operational correspondence (c.f. Appendix A.4.1).

Lemma 5.12 (Type Soundness – $[\![-]\!]_z$).

(1) *If* $\Psi \vdash M : \tau$ *then* $[\![\Psi]\!]; \cdot \vdash [\![M]\!]_z :: z{:}[\![\tau]\!]$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![\Psi]\!], [\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![P]\!] :: z{:}[\![A]\!]$

Proof. By induction on the given typing derivation. □

Theorem 5.13 (Operational Soundness – $[\![-]\!]_z$).

(1) *If* $\Psi \vdash M : \tau$ *and* $[\![M]\!]_z \longrightarrow Q$ *then* $M \longrightarrow^{+} N$ *such that* $[\![N]\!]_z \approx_{\mathsf{L}} Q$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $[\![P]\!] \longrightarrow Q$ *then* $P \longrightarrow^{+} P'$ *such that* $[\![P']\!] \approx_{\mathsf{L}} Q$

Theorem 5.14 (Operational Completeness – $[\![-]\!]_z$).

(1) *If* $\Psi \vdash M : \tau$ *and* $M \longrightarrow N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $P \approx_{\mathsf{L}} [\![N]\!]_z$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $[\![P]\!] \longrightarrow^{+} R$ *with* $R \approx_{\mathsf{L}} [\![Q]\!]$

Back to λ-Terms. We encode Sess$\pi\lambda^+$ into λ-terms, extending § 5 with:

$$(\!|\{\overline{x_i{:}A_i} \vdash z{:}A\}|\!) \triangleq \overline{(\!|A_i|\!)} \multimap (\!|A|\!)$$

$$(\!|x \leftarrow M \leftarrow \overline{y_i}; Q|\!) \triangleq (\!|Q|\!)\{((\!|M|\!)\,\overline{y_i})/x\} \qquad (\!|\{x \leftarrow P \leftarrow \overline{w_i}\}|\!) \triangleq \lambda w_0.\dots.\lambda w_n.(\!|P|\!)$$

The encoding translates the monadic type $\{\overline{x_i{:}A_i} \vdash z{:}A\}$ as a linear function $\overline{(\!|A_i|\!)} \multimap (\!|A|\!)$, which captures the fact that the underlying value must be provided with terms satisfying the requirements of the linear context. At the level of terms, the encoding for the monadic term constructor follows its type specification, generating a nesting of λ-abstractions that closes the term and proceeding inductively. For the process encoding, we translate the monadic application construct analogously to the translation of a linear cut, but applying the appropriate variables to the translated monadic term (which is of function type). We remark the similarity between our encoding and that of the previous section, where monadic terms are translated to a sequence of inputs (here a nesting of λ-abstractions). Our encoding satisfies type soundness and operational correspondence, as usual.

Lemma 5.15 (Type Soundness – $(\!|-|\!)$).
(1) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *then* $(\!|\Psi|\!), (\!|\Gamma|\!); (\!|\Delta|\!) \vdash (\!|P|\!) : (\!|A|\!)$
(2) *If* $\Psi \vdash M : \tau$ *then* $(\!|\Psi|\!); \cdot \vdash (\!|M|\!) : (\!|\tau|\!)$

Proof. By induction on the give typing derivation. □

The proofs of operational soundness and completeness are given in Appendix A.4.2. As in the corresponding encoding from Polyπ to Linear-F, we use full β-reduction to make the results more precise and without needing to appeal to extra-logical features such as a general let-binder.

Theorem 5.16 (Operational Soundness – $(\!|-|\!)$).
(1) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $(\!|P|\!) \to M$ *then* $P \mapsto^* Q$ *such that* $M =_\alpha (\!|Q|\!)$
(2) *If* $\Psi \vdash M : \tau$ *and* $(\!|M|\!) \to N$ *then* $M \to_\beta^+ M'$ *such that* $N =_\alpha (\!|M'|\!)$

Theorem 5.17 (Operational Completeness – $(\!|-|\!)$).
(1) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $P \to Q$ *then* $(\!|P|\!) \to_\beta^* (\!|Q|\!)$
(2) *If* $\Psi \vdash M : \tau$ *and* $M \to N$ *then* $(\!|M|\!) \to^+ (\!|N|\!)$

As before, we establish that the two encodings are mutually inverse and fully abstract (see Appendix A.4.3).

Theorem 5.18 (Inverse Encodings). *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *then* $[\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} [\![P]\!]$. *Also, if* $\Psi \vdash M : \tau$ *then* $(\!|[\![M]\!]_z|\!) =_\beta (\!|M|\!)$.

Theorem 5.19 (Full Abstraction – Terms). *Let* $\cdot \vdash M : \tau$ *and* $\cdot \vdash N : \tau$. $(\!|M|\!) =_{\beta\eta} (\!|N|\!)$ *iff* $[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$.

Theorem 5.20 (Full Abstraction – Processes). *Let* $\cdot \vdash P :: z{:}A$ *and* $\cdot \vdash Q :: z{:}A$. $[\![P]\!] \approx_{\mathsf{L}} [\![Q]\!]$ *iff* $(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$.

Further showcasing the applications of our development, we obtain a novel strong normalisation result for this higher-order session-calculus "for free", through encoding to the λ-calculus.

To achieve this, we rely on a slight modification of the encoding from processes to λ-terms by considering the encoding of derivations ending with the copy rule as follows (we write $(\!|-|\!)^+$ for this revised encoding):

$$(\!|(\nu x)u\langle x\rangle.P|\!)^+ \triangleq \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P|\!)^+\{u/x\}$$

All other cases of the encoding are as before. We now show that the revised encoding preserves all the desirable properties of the previous sections and then show how we can use it to prove strong normalisation.

It is immediate that the revised encoding preserves typing. The revised encoding allows us to formulate a tighter version of operational completeness, where process moves are matched by one or more β-reduction steps (as opposed to zero or more):

Theorem 5.21 (Operational Completeness). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \to Q$ *then* $(\!|P|\!)^+ \to_\beta^+ (\!|Q|\!)^+$

Proof. See Appendix A.5. □

We remark that with this revised encoding, operational soundness becomes:

Theorem 5.22 (Operational Soundness). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $(\!|P|\!)^+ \to M$ *then* $P \mapsto^* Q$ *such that* $(\!|Q|\!) \to^* M$.

Proof. See Appendix A.5. □

The revised encoding remains mutually inverse with the $[\![-]\!]_z$ encoding.

Theorem 5.23 (Inverse). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![(\!|P|\!)^+]\!]_z \approx_{\mathsf{L}} [\![P]\!]$

Having established the key properties of the encoding, we now show strong normalisation.

Theorem 5.24 (Strong Normalisation). *Let* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$. *There is no infinite reduction sequence starting from* P.

Proof. The result follows from the operational completeness result above (Lemma 5.21), which requires every process reduction to be matched with one or more reductions in the λ-calculus. We can thus prove our result via strong normalisation of $\to_\beta$: Assume an infinite reduction sequence $P \to P' \to P'' \to \ldots$, by completeness this implies that there must exist an infinite sequence $(\!|P|\!) \to_\beta^+ (\!|P'|\!) \to_\beta^+ (\!|P''|\!) \to_\beta^+ \ldots$, deriving a contradiction. □

6 RELATED WORK

Process Encodings of Functions. Toninho et al. [70] study encodings of the simply-typed λ-calculus in a logically motivated session π-calculus, via encodings to the linear λ-calculus, as a means to explicate various operational semantics. Our work differs since they do not study polymorphism nor encodings of processes as functions. Moreover, we provide deeper insights through our applications of the encodings. Full abstraction or inverse properties are not studied.

Sangiorgi [62] uses a fully abstract compilation from the higher-order π-calculus (HOπ) to the π-calculus to study full abstraction for Milner's encodings of the λ-calculus. The work shows that Milner's encoding of the lazy λ-calculus can be recovered by restricting the semantic domain of processes (the so-called *restrictive* approach) or by enriching the λ-calculus with suitable constants. This work was later refined in [64], which does not use HOπ and considers an operational equivalence on λ-terms called *open applicative bisimulation* which coincides with Lévy-Longo tree equality. The work [66] studies general conditions under which encodings of the λ-calculus in the π-calculus are fully abstract wrt Lévy-Longo and Böhm Trees, which are then applied to several encodings of (call-by-name) λ-calculus. The works above deal with *untyped calculi*, and so reverse encodings are unfeasible. In a broader sense, our approach takes the restrictive approach using linear logic-based session typing and the induced observational equivalence. We use a λ-calculus with booleans as observables and reason with a Morris-style equivalence instead of tree equalities. It would be an interesting future work to apply the conditions in [66] in our typed setting.

Recently, Balzer et al. [4] study the problem of encoding untyped asynchronous communication in a session-typed π-calculus based on intuitionistic linear logic with *manifest sharing* by means of a universal (recursive) session type, akin to that used to encode the untyped λ-calculus in typed λ-calculus with recursive types. Their work considers properties of the encoding up-to contextual closure but does not develop typed behavioral equivalences as we do, leaving open the problems of full abstraction or completeness. Their work does not develop encodings to or from λ-calculi. It would be interesting to study notions of typed behavioural equivalences in settings with sharing and recursive types and see the status of their encoding up-to behavioural equivalence. A natural follow-up of their work would be to study what substructural λ-calculus [54, Chapter 1] can faithfully encode their session typed language.

Wadler [76] shows a correspondence between a linear functional language with session types GV and a session-typed process calculus with polymorphism based on classical linear logic CP. Along the lines of this work, Lindley and Morris [37], in an exploration of inductive and coinductive session types through the addition of least and greatest fixed points to CP and GV, develop an encoding from a linear λ-calculus with session primitives (Concurrent μGV) to a pure linear λ-calculus (Functional μGV) via a CPS transformation. They also develop translations between μCP and Concurrent μGV, extending [36]. Mapping to the terminology used in our work [25], their encodings are shown to be operationally complete, but no results are shown for the operational soundness directions and neither full abstraction nor inverse properties are studied. In addition, their operational characterisations do not compose across encodings. For instance, while strong normalisation of Functional μGV implies the same property for Concurrent μGV through their operationally complete encoding, the encoding from μCP to μGV does not necessarily preserve this property.

Types for π-calculi delineate sequential behaviours by restricting composition and name usages, limiting the contexts in which processes can interact. Therefore typed equivalences offer a *coarser* semantics than untyped semantics. Pierce and Sangiorgi [56] first observed semantic consequences of typed equivalences, demonstrating that the observational congruence under the IO-subtyping can prove correctness of the optimal version of Milner's λ-encoding. This was impossible in the π-calculus without controlling IO channel usages by types. After [56], many works on typed π-calculi have investigated correctness of Milner's encodings in order to examine powers of proposed typing systems.

As an alternative approach, Berger et al. [7] study an affine typing system of the π-calculus and examine its expressiveness, showing encodings of call-by-value/name PCFs to be fully abstract. This work was extended to encode the λ-calculus with sum and product types into linear causal types [78]. Berger et al. [8] further study an encoding of System F in a polymorphic linear π-calculus, showing it to be fully abstract. Their typing systems and proofs are much more complex due to the fine-grained constraints from game semantics. Moreover, none of their work studies a reverse encoding.

Orchard and Yoshida [47] develop embeddings to-and-from PCF with parallel effects and a session-typed π-calculus, but only develop operational correspondence and semantic soundness results, leaving the full abstraction problem open.

Polymorphism and Typed Behavioural Semantics. The work of [11] studies parametric session polymorphism for the intuitionistic setting, developing a behavioural equivalence that captures parametricity, which is used (denoted as $\approx_{\mathsf{L}}$) in our paper. Their work does not address inductive or coinductive types, which we obtain for free by virtue of our mutually inverse encodings. The work [56] introduces a typed bisimilarity for polymorphism in the π-calculus. Their bisimilarity is of an intensional flavour, whereas the one used in our work follows the extensional style of

Reynolds [59]. Their typing discipline (originally from [75], which also develops type-preserving encodings of polymorphic λ-calculus into polymorphic π-calculus) differs significantly from the linear logic-based session typing of our work (e.g. theirs does not ensure deadlock-freedom). A key observation in their work is the coarser nature of typed equivalences with polymorphism (in analogy to those for IO-subtyping [55]) and their interaction with channel aliasing, suggesting a use of typed semantics and encodings of the π-calculus for fine-grained analyses of program behaviour.

In the higher-order process setting, Sangiorgi [61] was the first to propose encodings of process-passing as channel-passing. Higher-order session calculi and their encodings have been studied in [35]. Termination for higher-order processes has been studied in [17, 18].

F-Algebras and Linear-F. The use of initial and final (co)algebras to give a semantics to inductive and coinductive types dates back to Mendler [40], with their strong definability in System F appearing in [2] and [27] (for the parametric PER model of System F in the former and classes of models in the latter). The definability of inductive and coinductive types using parametricity also appears in [58] in the context of a logic for parametric polymorphism and later in [10] in a linear variant of such a logic. The work of [79] studies parametricity for the polymorphic linear λ-calculus of this work, developing encodings of a few inductive types but not the initial (or final) algebraic encodings in their full generality. Inductive and coinductive session types in a logical process setting appear in [72] and [37]. Both works consider a calculus with built-in recursion – the former in an intuitionistic setting where a process that offers a (co)inductive protocol is composed with another that consumes the (co)inductive protocol and the latter in a classical framework where composed recursive session types are dual each other.

Recently, Toninho and Yoshida [74] developed a direct encoding of inductive and coinductive session types in the polymorphic session calculus, justified using the theory of initial algebras and final co-algebras in a processes-as-morphisms viewpoint. Their work is an alternative formulation of the development of § 4, where instead of deriving inductive and coinductive session types and their associated combinators from encodings from System F, inductive and coinductive sessions are constructed directly in the process language using an algebraic approach, with the construction being validated through semantic reasoning.

Encoding-Based Programming Language Implementations of Session Types. Encodings of session types or session π-calculi have been used to implement session primitives in mainstream programming languages. See a recent survey in Haskell [46].

In the area of linear logic-based session calculi, we highlight the work [70], which employs Girard's original encodings of intuitionistic logic in linear logic to study evaluation strategies in the λ-calculus, giving a logically motivated account of *futures*. We also highlight the encodings of Lindley and Morris [36] between a functional language with session primitives (Wadler's GV) and a process algebra with sessions, effectively providing a semantics to Wadler's GV through the encoding. This, combined with the subsequent encodings of fixed-points [37], can be seen as the semantic foundation for the works extending the web-based programming language Links with session types [19, 20, 38]. We further note the addition of session-based concurrency to the language C0 [69, 77], drawing upon the semantic foundation provided by the encodings for the intuitionistic setting [70, 73].

In a wider context of session types, Scalas and Yoshida [68] use an encoding of the binary session calculus into the linear π-calculus [16] to implement binary session types in Scala. This work is extended by Scalas et al. [67] to implement multiparty session types in Scala based on the encoding of the multiparty session π-calculus into the linear π-calculus. The encoding of binary session types in an effect system is used to design a session-typed library in Haskell [47]. In OCaml, Padovani

[48] implements context free session types providing two kinds of encodings from context free session types into functional data structures. A different approach is taken in the work of Imai et al. [34] where session types are encoded leveraging parametric polymorphism in OCaml to statically ensure linear usage of channels. Extending this approach, Imai et al. [32] propose a library for *global combinators*, which are a set of functions for writing and verifying multiparty protocols in OCaml. By encoding a set of local types to a data structure called a *channel vector*, local types are automatically inferred from a global combinator, statically providing linear channel usage in end-point processes.

7 CONCLUSION AND FUTURE WORK

This work answers the question of what kind of type discipline of the π-calculus can exactly capture and is captured by λ-calculus behaviours, dating back to Milner [42] who asks "how to *exactly* match the behavioural semantics induced upon the encodings of the λ-calculus with that of the λ-calculus". Our answer is given by showing the first mutually inverse and fully abstract encodings between two calculi with polymorphism, one being the Polyπ session calculus based on intuitionistic linear logic, and the other (a linear) System F. This further demonstrates that the original linear logic-based articulation of sessions [12] (and subsequent studies e.g. [11, 13, 36, 50, 71, 72, 76]) provides a clear and applicable tool for a wide range of session-based interactions. By exploiting the proof theoretic equivalences between natural deduction and sequent calculus we develop mutually inverse and fully abstract encodings, which naturally extend to more intricate settings such as process passing (in the sense of HOπ). Our encodings also enable us to derive properties of the π-calculi "for free". Specifically, we show how to obtain adequate representations of least and greatest fixed points in Polyπ through the encoding of initial and final (co)algebras in the λ-calculus. We also straightforwardly derive a strong normalisation result for the higher-order session calculus, which otherwise involves non-trivial proof techniques [8, 11, 17, 18, 50]. Future work includes extensions to the classical linear logic-based framework, including multiparty session types [14, 15].

Our work thus shows that the session-based interpretation of linear logic is fully compatible with the standard semantics of (typed) lambda-calculus, allowing us to uniformly represent value passing and even higher-order process passing. Such results can be seen has both positive and negative: on one hand, session types in this logically-grounded sense can be seen to be fundamentally not about non-determinism (in the sense of non-confluent computation) but rather about the well-structuring of *confluent* interactive programs, as made clear by full abstraction; on the other hand, our results show that a functional language with session types based on the session interpretation of linear logic, e.g. SILL [53, 71]) can include higher-order processes either as primitive or through encoding, and remain semantically well-behaved.

Following the line of work on shallow embeddings of session types [32–34, 46, 48, 67, 68], we plan to develop encoding-based implementations of this work as embedded DSLs. This would potentially enable an exploration of algebraic constructs beyond initial and final co-algebras in a session programming setting. Exploring a processes-as-morphisms viewpoint, recent work [74] investigates a *direct* encodinging of inductive and coinductive session types, justified via the theory of initial algebras and final co-algebras. The correctness of the encoding (i.e. universality) relies crucially on parametricity and the associated relational lifting of sessions. We plan to further study the meaning of functors, natural transformations and related constructions [9] in a session-typed setting, both from a more fundamental viewpoint but also in terms of programming patterns.

Acknowledgements. We thank TOPLAS reviewers for their helpful comments and suggestions. We thank Domenico Ruoppolo for his detailed comments on the first version of this article; and Uwe Nestmann and Kristin Peters for their suggestions for the literature of expressiveness. The work is supported by NOVA LINCS (UIDB/04516/2020), EPSRC EP/N028201/1, EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, and EPSRC/NCSC/GCHQ VeTSS.

REFERENCES

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[2] E. S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. 1990. Functorial Polymorphism. *Theor. Comput. Sci.* 70, 1 (1990), 35–64. https://doi.org/10.1016/0304-3975(90)90151-7

[3] Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *PACMPL* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

[4] Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

[5] Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report ECS-LFCS-96-347. School of Informatics, University of Edinburgh.

[6] P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*. 121–135. https://doi.org/10.1007/BFb0022251

[7] Martin Berger, Kohei Honda, and Nobuko Yoshida. 2001. Sequentiality and the π-Calculus. In *Proc. TLCA'01 (LNCS)*, Vol. 2044. 29–45.

[8] Martin Berger, Kohei Honda, and Nobuko Yoshida. 2005. Genericity and the pi-calculus. *Acta Inf.* 42, 2-3 (2005), 83–141. https://doi.org/10.1007/s00236-005-0175-1

[9] Richard Bird and Oege De Moor. 1997. *The Algebra of Programming.* Prentice Hall.

[10] Lars Birkedal, Rasmus Ejlers Møgelberg, and Rasmus Lerchedahl Petersen. 2006. Linear Abadi and Plotkin Logic. *Logical Methods in Computer Science* 2, 5 (2006). https://doi.org/10.2168/LMCS-2(5:2)2006

[11] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

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

[13] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423.

[14] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

[15] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 412–426. https://doi.org/10.4230/LIPIcs.CONCUR.2015.412

[16] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session Types Revisited. In *PPDP '12: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 139–150. https://doi.org/10.1145/2370776.2370794

[17] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2009. Mobile Processes and Termination. In *Semantics and Algebraic Specification*. 250–273.

[18] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010. Termination in higher-order concurrent calculi. *J. Log. Algebr. Program.* 79, 7 (2010), 550–577.

[19] Simon Fowler. 2020. Model-View-Update-Communicate: Session Types Meet the Elm Architecture. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs)*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.14

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

[21] Simon Gay and Antonio Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools.* River Publishers.

[22] Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift* 39 (1935), 176–210.

[23] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[24] Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types.* Cambridge University Press.

[25] Daniele Gorla. 2010. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.* 208, 9 (2010), 1031–1053.

[26] Daniele Gorla and Uwe Nestmann. 2016. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science* 26, 4 (2016), 639–654.

[27] Ryu Hasegawa. 1994. Categorical Data Types in Parametric Polymorphism. *Mathematical Structures in Computer Science* 4, 1 (1994), 71–109. https://doi.org/10.1017/S0960129500000372

[28] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings.* 509–523. https://doi.org/10.1007/3-540-57208-2_35

[29] Kohei Honda. 2012. Session Types and Distributed Computing. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II.* 23. https://doi.org/10.1007/978-3-642-31585-5_4

[30] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings.* 122–138. https://doi.org/10.1007/BFb0053567

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

[32] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming with Global Protocol Combinators. In *34th European Conference on Object-Oriented Programming.* 9:1–9:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.9

[33] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2017. Session-ocaml: A Session-Based Library with Polarities and Lenses. In *Coordination Models and Languages - 19th IFIP WG 6.1 International Conference, COORDINATION 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings.* 99–118. https://doi.org/10.1007/978-3-319-59746-1_6

[34] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-Ocaml: a Session-based Library with Polarities and Lenses. *scico* (2019), 1–50.

[35] Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. 2016. On the Relative Expressiveness of Higher-Order Session Processes. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* 446–475. https://doi.org/10.1007/978-3-662-49498-1_18

[36] Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

[37] Sam Lindley and J. Garrett Morris. 2016. Talking bananas: structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* 434–447. https://doi.org/10.1145/2951913.2951921

[38] Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools.* River Publishers.

[39] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1999. Call-by-name, Call-by-value, Call-by-need and the Linear lambda Calculus. *Theor. Comput. Sci.* 228, 1-2 (1999), 175–210. https://doi.org/10.1016/S0304-3975(98)00358-2

[40] N. P. Mendler. 1987. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987.* 30–36.

[41] Robin Miler. 2001. Speech on receiving an Honorary Degree from the University of Bologna. www.cs.unibo.it/icalp/Lauree_milner.html.

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

[43] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I and II. *Inf. Comput.* 100, 1 (1992), 1–77.

[44] Uwe Nestmann and Benjamin C. Pierce. 2000. Decoding Choice Encodings. *Information and Computation* 163, 1 (2000), 1 – 59. https://doi.org/10.1006/inco.2000.2868
[45] Yo Ohta and Masahito Hasegawa. 2006. A Terminating and Confluent Linear Lambda Calculus. In *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*. 166–180. https://doi.org/10.1007/11805618_13
[46] Dominic Orchard and Nobuko Yoshida. 2017. Session types with linearity in Haskell. In *Behavioural Types: from Theory to Tools*. River Publishers.
[47] Dominic A. Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 568–581. https://doi.org/10.1145/2837614.2837634
[48] Luca Padovani. 2017. Context-Free Session Type Inference. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 804–830. https://doi.org/10.1007/978-3-662-54434-1_30
[49] Joachim Parrow. 2008. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science* 209 (2008), 173 – 186. https://doi.org/10.1016/j.entcs.2008.04.011 Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory (LIX 2006).
[50] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 539–558. https://doi.org/10.1007/978-3-642-28869-2_27
[51] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.* 239 (2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001
[52] Kirstin Peters. 2019. Comparing Process Calculi Using Encodings. In *Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019 (EPTCS)*, Jorge A. Pérez and Jurriaan Rot (Eds.), Vol. 300. 19–38. https://doi.org/10.4204/EPTCS.300.2
[53] Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 3–22. https://doi.org/10.1007/978-3-662-46678-0_1
[54] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.
[55] Benjamin C. Pierce and Davide Sangiorgi. 1996. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science* 6, 5 (1996), 409–453.
[56] Benjamin C. Pierce and Davide Sangiorgi. 2000. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM* 47, 3 (2000), 531–584. https://doi.org/10.1145/337244.337261
[57] Benhamin C. Pierce and David N. Turner. 1990. Pict Programming Language homepage. https://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html.
[58] Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 361–375. https://doi.org/10.1007/BFb0037118
[59] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. 513–523.
[60] John C. Reynolds and Gordon D. Plotkin. 1993. On Functors Expressible in the Polymorphic Typed Lambda Calculus. *Inf. Comput.* 105, 1 (1993), 1–29. https://doi.org/10.1006/inco.1993.1037
[61] Davide Sangiorgi. 1993. From pi-Calculus to Higher-Order pi-Calculus - and Back. In *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*. 151–166. https://doi.org/10.1007/3-540-56610-4_62
[62] Davide Sangiorgi. 1993. An Investigation into Functions as Processes. In *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. 143–159. https://doi.org/10.1007/3-540-58027-1_7
[63] Davide Sangiorgi. 1996. Pi-Calculus, Internal Mobility, and Agent-Passing Calculi. *Theor. Comput. Sci.* 167, 1&2 (1996), 235–274.
[64] Davide Sangiorgi. 2000. Lazy functions and mobile processes. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. 691–720.
[65] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.
[66] Davide Sangiorgi and Xian Xu. 2014. Trees from Functions as Processes. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. 78–92. https://doi.org/10.1007/978-

3-662-44584-6_7

[67] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[68] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming (LIPIcs)*. Dagstuhl, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[69] Miguel Silva, Mário Florido, and Frank Pfenning. 2016. Non-Blocking Concurrent Imperative Programming with Session Types. In *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016 (EPTCS)*, Iliano Cervesato and Maribel Fernández (Eds.), Vol. 238. 64–72. https://doi.org/10.4204/EPTCS.238.7

[70] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2012. Functions as Session-Typed Processes. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 346–360. https://doi.org/10.1007/978-3-642-28729-9_23

[71] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

[72] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2014. Corecursion and Non-divergence in Session-Typed Processes. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*. 159–175. https://doi.org/10.1007/978-3-662-45917-1_11

[73] Bernardo Toninho and Nobuko Yoshida. 2018. On Polymorphic Sessions and Functions - A Tale of Two (Fully Abstract) Encodings. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 827–855. https://doi.org/10.1007/978-3-319-89884-1_29

[74] Bernardo Toninho and Nobuko Yoshida. 2019. Polymorphic Session Processes as Morphisms. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*. 101–117. https://doi.org/10.1007/978-3-030-31175-9_7

[75] David Turner. 1996. *The Polymorphic Pi-Calculus: Theory and Implementation*. Technical Report ECS-LFCS-96-345. School of Informatics, University of Edinburgh.

[76] Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418.

[77] Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016 (EPTCS)*, Iliano Cervesato and Maribel Fernández (Eds.), Vol. 238. 73–82. https://doi.org/10.4204/EPTCS.238.8

[78] Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the pi-calculus. *Inf. Comput.* 191, 2 (2004), 145–202.

[79] Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational Parametricity for a Polymorphic Linear Lambda Calculus. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*. 344–359. https://doi.org/10.1007/978-3-642-17164-2_24

A APPENDIX

A.1 Proofs for § 3.2 – Encoding from Polyπ to Linear-F

Theorem 3.9 (Operational Completeness). *Let $\Omega;\Gamma;\Delta \vdash P :: z{:}A$. If $P \longrightarrow Q$ then $\llparenthesis P \rrparenthesis \longrightarrow^*_\beta \llparenthesis Q \rrparenthesis$.*

Proof. Induction on typing and case analysis on the possibility of reduction.

Case:

$$(\text{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash P_1 :: x{:}A \quad \Omega;\Gamma;\Delta_2, x{:}A \vdash P_2 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(P_1 \mid P_2) :: z{:}C}$$

where $P_1 \longrightarrow P_1'$ or $P_2 \longrightarrow P_2'$.

$\llparenthesis (\nu x)(P_1 \mid P_2) \rrparenthesis = \llparenthesis P_2 \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x\}$ by definition

Subcase: $P_1 \longrightarrow P_1'$

$(\nu x)(P_1 \mid P_2) \longrightarrow (\nu x)(P_1' \mid P_2)$

$\llparenthesis P_1 \rrparenthesis \longrightarrow^*_\beta \llparenthesis P_1' \rrparenthesis$ by i.h.

$\llparenthesis P_2 \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x\} \longrightarrow^*_\beta \llparenthesis P_2 \rrparenthesis\{\llparenthesis P_1' \rrparenthesis/x\}$ by definition

$\llparenthesis (\nu x)(P_1' \mid P_2) \rrparenthesis = \llparenthesis P_2 \rrparenthesis\{\llparenthesis P_1' \rrparenthesis/x\}$ by definition

Subcase: $P_2 \longrightarrow P_2'$

$(\nu x)(P_1 \mid P_2) \longrightarrow (\nu x)(P_1 \mid P_2')$

$\llparenthesis P_2 \rrparenthesis \longrightarrow^*_\beta \llparenthesis P_2' \rrparenthesis$ by i.h.

$\llparenthesis P_2 \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x\} \longrightarrow^*_\beta \llparenthesis P_2' \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x\}$ by definition

$\llparenthesis (\nu x)(P_1 \mid P_2') \rrparenthesis = \llparenthesis P_2' \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x\}$ by definition

Case:

$$(\text{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash x(y).P_1 :: x{:}A \multimap B \quad \Omega;\Gamma;\Delta_2, x{:}A \multimap B \vdash (\nu y)x\langle y\rangle.(Q_1 \mid Q_2) :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(x(y).P_1 \mid (\nu y)x\langle y\rangle.(Q_1 \mid Q_2)) :: z{:}C}$$

$(\nu x)(x(y).P_1 \mid (\nu y)x\langle y\rangle.(Q_1 \mid Q_2)) \longrightarrow (\nu x)((\nu y)(Q_1 \mid P_1) \mid Q_2)$ by reduction

$\llparenthesis (\nu x)(x(y).P_1 \mid (\nu y)x\langle y\rangle.(Q_1 \mid Q_2)) \rrparenthesis = (\llparenthesis Q_2 \rrparenthesis\{(x\ \llparenthesis Q_1 \rrparenthesis)/x\})\{(\lambda y.\llparenthesis P_1 \rrparenthesis)/x\}$ by definition

$(\llparenthesis Q_2 \rrparenthesis\{(x\ \llparenthesis Q_1 \rrparenthesis)/x\})\{(\lambda y.\llparenthesis P_1 \rrparenthesis)/x\} = \llparenthesis Q_2 \rrparenthesis\{((\lambda y.\llparenthesis P_1 \rrparenthesis)\ \llparenthesis Q_1 \rrparenthesis)/x\}$

$\llparenthesis (\nu x)((\nu y)(Q_1 \mid P_1) \mid Q_2) \rrparenthesis = \llparenthesis Q_2 \rrparenthesis\{(\llparenthesis P_1 \rrparenthesis\{\llparenthesis Q_1 \rrparenthesis/y\})/x\}$ by definition

$\llparenthesis Q_2 \rrparenthesis\{((\lambda y.\llparenthesis P_1 \rrparenthesis)\ \llparenthesis Q_1 \rrparenthesis)/x\} \longrightarrow_\beta \llparenthesis Q_2 \rrparenthesis\{(\llparenthesis P_1 \rrparenthesis\{\llparenthesis Q_1 \rrparenthesis/y\})/x\}$ redex

$\llparenthesis (\nu x)((\nu y)(Q_1 \mid P_1) \mid Q_2) \longrightarrow^*_\beta \llparenthesis Q_2 \rrparenthesis\{(\llparenthesis P_1 \rrparenthesis\{\llparenthesis Q_1 \rrparenthesis/y\})/x\}$ by definition

Case:

$$(\text{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash (\nu y)x\langle y\rangle.(P_1 \mid P_2) :: x{:}A \otimes B \quad \Omega;\Gamma;\Delta_2, x{:}A \otimes B \vdash x(y).Q_1 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)((\nu y)x\langle y\rangle.(P_1 \mid P_2) \mid x(y).Q_1) :: z{:}C}$$

$(\nu x)((\nu y)x\langle y\rangle.(P_1 \mid P_2) \mid x(y).Q_1) \longrightarrow (\nu x)(P_2 \mid (\nu y)(P_1 \mid Q_1))$ by reduction

$\llparenthesis (\nu x)((\nu y)x\langle y\rangle.(P_1 \mid P_2) \mid x(y).Q_1) \rrparenthesis = \mathsf{let}\ x \otimes y = \langle \llparenthesis P_2 \rrparenthesis \otimes \llparenthesis P_1 \rrparenthesis \rangle\ \mathsf{in}\ \llparenthesis Q_1 \rrparenthesis$

$\llparenthesis (\nu x)(P_2 \mid (\nu y)(P_1 \mid Q_1)) \rrparenthesis = \llparenthesis Q_1 \rrparenthesis\{\llparenthesis P_2 \rrparenthesis/x\}\{\llparenthesis P_1 \rrparenthesis/y\}$ by def.

$\mathsf{let}\ x \otimes y = \langle \llparenthesis P_2 \rrparenthesis \otimes \llparenthesis P_1 \rrparenthesis \rangle\ \mathsf{in}\ \llparenthesis Q_1 \rrparenthesis \longrightarrow \llparenthesis Q_1 \rrparenthesis\{\llparenthesis P_2 \rrparenthesis/x\}\{\llparenthesis P_1 \rrparenthesis/y\}$

Case:

$$(\text{cut}^!)\ \frac{\Omega;\Gamma;\cdot \vdash P_1 :: x{:}A \quad \Omega;\Gamma, u{:}A;\Delta \vdash (\nu x)u\langle x\rangle.Q_1 :: z{:}C}{\Omega;\Gamma;\Delta \vdash (\nu u)(!u(x).P_1 \mid (\nu x)u\langle x\rangle.Q_1) :: z{:}C}$$

$(\nu u)(!u(x).P_1 \mid (\nu x)u\langle x\rangle.Q_1) \longrightarrow (\nu u)(!u(x).P_1 \mid (\nu x)(P_1 \mid Q_1))$ by reduction

$\llparenthesis (\nu u)(!u(x).P_1 \mid (\nu x)u\langle x\rangle.Q_1) \rrparenthesis = \llparenthesis Q_1 \rrparenthesis\{u/x\}\{\llparenthesis P_1 \rrparenthesis/u\}$

$= \llparenthesis Q_1 \rrparenthesis\{\llparenthesis P_1 \rrparenthesis/x, \llparenthesis P_1 \rrparenthesis/u\}$ by def.

$\llparenthesis(\nu u)(!u(x).P_1 \mid (\nu x)(P_1 \mid Q_1))\rrparenthesis = (\llparenthesis Q_1\rrparenthesis\{\llparenthesis P_1\rrparenthesis/x\})\{\llparenthesis P_1\rrparenthesis/u\}$

Case:

$$(\text{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash x(Y).P_1 :: x{:}\forall Y.A \quad \Omega;\Gamma;\Delta_2, x{:}\forall Y.A \vdash x\langle B\rangle.Q_1 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(x(Y).P_1 \mid x\langle B\rangle.Q_1) :: z{:}C}$$

$(\nu x)(x(Y).P_1 \mid x\langle B\rangle.Q_1) \to (\nu x)(P_1\{B_1/Y\} \mid Q_1)$ by reduction
$\llparenthesis(\nu x)(x(Y).P_1 \mid x\langle B\rangle.Q_1)\rrparenthesis = (\llparenthesis Q_1\rrparenthesis\{x[B]/x\})\{(\Lambda Y.\llparenthesis P_1\rrparenthesis)/x\}$
$= \llparenthesis Q_1\rrparenthesis\{(\Lambda Y.\llparenthesis P_1\rrparenthesis[B])/x\} \to_\beta \llparenthesis Q_1\rrparenthesis\{\llparenthesis P_1\rrparenthesis\{B_1/Y\}/x\}$ by definition
$\llparenthesis(\nu x)(P_1\{B_1/Y\} \mid Q_1)\rrparenthesis = \llparenthesis Q_1\rrparenthesis\{\llparenthesis P_1\rrparenthesis\{B_1/Y\}/x\}$

Case:

$$(\text{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash x\langle B\rangle.P_1 :: x{:}\exists Y.A \quad \Omega;\Gamma;\Delta_2, x{:}\exists Y.A \vdash x(Y).Q_1 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(x\langle B\rangle.P_1 \mid x(Y).Q_1) :: z{:}C}$$

$(\nu x)(x\langle B\rangle.P_1 \mid x(Y).Q_1) \to (\nu x)(P_1 \mid Q_1\{B/Y\})$ by reduction
$\llparenthesis(\nu x)(x\langle B\rangle.P_1 \mid x(Y).Q_1)\rrparenthesis = \mathsf{let}\ (Y,x) = \mathsf{pack}\ B\ \mathsf{with}\ \llparenthesis P_1\rrparenthesis\ \mathsf{in}\ \llparenthesis Q_1\rrparenthesis$ by def.
$(\mathsf{pack}\ B\ \mathsf{with}\ \llparenthesis P_1\rrparenthesis \llparenthesis Q_1\rrparenthesis \to_\beta \llparenthesis Q_1\rrparenthesis\{\llparenthesis P_1\rrparenthesis/x, B/Y\}$
$\llparenthesis(\nu x)(P_1 \mid Q_1\{B/Y\})\rrparenthesis = \llparenthesis Q_1\rrparenthesis\{B/Y\})\{\llparenthesis P_1\rrparenthesis/x\}$

□

Theorem 3.11 (Operational Soundness). *Let $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ and $\llparenthesis P\rrparenthesis \to M$, there exists Q such that $P \mapsto^* Q$ and $\llparenthesis Q\rrparenthesis =_\alpha M$.*

Proof. By induction on typing.

Case:

$$(\multimap\text{L})\ \frac{\Omega;\Gamma;\Delta_1 \vdash P_1 :: y{:}A \quad \Omega;\Gamma;\Delta_2, x{:}B \vdash P_2 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2, x{:}A \multimap B \vdash (\nu y)x\langle y\rangle.(P_1 \mid P_2) :: z{:}C}$$

$\llparenthesis(\nu y)x\langle y\rangle.(P_1 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{(x\ \llparenthesis P_1\rrparenthesis)/x\}$ with $\llparenthesis P_2\rrparenthesis\{(x\ \llparenthesis P_1\rrparenthesis)/x\} = M \to M'$
by assumption
Subcase: $M \to M'$ due to redex in $\llparenthesis P_1\rrparenthesis$
$\llparenthesis P_1\rrparenthesis \to M_0$ by assumption
$\exists Q_0$ such that $P_1 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis \equiv_\alpha M_0$ by i.h.
$(\nu y)x\langle y\rangle.(P_1 \mid P_2) \mapsto^* (\nu y)x\langle y\rangle.(Q_0 \mid P_2)$ by compatibility of $\mapsto$
$\llparenthesis(\nu y)x\langle y\rangle.(Q_0 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{(x\ \llparenthesis Q_0\rrparenthesis)/x\} = \llparenthesis P_2\rrparenthesis\{(x\ M_0)/x\}$
Subcase: $M \to M'$ due to redex in $\llparenthesis P_2\rrparenthesis$
$\llparenthesis P_2\rrparenthesis \to M_0$ by assumption
$\exists Q_0$ such that $P_2 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis = M_0$ by i.h
$(\nu y)x\langle y\rangle.(P_1 \mid P_2) \mapsto^* (\nu y)x\langle y\rangle.(P_1 \mid Q_0)$ by compatibility of $\mapsto$
$\llparenthesis(\nu y)x\langle y\rangle.(P_1 \mid Q_0)\rrparenthesis = \llparenthesis Q_0\rrparenthesis\{(x\ \llparenthesis P_1\rrparenthesis)/x\} = M_0\{x\ \llparenthesis P_1\rrparenthesis)/x\}$

Case:

$$(\text{copy})\ \frac{\Omega;\Gamma, u{:}A;\Delta, x{:}A \vdash P_1 :: z{:}C}{\Omega;\Gamma, u{:}A;\Delta \vdash (\nu x)u\langle x\rangle.P_1 :: z{:}C}$$

$\llparenthesis(\nu x)u\langle x\rangle.P_1\rrparenthesis = \llparenthesis P_1\rrparenthesis\{u/x\} = M \to M'$ by assumption
$\llparenthesis P_1\rrparenthesis \to M_0$ by inversion on $\to$
$\exists Q_0$ such that $P_1 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis =_\alpha M_0$ by i.h.
$(\nu x)u\langle x\rangle.P_1 \mapsto^* (\nu x)u\langle x\rangle.Q_0$ by compatibility
$\llparenthesis(\nu x)u\langle x\rangle.Q_0\rrparenthesis = \llparenthesis Q_0\rrparenthesis\{u/x\} = M_0\{u/x\}$

Case:

$$(\forall \mathsf{L})\ \frac{\Omega \vdash B\,\mathsf{type} \quad \Omega;\Gamma;\Delta, x{:}A\{B/X\} \vdash P_1 :: z{:}C}{\Omega;\Gamma;\Delta, x{:}\forall X.A \vdash x\langle B\rangle.P_1 :: z{:}C}$$

$\llparenthesis x\langle B\rangle.P_1\rrparenthesis = \llparenthesis P_1\rrparenthesis\{x[B]/x\}$ with $\llparenthesis P_1\rrparenthesis\{x[B]/x\} \longrightarrow M$ — by assumption
$\llparenthesis P_1\rrparenthesis \longrightarrow M_0$ — by inversion
$\exists Q_0$ such that $P_1 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis =_\alpha M_0$ — by i.h.
$x\langle B\rangle.P_1 \mapsto^* x\langle B\rangle.Q_0$ — by compatibility
$\llparenthesis x\langle B\rangle.Q_0\rrparenthesis = \llparenthesis Q_0\rrparenthesis\{x[B]/x\} = M_0\{x[B]/x\}$

Case:

$$(\mathsf{cut})\ \frac{\Omega;\Gamma;\Delta_1 \vdash P_1 :: x{:}A \quad \Omega;\Gamma;\Delta_2, x{:}A \vdash P_2 :: z{:}C}{\Omega;\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(P_1 \mid P_2) :: z{:}C}$$

$\llparenthesis (\nu x)(P_1 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{\llparenthesis P_1\rrparenthesis/x\}$ with $\llparenthesis P_2\rrparenthesis\{\llparenthesis P_1\rrparenthesis/x\} = M \longrightarrow M'$ — by assumption
Subcase: $M \longrightarrow M'$ due to redex in $\llparenthesis P_1\rrparenthesis$
$\llparenthesis P_1\rrparenthesis \longrightarrow M_0$ — by assumption
$\exists Q_0$ such that $P_1 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis =_\alpha M_0$ — by i.h.
$(\nu x)(P_1 \mid P_2) \mapsto^* (\nu x)(Q_0 \mid P_2)$ — by reduction
$\llparenthesis (\nu x)(Q_0 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{\llparenthesis Q_0\rrparenthesis/x\} = \llparenthesis P_2\rrparenthesis\{M_0/x\}$
Subcase: $M \longrightarrow M'$ due to redex in $\llparenthesis P_2\rrparenthesis$
$\llparenthesis P_2\rrparenthesis \longrightarrow M_0$ — by assumption
$\exists Q_0$ such that $P_2 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis = M_0$ — by i.h.
$(\nu x)(P_1 \mid P_2) \mapsto^* (\nu x)(Q_0 \mid P_2)$ — by compatibility
$\llparenthesis (\nu x)(P_1 \mid Q_0)\rrparenthesis = \llparenthesis Q_0\rrparenthesis\{\llparenthesis P_1\rrparenthesis/x\} = M_0\{\llparenthesis P_1\rrparenthesis/x\}$
Subcase: $M \longrightarrow M'$ where the redex arises due to the substitution of $\llparenthesis P_1\rrparenthesis$ for x
Subsubcase: Last rule of deriv. of P_2 is a left rule on x:
In all cases except !L, a top-level process reduction is exposed (viz. Theorem 3.9).
If last rule is !L, then either x does not occur in P_2 and we conclude by $\mapsto$.
Subsubcase: Last rule of deriv. of P_2 is not a left rule on x:
For rule (id) we have a process reduction immediately. In all other cases either there is no possible β-redex or we can conclude via compatibility of $\mapsto$.

Case:

$$(\mathsf{cut}^!)\ \frac{\Omega;\Gamma;\cdot \vdash P_1 :: x{:}A \quad \Omega;\Gamma, u{:}A;\Delta \vdash P_2 :: z{:}C}{\Omega;\Gamma;\Delta \vdash (\nu u)(!u(x).P_1 \mid P_2) :: z{:}C}$$

$\llparenthesis (\nu u)(!u(x).P_1 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{\llparenthesis P_1\rrparenthesis/u\}$ with $\llparenthesis P_2\rrparenthesis\{\llparenthesis P_1\rrparenthesis/u\} \longrightarrow M$ — by assumption
Subcase: $M \longrightarrow M'$ due to redex in $\llparenthesis P_1\rrparenthesis$
$\llparenthesis P_1\rrparenthesis \longrightarrow M_0$ — by assumption
$\exists Q_0$ such that $P_1 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis =_\alpha M_0$ — by i.h.
$(\nu u)(!u(x).P_1 \mid P_2) \mapsto^* (\nu u)(!u(x).Q_0 \mid P_2)$ — by compatibility
$\llparenthesis (\nu u)(!u(x).Q_0 \mid P_2)\rrparenthesis = \llparenthesis P_2\rrparenthesis\{\llparenthesis Q_0\rrparenthesis/u\} = \llparenthesis P_2\rrparenthesis\{M_0/u\}$
Subcase: $M \longrightarrow M'$ due to redex in $\llparenthesis P_2\rrparenthesis$
$\llparenthesis P_2\rrparenthesis \longrightarrow M_0$ — by assumption
$\exists Q_0$ such that $P_2 \mapsto^* Q_0$ and $\llparenthesis Q_0\rrparenthesis = M_0$ — by i.h.
$(\nu u)(!u(x).P_1 \mid P_2) \mapsto^* (\nu u)(!u(x).P_1 \mid Q_0)$ — by compatibility
$\llparenthesis (\nu u)(!u(x).P_1 \mid Q_0)\rrparenthesis = \llparenthesis Q_0\rrparenthesis\{\llparenthesis P_1\rrparenthesis/u\} = M_0\{\llparenthesis P_1\rrparenthesis/u\}$
Subcase: $M \longrightarrow M'$ where the redex arises due to the substitution of $\llparenthesis P_1\rrparenthesis$ for u
If last rule in deriv. of P_2 is copy then we have = terms in 0 process reductions.
Otherwise, the result follows by compatibility of $\mapsto$.

In all other cases the λ-term in the image of the translation does not reduce.

□

A.2 Proofs for § 3.3 – Inversion and Full Abstraction

The proofs below rely on the fact that all commuting conversions of linear logic are sound observational equivalences in the sense of $\approx_{\mathsf{L}}$.

Theorem 3.12 (Inverse).

- *If* $\Omega; \Gamma; \Delta \vdash M : A$ *then* $\Omega; \Gamma; \Delta \vdash \llparenthesis \llbracket M \rrbracket_z \rrparenthesis \cong M : A$
- *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then* $\Omega; \Gamma; \Delta \vdash \llbracket \llparenthesis P \rrparenthesis \rrbracket_z \approx_{\mathsf{L}} P :: z{:}A$

We prove (1) and (2) above separately.

Theorem A.1. *If* $\Omega; \Gamma; \Delta \vdash M : A$ *then* $\Omega; \Gamma; \Delta \vdash \llparenthesis \llbracket M \rrbracket_z \rrparenthesis \cong M : A$

Proof. By induction on the given typing derivation.

Case: Linear variable

$\llparenthesis \llbracket x \rrbracket_z \rrparenthesis = x \cong x$

Case: Unrestricted variable

$\llbracket u \rrbracket_z = (\nu x) u\langle x \rangle.[x \leftrightarrow z]$ by def.
$\llparenthesis (\nu x)(u\langle x \rangle.[x \leftrightarrow z]) \rrparenthesis = u \cong u$

Case: λ-abstraction

$\llbracket \lambda x.M \rrbracket_z = z(x).\llbracket M \rrbracket_z$ by def.
$\llparenthesis z(x).\llbracket M \rrbracket_z \rrparenthesis = \lambda x.\llparenthesis \llbracket M \rrbracket_z \rrparenthesis \cong \lambda x.M$ by i.h. and congruence

Case: Application

$\llbracket M\, N \rrbracket_z = (\nu x)(\llbracket M \rrbracket_x \mid (\nu y) x\langle y \rangle.(\llbracket N \rrbracket_y \mid [x \leftrightarrow z]))$ by def.
$\llparenthesis (\nu x)(\llbracket M \rrbracket_x \mid (\nu y) x\langle y \rangle.(\llbracket N \rrbracket_y \mid [x \leftrightarrow z])) \rrparenthesis = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\, \llparenthesis \llbracket N \rrbracket_y \rrparenthesis$ by def.
$\llparenthesis \llbracket M \rrbracket_x \rrparenthesis\, \llparenthesis \llbracket N \rrbracket_y \rrparenthesis \cong M\, N$ by i.h. and congruence

Case: Exponential

$\llbracket !M \rrbracket_z = !z(x).\llbracket M \rrbracket_x$ by def.
$\llparenthesis !z(x).\llbracket M \rrbracket_x \rrparenthesis = !\llparenthesis \llbracket M \rrbracket_x \rrparenthesis \cong \llparenthesis \llbracket !M \rrbracket_z \rrparenthesis$ by def, i.h. and congruence

Case: Exponential elim.

$\llbracket \mathsf{let}\ !u = M\ \mathsf{in}\ N \rrbracket_z = (\nu x)(\llbracket M \rrbracket_x \mid \llbracket N \rrbracket_z\{x/u\})$ by def.
$\llparenthesis (\nu x)(\llbracket M \rrbracket_x \mid \llbracket N \rrbracket_z\{x/u\}) \rrparenthesis = \mathsf{let}\ !u = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\ \mathsf{in}\ \llparenthesis \llbracket N \rrbracket_z \rrparenthesis$ by def.
$\mathsf{let}\ !u = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\ \mathsf{in}\ \llparenthesis \llbracket N \rrbracket_z \rrparenthesis \cong \mathsf{let}\ !u = M\ \mathsf{in}\ N$ by congruence and i.h.

Case: Multiplicative Pairing

$\llbracket \langle M \otimes N \rangle \rrbracket_z = (\nu y) z\langle y \rangle.(\llbracket M \rrbracket_y \mid \llbracket N \rrbracket_z)$ by def.
$\llparenthesis (\nu y) z\langle y \rangle.(\llbracket M \rrbracket_y \mid \llbracket N \rrbracket_z) \rrparenthesis = \langle \llparenthesis \llbracket M \rrbracket_y \rrparenthesis \otimes \llparenthesis \llbracket N \rrbracket_z \rrparenthesis \rangle$ by def.
$\langle \llparenthesis \llbracket M \rrbracket_y \rrparenthesis \otimes \llparenthesis \llbracket N \rrbracket_z \rrparenthesis \rangle \cong \langle M \otimes N \rangle$ by i.h. and congruence

Case: Mult. Pairing Elimination

$\llbracket \mathsf{let}\ x \otimes y = M\ \mathsf{in}\ N \rrbracket_z = (\nu y)(\llbracket M \rrbracket_x \mid x(y).\llbracket N \rrbracket_z)$ by def.
$\llparenthesis (\nu y)(\llbracket M \rrbracket_x \mid x(y).\llbracket N \rrbracket_z) \rrparenthesis = \mathsf{let}\ x \otimes y = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\ \mathsf{in}\ \llparenthesis \llbracket N \rrbracket_z \rrparenthesis$ by def.
$\mathsf{let}\ x \otimes y = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\ \mathsf{in}\ \llparenthesis \llbracket N \rrbracket_z \rrparenthesis \cong \mathsf{let}\ x \otimes y = M\ \mathsf{in}\ N$ by i.h. and congruence

Case: Λ-abstraction
$\llparenthesis \llbracket \Lambda X.M \rrbracket_z \rrparenthesis = \Lambda X.\llparenthesis \llbracket M \rrbracket_z \rrparenthesis \cong \Lambda X.M$ by i.h. and congruence

Case: Type application
$\llparenthesis \llbracket M[A] \rrbracket_z \rrparenthesis = \llparenthesis \llbracket M \rrbracket_z \rrparenthesis [A] \cong M[A]$ by i.h. and congruence

Case: Existential Intro.
$\llparenthesis \llbracket \mathsf{pack}\ A\ \mathsf{with}\ M \rrbracket_z \rrparenthesis = \mathsf{pack}\ A\ \mathsf{with}\ \llparenthesis \llbracket M \rrbracket_z \rrparenthesis \cong \mathsf{pack}\ A\ \mathsf{with}\ M$ by i.h. and congruence

Case: Existential Elim.
$\llparenthesis \llbracket \mathsf{let}\ (X, y) = M\ \mathsf{in}\ N \rrbracket_z \rrparenthesis = \mathsf{let}\ (X, y) = \llparenthesis \llbracket M \rrbracket_x \rrparenthesis\ \mathsf{in}\ \llparenthesis \llbracket N \rrbracket_z \rrparenthesis \cong \mathsf{let}\ (X, y) = M\ \mathsf{in}\ N$
by i.h. and congruence

□

Theorem A.2. *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then* $\Omega; \Gamma; \Delta \vdash \llbracket \llparenthesis P \rrparenthesis \rrbracket_z \approx_{\mathsf{L}} P :: z{:}A$

Proof. By induction on the given typing derivation.

Case: (id) or any right rule
Immediate by definition in the case of (id) and by i.h. and congruence in all other cases.

Case: $\multimap\mathsf{L}$
$\llparenthesis (\nu y)x\langle y\rangle.(P \mid Q) \rrparenthesis = \llparenthesis Q \rrparenthesis\{(x\ \llparenthesis P \rrparenthesis)/x\}$ by def.
$\llbracket \llparenthesis Q \rrparenthesis\{(x\ \llparenthesis P \rrparenthesis))/x\} \rrbracket_z \approx_{\mathsf{L}} (\nu a)(\llbracket (x\ \llparenthesis P \rrparenthesis) \rrbracket_a \mid \llbracket \llparenthesis Q \rrparenthesis \rrbracket_z\{a/x\})$ by Lemma 3.4, with a fresh
$= (\nu a)((\nu w)([x \leftrightarrow w] \mid (\nu y)w\langle y\rangle.(\llbracket \llparenthesis P \rrparenthesis \rrbracket_y \mid [w \leftrightarrow a])) \mid \llbracket \llparenthesis Q \rrparenthesis \rrbracket_z\{a/x\})$ by def.
$\rightarrow (\nu a)((\nu y)x\langle y\rangle.(\llbracket \llparenthesis P \rrparenthesis \rrbracket_y \mid [x \leftrightarrow a]) \mid \llbracket \llparenthesis Q \rrparenthesis \rrbracket_z\{a/x\})$ by reduction
$\approx_{\mathsf{L}} (\nu y)x\langle y\rangle.(\llbracket \llparenthesis P \rrparenthesis \rrbracket_y \mid \llbracket \llparenthesis Q \rrparenthesis \rrbracket_z)$ commuting conversion + reduction
$\approx_{\mathsf{L}} (\nu y)x\langle y\rangle.(P \mid Q)$ by i.h. + congruence

Case: $\otimes\mathsf{L}$
$\llparenthesis x(y).P \rrparenthesis = \mathsf{let}\ x \otimes y = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$ by def.
$\llbracket \mathsf{let}\ x \otimes y = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis \rrbracket_z = (\nu w)([x \leftrightarrow w] \mid w(y).\llbracket \llparenthesis P \rrparenthesis \rrbracket_z)$ by def.
$\rightarrow x(y).\llbracket \llparenthesis P \rrparenthesis \rrbracket_z \approx_{\mathsf{L}} x(y).P$ by i.h. and congruence

Case: $!\mathsf{L}$
$\llparenthesis P\{x/u\} \rrparenthesis = \mathsf{let}\ !u = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis$ by def.
$\llbracket \mathsf{let}\ !u = x\ \mathsf{in}\ \llparenthesis P \rrparenthesis \rrbracket_z = (\nu w)([x \leftrightarrow w] \mid \llbracket \llparenthesis P \rrparenthesis \rrbracket_z\{w/u\})$ by def.
$\rightarrow \llbracket \llparenthesis P \rrparenthesis \rrbracket_z\{x/u\} \approx_{\mathsf{L}} P\{x/u\}$ by i.h.

Case: copy
$\llparenthesis (\nu x)u\langle x\rangle.P \rrparenthesis = \llparenthesis P \rrparenthesis\{u/x\}$ by def.
$\llbracket \llparenthesis P \rrparenthesis\{u/x\} \rrbracket_z \approx_{\mathsf{L}} (\nu x)(\overline{u}\langle w\rangle.[w \leftrightarrow x] \mid \llbracket \llparenthesis P \rrparenthesis \rrbracket_z)$ by Lemma 3.4
$\approx_{\mathsf{L}} (\nu x)(\overline{u}\langle w\rangle.[w \leftrightarrow x] \mid P)$ by i.h. and congruence
$\approx_{\mathsf{L}} (\nu x)u\langle x\rangle.P$ by definition of $\approx_{\mathsf{L}}$ for open processes
(i.e. closing for $u{:}A$ and observing that no actions on z are blocked)

Case: $\forall$L

$(\!|x\langle B\rangle.P|\!) = (\!|P|\!)\{(x[B])/x\}$ by def.
$[\![(\!|P|\!)\{(x[B])/x\}]\!]_z \approx_{\mathsf{L}} (\nu a)([\![x[B]]\!]_a \mid [\![(\!|P|\!)]\!]_z\{a/x\})$ by Lemma 3.4, with a fresh
$(\nu a)((\nu w)([x \leftrightarrow w] \mid w\langle B\rangle.[w \leftrightarrow a]) \mid [\![(\!|P|\!)]\!]_z\{a/x\})$ by def.
$\longrightarrow (\nu a)(x\langle B\rangle.[x \leftrightarrow a] \mid [\![(\!|P|\!)]\!]_z\{a/x\})$
$\approx_{\mathsf{L}} x\langle B\rangle.[\![(\!|P|\!)]\!]_z$ commuting conversion + reduction
$\approx_{\mathsf{L}} x\langle B\rangle.P$ by i.h. + congruence

Case: $\exists$L

$(\!|x(Y).P|\!) = \mathsf{let}\ (Y,x) = x\ \mathsf{in}\ (\!|P|\!)$ by def.
$[\![\mathsf{let}\ (Y,x) = x\ \mathsf{in}\ (\!|P|\!)]\!]_z = (\nu y)([x \leftrightarrow y] \mid y(Y).[\![(\!|P|\!)]\!]_z)$ by def.
$\longrightarrow x(Y).[\![(\!|P|\!)]\!]_z\{y/x\})$ by reduction
$\approx_{\mathsf{L}} x(Y).P$ by i.h. + congruence

Case: cut

$(\!|(\nu x)(P \mid Q)|\!) = (\!|Q|\!)\{(\!|P|\!)/x\}$ by definition
$[\![(\!|Q|\!)\{(\!|P|\!)/x\}]\!]_z \approx_{\mathsf{L}} (\nu y)([\![(\!|P|\!)]\!]_y \mid [\![(\!|Q|\!)]\!]_z\{y/x\})$ by Lemma 3.4, with y fresh
$\equiv (\nu x)(P \mid Q)$ by i.h. + congruence and $\equiv_\alpha$

Case: $\mathsf{cut}^!$

$(\!|((\nu u)(!u(x).P \mid Q))|\!) = (\!|Q|\!)\{(\!|P|\!)/u\}$ by definition
$[\![(\!|Q|\!)\{(\!|P|\!)/u\}]\!]_z \approx_{\mathsf{L}} (\nu u)(!u(x).[\![(\!|P|\!)]\!]_x \mid [\![(\!|Q|\!)]\!]_z\{v/u\})$ by Lemma 3.4
$\approx_{\mathsf{L}} (\nu u)(!u(x).P \mid Q)$ by i.h. + congruence and $\equiv_\alpha$

□

A.3 Proofs for § 5 – Communicating Values

A.3.1 Proofs of Encoding from λ to Sess$\pi\lambda$.

Lemma 5.2 (Compositionality). *Let $\Psi, x{:}\tau \vdash M : \sigma$ and $\Psi \vdash N : \tau$. We have that $[\![M\{N/x\}]\!]_z \approx_{\mathsf{L}} (\nu x)([\![M]\!]_z \mid !x(y).[\![N]\!]_y)$*

Proof. By induction on the typing for M. We make use of the fact that $\approx_{\mathsf{L}}$ includes $\equiv_!$.

Case: $M = y$ with $y = x$

$[\![M\{N/x\}]\!]_z = [\![N]\!]_z$
$(\nu x)([\![M]\!]_z \mid !x(y).[\![N]\!]_y) = (\nu x)(\overline{x}\langle y\rangle.[y \leftrightarrow z] \mid !x(y).[\![N]\!]_y)$ by definition
$\longrightarrow^+ (\nu x)([\![N]\!]_z \mid !x(y).[\![N]\!]_y)$ by the reduction semantics
$\approx_{\mathsf{L}} [\![N]\!]_z$ by $\equiv_!$, since $x \notin \mathsf{fn}([\![N]\!]_z)$

Case: $M = y$ with $y \neq x$

$[\![M\{N/x\}]\!]_z = [\![y]\!]_z = \overline{y}\langle w\rangle.[w \leftrightarrow z]$
$(\nu x)([\![M]\!] \mid !x(y).[\![N]\!]_y) = (\nu x)(\overline{y}\langle w\rangle.[w \leftrightarrow z] \mid !x(y).[\![N]\!]_y)$ by definition
$\approx_{\mathsf{L}} \overline{y}\langle w\rangle.[w \leftrightarrow z]$ by $\equiv_!$

Case: $M = M_1\, M_2$

$[\![M_1\, M_2\{N/x\}]\!]_z = [\![M_1\{N/x\}\, M_2\{N/x\}]\!]_z =$
$(\nu y)([\![M_1\{N/x\}]\!]_y \mid \overline{y}\langle u\rangle.(!u(w).[\![M_2\{N/x\}]\!]_w \mid [y \leftrightarrow z])$ by definition
$(\nu x)([\![M_1\, M_2]\!]_z \mid !x(y).[\![N]\!]_y) = (\nu x)((\nu y)([\![M_1]\!]_y \mid \overline{y}\langle u\rangle.(!u(w).[\![M_2]\!]_w \mid [y \leftrightarrow z]) \mid !x(y).[\![N]\!]_y))$
by definition
$[\![M_1\{N/x\}]\!]_y \approx_{\mathsf{L}} (\nu x)([\![M_1]\!]_y \mid !x(a).[\![N]\!]_a)$ by i.h.
$[\![M_2\{N/x\}]\!]_w \approx_{\mathsf{L}} (\nu x)([\![M_2]\!]_w \mid !x(a).[\![N]\!]_a)$ by i.h.
$[\![M_1\, M_2\{N/x\}]\!]_z \approx_{\mathsf{L}} (\nu y)((\nu x)([\![M_1]\!]_y \mid !x(a).[\![N]\!]_a) \mid \overline{y}\langle u\rangle.(!u(w).[\![M_2\{N/x\}]\!]_w \mid [y \leftrightarrow z]))$

by congruence
$\approx_{\mathsf{L}} (\nu y)((\nu x)(\llbracket M_1 \rrbracket_y \mid !x(a).\llbracket N \rrbracket_a) \mid \overline{y}\langle u\rangle.(!u(w).(\nu x)(\llbracket M_2 \rrbracket_w \mid !x(a).\llbracket N \rrbracket_a) \mid [y \leftrightarrow z]))$
by congruence
$\approx_{\mathsf{L}} (\nu x)(\nu y)(\llbracket M_1 \rrbracket_y \mid \overline{y}\langle u\rangle.(!u(w).\llbracket M \rrbracket_w \mid [y \leftrightarrow z] \mid !x(a).\llbracket N \rrbracket_a))$ by $\equiv_!$

Case: $M = \lambda y{:}\tau_0.M'$

$\llbracket \lambda y{:}\tau_0.M'\{N/x\} \rrbracket_z = z(y).\llbracket M'\{N/x\} \rrbracket_z$
$(\nu x)(\llbracket M \rrbracket_z \mid !x(y).\llbracket N \rrbracket_y) = (\nu x)(z(y).\llbracket M' \rrbracket_z \mid !x(y).\llbracket N \rrbracket_y)$ by definition
$\llbracket M'\{N/x\} \rrbracket_z \approx_{\mathsf{L}} (\nu x)(\llbracket M \rrbracket_z \mid !x(w).\llbracket N \rrbracket_w)$ by i.h.
$\llbracket \lambda y{:}\tau_0.M'\{N/x\} \rrbracket_z \approx_{\mathsf{L}} z(y).(\nu x)(\llbracket M' \rrbracket_z \mid !x(w).\llbracket N \rrbracket_w)$ by congruence
$\approx_{\mathsf{L}} (\nu x)(z(y).\llbracket M' \rrbracket_z \mid !x(w).\llbracket N \rrbracket_w)$ by commuting conversion

□

Theorem 5.3 (Operational Soundness – $\llbracket - \rrbracket_z$).

(1) *If* $\Psi \vdash M : \tau$ *and* $\llbracket M \rrbracket_z \longrightarrow Q$ *then* $M \longrightarrow^+ N$ *such that* $\llbracket N \rrbracket_z \approx_{\mathsf{L}} Q$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $\llbracket P \rrbracket \longrightarrow Q$ *then* $P \longrightarrow^+ P'$ *such that* $\llbracket P' \rrbracket \approx_{\mathsf{L}} Q$

Proof. By induction on the given derivation and case analysis on the reduction step.

Case: $M = M_1\, M_2$ with $\llbracket M_1 \rrbracket_y \longrightarrow R$

$\llbracket M_1\, M_2 \rrbracket_z = (\nu y)(\llbracket M_1 \rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]))$ by definition
$\longrightarrow (\nu y)(R \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]))$ by reduction semantics
$M_1 \longrightarrow^+ M_1'$ with $\llbracket M_1' \rrbracket_y \approx_{\mathsf{L}} R$ by i.h.
$M_1\, M_2 \longrightarrow^+ M_1'\, M_2$ by the operational semantics
$\llbracket M_1'\, M_2 \rrbracket_z = (\nu y)(\llbracket M_1' \rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]))$ by definition
$\approx_{\mathsf{L}} (\nu y)(R \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]))$ by congruence

Case: $M = M_1\, M_2$ with $(\nu y)(\llbracket M_1 \rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z])) \longrightarrow (\nu y, x)(R \mid !x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z])$

$\llbracket M_1 \rrbracket_y \equiv (\nu \overline{a})(y(x).R_1 \mid R_2)$ by the reduction semantics, for some R_1, R_2 and $\overline{a}$
$\Psi \vdash M_1 : \tau_0 \longrightarrow \tau_1$ by inversion
Subcase: $M_1 = y$, for some $y \in \Psi$
Impossible reduction.
Subcase: $M_1 = \lambda x{:}\tau_0.M_1'$
$(\lambda x{:}\tau_0.M_1')\, M_2 \longrightarrow M_1'\{M_2/x\}$ by operational semantics
$\llbracket M_1'\{M_2/x\} \rrbracket_z \approx_{\mathsf{L}} (\nu x)(\llbracket M_1' \rrbracket_z \mid !x(w).\llbracket M_2 \rrbracket_w)$ by Lemma 5.2
$\llbracket (\lambda x{:}\tau_0.M_1')\, M_2 \rrbracket_z = (\nu y)(y(x).\llbracket M_1' \rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]))$ by definition
$R = \llbracket M_1' \rrbracket_y$ by inversion
$(\nu y, x)(R \mid !x(w).\llbracket M_2 \rrbracket_w \mid [y \leftrightarrow z]) \approx_{\mathsf{L}} (\nu x)(\llbracket M_1' \rrbracket_z \mid !x(w).\llbracket M_2 \rrbracket_w)$ by reduction closure
Subcase: $M_1 = N_1\, N_2$, for some N_1 and N_2
$\llbracket N_1\, N_2 \rrbracket_y = (\nu a)(\llbracket N_1 \rrbracket_a \mid \overline{a}\langle b\rangle.(!b(d).\llbracket N_2 \rrbracket_d \mid [a \leftrightarrow y]))$ by definition
Impossible reduction.

Case: $P = (\nu x)(x\langle M\rangle.P_1 \mid x(y).P_2)$

$\llbracket P \rrbracket = (\nu x)(\overline{x}\langle y\rangle.(!y(w).\llbracket M \rrbracket_w \mid \llbracket P_1 \rrbracket) \mid x(y).\llbracket P_2 \rrbracket)$ by definition
$\llbracket P \rrbracket \longrightarrow (\nu x, y)(!y(w).\llbracket M \rrbracket_w \mid \llbracket P_1 \rrbracket \mid \llbracket P_2 \rrbracket)$ by reduction semantics
$P \longrightarrow (\nu x)(P_1 \mid P_2\{M/y\})$ by reduction semantics
$\llbracket (\nu x)(P_1 \mid P_2\{M/y\}) \rrbracket \approx_{\mathsf{L}} (\nu x, y)(\llbracket P_1 \rrbracket \mid \llbracket P_2 \rrbracket \mid !y(w).\llbracket M \rrbracket_w)$ by Lemma 5.2 and congruence

Case: $P = (\nu x)(x\langle M\rangle.P_1 \mid P_2)$

$[\![P]\!] = (\nu x)(\overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P_1]\!]) \mid [\![P_2]\!])$ by definition
$[\![P]\!] \longrightarrow (\nu x)(\overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P_1]\!]) \mid R)$ assumption, with $[\![P_2]\!] \longrightarrow R$
$P_2 \longrightarrow^+ P_2'$ with $[\![P_2']\!] \approx_{\mathsf{L}} R$ by i.h.
$P \longrightarrow^+ (\nu x)(x\langle M\rangle.P_1 \mid P_2')$ by reduction semantics
$[\![(\nu x)(x\langle M\rangle.P_1 \mid P_2')]\!] = (\nu x)(\overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P_1]\!]) \mid [\![P_2']\!])$ by definition
$\approx_{\mathsf{L}} (\nu x)(\overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P_1]\!]) \mid R)$ by congruence
All other process reductions follow straightforwardly from the inductive hypothesis.

□

Theorem 5.4 (Operational Completeness – $[\![-]\!]_z$).
(1) *If* $\Psi \vdash M : \tau$ *and* $M \longrightarrow N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $P \approx_{\mathsf{L}} [\![N]\!]_z$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $[\![P]\!] \longrightarrow^+ R$ *with* $R \approx_{\mathsf{L}} [\![Q]\!]$

Proof. We proceed by induction on the given derivation and case analysis on the reduction.

Case: $M = (\lambda x{:}\tau.M')\, N'$ with $M \longrightarrow M'\{N'/x\}$

$[\![M]\!]_z = (\nu y)([\![\lambda x{:}\tau.M']\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![N']\!]_w \mid [y \leftrightarrow z]) =$
$(\nu y)(y(x).[\![M']\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![N']\!]_w \mid [y \leftrightarrow z])$ by definition of $[\![-]\!]$
$\longrightarrow^+ (\nu x)([\![M']\!]_z \mid !x(w).[\![N']\!]_w)$ by the reduction semantics
$\approx_{\mathsf{L}} [\![M'\{N'/x\}]\!]_z$ by Lemma 5.2

Case: $M = M_1\, M_2$ with $M \longrightarrow M_1'\, M_2$ by $M_1 \longrightarrow M_1'$

$[\![M_1\, M_2]\!]_z = (\nu y)([\![M_1]\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![M_2]\!]_w \mid [y \leftrightarrow z])$ by definition
$[\![M_1'\, M_2]\!]_z = (\nu y)([\![M_1']\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![M_2]\!]_w \mid [y \leftrightarrow z])$ by definition
$[\![M_1]\!]_y \Longrightarrow P_1'$ such that $P_1' \approx_{\mathsf{L}} [\![M_1']\!]_y$ by i.h.
$[\![M_1\, M_2]\!]_z \Longrightarrow (\nu y)(P_1' \mid \overline{y}\langle x\rangle.(!x(w).[\![M_2]\!]_w \mid [y \leftrightarrow z])$ by reduction semantics
$\approx_{\mathsf{L}} (\nu y)([\![M_1']\!]_y \mid \overline{y}\langle x\rangle.(!x(w).[\![M_2]\!]_w \mid [y \leftrightarrow z])$ by congruence

Case: $P = (\nu x)(x\langle M\rangle.P' \mid x(y).Q')$ with $P \longrightarrow (\nu x)(P' \mid Q'\{M/y\})$

$[\![P]\!] = (\nu x)(\overline{x}\langle y\rangle.(!y(w).[\![M]\!]_w \mid [\![P']\!]) \mid x(y).[\![Q']\!])$ by definition
$[\![P]\!] \longrightarrow (\nu x, y)(!y(w).[\![M]\!]_w \mid [\![P']\!] \mid [\![Q']\!])$ by the reduction semantics
$[\![(\nu x)(P' \mid Q'\{M/y\})]\!] = (\nu x)([\![P']\!] \mid [\![Q'\{M/y\}]\!])$ by definition
$\approx_{\mathsf{L}} (\nu x, y)([\![P']\!] \mid [\![Q']\!] \mid !y(w).[\![M]\!]_w)$ by Lemma 5.2 and congruence

All remaining cases follow straightforwardly by induction.

□

A.3.2 Proofs of Encoding from Sess$\pi\lambda$to λ.

Theorem 5.7 (Operational Soundness – $(\!|-|\!)$).
(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $(\!|P|\!) \longrightarrow M$ *then* $P \mapsto^* Q$ *such that* $M =_\alpha (\!|Q|\!)$
(2) *If* $\Psi \vdash M : \tau$ *and* $(\!|M|\!) \longrightarrow N$ *then* $M \longrightarrow_\beta^+ M'$ *such that* $N =_\alpha (\!|M'|\!)$

Proof. We proceed by induction on the given reduction and case analysis on typing.

Case: $(\!|P_0|\!)\{(x\,!(\!|M_0|\!))/x\} \longrightarrow M$

$(\!|P_0|\!)\{(x\,!(\!|M_0|\!))/x\} \longrightarrow M'\{(x\,!(\!|M_0|\!))/x\}$ by operational semantics
$P_0 \mapsto P_0'$ with $P_0' =_\beta M'$ by i.h.
$x\langle M_0\rangle.P_0 \mapsto x\langle M_0\rangle.P_0'$ by extended reduction
$(\!|x\langle M_0\rangle.P_0'|\!) = (\!|P_0'|\!)\{(x\,!(\!|M_0|\!))/x\}$ by definition
$=_\alpha M'\{(x\,!(\!|M_0|\!))/x\}$ by congruence
The other cases are covered by our previous result for the reverse encoding of processes.

Case: $\llparenthesis M_0 \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis \longrightarrow M_0' \, !\llparenthesis M_1 \rrparenthesis$

$\llparenthesis M_0 \rrparenthesis \longrightarrow M_0'$ by inversion
$M_0 \longrightarrow_\beta^+ M_0''$ such that $M_0' =_\alpha \llparenthesis M_0'' \rrparenthesis$ by i.h.
$M_0\, M_1 \longrightarrow_\beta^+ M_0''\, M_1$ by operational semantics
$\llparenthesis M_0''\, M_1 \rrparenthesis = \llparenthesis M_0'' \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis =_\alpha M_0' \, !\llparenthesis M_1 \rrparenthesis$ by definition and by congruence

Case: $(\lambda x{:}!\llparenthesis \tau_0 \rrparenthesis.\mathsf{let}\, !x = x \,\mathsf{in}\, \llparenthesis M_0 \rrparenthesis) \, !\llparenthesis M_1 \rrparenthesis \longrightarrow \mathsf{let}\, !x = !\llparenthesis M_1 \rrparenthesis \,\mathsf{in}\, \llparenthesis M_0 \rrparenthesis$

$(\lambda x{:}\tau_0.M_0)\, M_1 \longrightarrow M_0\{M_1/x\}$ by inversion and operational semantics
$\mathsf{let}\, !x = !\llparenthesis M_1 \rrparenthesis \,\mathsf{in}\, \llparenthesis M_0 \rrparenthesis \longrightarrow \llparenthesis M_0 \rrparenthesis \{\llparenthesis M_1 \rrparenthesis / x\}$ by operational semantics
$=_\alpha \llparenthesis M_0\{M_1/x\} \rrparenthesis$ by Lemma 5.6

□

Theorem 5.8 (Operational Completeness – $\llparenthesis - \rrparenthesis$).
(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $\llparenthesis P \rrparenthesis \longrightarrow_\beta^* \llparenthesis Q \rrparenthesis$
(2) *If* $\Psi \vdash M : \tau$ *and* $M \longrightarrow N$ *then* $\llparenthesis M \rrparenthesis \longrightarrow^+ \llparenthesis N \rrparenthesis$.

Proof. We proceed by induction on the given reduction.

Case: $(\nu x)(x\langle M \rangle.P_1 \mid x(y).P_2) \longrightarrow (\nu x)(P_1 \mid P_2\{M/x\})$ with P typed via cut of $\wedge$R and $\wedge$L

$\llparenthesis P \rrparenthesis = \mathsf{let}\, y \otimes x = \langle !\llparenthesis M \rrparenthesis \otimes \llparenthesis P_1 \rrparenthesis \rangle \,\mathsf{in}\, \mathsf{let}\, !y = y \,\mathsf{in}\, \llparenthesis P_2 \rrparenthesis$ by definition
$\longrightarrow \mathsf{let}\, !y = !\llparenthesis M \rrparenthesis \,\mathsf{in}\, \llparenthesis P_2 \rrparenthesis \{\llparenthesis P_1 \rrparenthesis / x\}$ by operational semantics
$\longrightarrow \llparenthesis P_2 \rrparenthesis \{\llparenthesis P_1 \rrparenthesis / x\}\{\llparenthesis M \rrparenthesis / x\}$ by operational semantics
$\llparenthesis (\nu x)(P_1 \mid P_2\{M/x\}) \rrparenthesis = \llparenthesis P_2\{M/x\} \rrparenthesis \{\llparenthesis P_1 \rrparenthesis / x\}$ by definition
$=_\alpha \llparenthesis P_2 \rrparenthesis \{\llparenthesis P_1 \rrparenthesis / x\}\{\llparenthesis M \rrparenthesis / x\}$ by Lemma 5.6

Case: $(\nu x)(x(y).P_1 \mid x\langle M \rangle.P_2) \longrightarrow (\nu x)(P_1\{M/x\} \mid P_2)$ with P typed via cut of $\supset$R and $\supset$L

$\llparenthesis P \rrparenthesis = \llparenthesis P_2 \rrparenthesis \{(\lambda x{:}!\llparenthesis \tau_0 \rrparenthesis.\mathsf{let}\, !x = x \,\mathsf{in}\, \llparenthesis P_1 \rrparenthesis) \, !\llparenthesis M \rrparenthesis / x\}$ by definition
$\longrightarrow_\beta^+ \llparenthesis P_2 \rrparenthesis \{(\llparenthesis P_1 \rrparenthesis \{\llparenthesis M \rrparenthesis / x\}) / x\}$ by β conversion
$\llparenthesis (\nu x)(P_1\{M/x\} \mid P_2) \rrparenthesis = \llparenthesis P_2 \rrparenthesis \{\llparenthesis P_1\{M/x\} \rrparenthesis / x\}$ by definition
$=_\alpha \llparenthesis P_2 \rrparenthesis \{(\llparenthesis P_1 \rrparenthesis \{\llparenthesis M \rrparenthesis / x\}) / x\}$ by Lemma 5.6

The remaining process cases follow by induction.

Case: $(\lambda x{:}\tau_0.M_0)\, M_1 \longrightarrow M_0\{M_1/x\}$

$\llparenthesis M \rrparenthesis = (\lambda x{:}!\llparenthesis \tau_0 \rrparenthesis.\mathsf{let}\, !x = x \,\mathsf{in}\, \llparenthesis M_0 \rrparenthesis) \, !\llparenthesis M_1 \rrparenthesis$ by definition
$\longrightarrow^+ \llparenthesis M_0 \rrparenthesis \{\llparenthesis M_1 \rrparenthesis / x\} =_\alpha \llparenthesis M_0\{M_1/x\} \rrparenthesis$ by operational semantics and Lemma 5.6

Case: $M_0\, M_1 \longrightarrow M_0'\, M_1$ by $M_0 \longrightarrow M_0'$

$\llparenthesis M_0\, M_1 \rrparenthesis = \llparenthesis M_0 \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis$ by definition
$\llparenthesis M_0'\, M_1 \rrparenthesis = \llparenthesis M_0' \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis$ by definition
$\llparenthesis M_0 \longrightarrow^+ \llparenthesis M_0' \rrparenthesis$ by i.h.
$\llparenthesis M_0 \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis \longrightarrow^+ \llparenthesis M_0' \rrparenthesis \, !\llparenthesis M_1 \rrparenthesis$ by operational semantics

□

A.3.3 Proofs of Inverse Theorem and Full Abstraction in Sess$\pi\lambda$.

Theorem 5.9 (Inverse). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![\llparenthesis P \rrparenthesis]\!]_z \approx_{\mathsf{L}} [\![P]\!]$. *If* $\Psi \vdash M : \tau$ *then* $\llparenthesis [\![M]\!]_z \rrparenthesis =_\beta \llparenthesis M \rrparenthesis$.

We establish the proofs of the two statements separately:

Theorem A.3 (Inverse – Processes). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![\llparenthesis P \rrparenthesis]\!]_z \approx_{\mathsf{L}} [\![P]\!]$

Proof. By induction on typing.

Case: $\wedge$R

$P = z\langle M\rangle.P_0$ by assumption
$\llparenthesis P\rrparenthesis = \langle !\llparenthesis M\rrparenthesis \otimes \llparenthesis P_0\rrparenthesis\rangle$ by definition
$\llbracket \langle !\llparenthesis M\rrparenthesis \otimes \llparenthesis P_0\rrparenthesis\rangle \rrbracket_z = \overline{z}\langle x\rangle.(!x(u).\llbracket \llparenthesis M\rrparenthesis\rrbracket_u \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z)$ by definition
$\llbracket z\langle M\rangle.P_0\rrbracket = \overline{z}\langle x\rangle.(!x(u).\llbracket M\rrbracket_u \mid \llbracket P_0\rrbracket)$ by definition
$\approx_{\mathsf{L}} \overline{z}\langle x\rangle.(!x(u).\llbracket \llparenthesis M\rrparenthesis\rrbracket_u \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z)$ by i.h. and congruence

Case: $\wedge$L

$P = x(y).P_0$ by assumption
$\llparenthesis P\rrparenthesis = \mathsf{let}\ y \otimes x = x\ \mathsf{in}\ \mathsf{let}\ !y = y\ \mathsf{in}\ \llparenthesis P_0\rrparenthesis$ by definition
$\llbracket \mathsf{let}\ y \otimes x = x\ \mathsf{in}\ \mathsf{let}\ !y = y\ \mathsf{in}\ \llparenthesis P_0\rrparenthesis\rrbracket_z = x(y).\llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z$ by definition
$\llbracket x(y).P_0\rrbracket = x(y).\llbracket P_0\rrbracket$ by definition
$\approx_{\mathsf{L}} x(y).\llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z$ by i.h. and congruence

Case: $\supset$R

$P = x(y).P_0$ by assumption
$\llparenthesis P\rrparenthesis = \lambda x{:}!\llparenthesis \tau\rrparenthesis.\mathsf{let}\ !x = x\ \mathsf{in}\ \llparenthesis P_0\rrparenthesis$ by definition
$\llbracket \lambda x{:}!\llparenthesis \tau\rrparenthesis.\mathsf{let}\ !x = x\ \mathsf{in}\ \llparenthesis P_0\rrparenthesis\rrbracket_z = x(y).\llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z$ by definition
$\llbracket x(y).P_0\rrbracket = x(y).\llbracket P_0\rrbracket$ by definition
$\approx_{\mathsf{L}} x(y).\llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z$ by i.h. and congruence

Case: $\supset$L

$P = x\langle M\rangle.P_0$ by assumption
$\llparenthesis P\rrparenthesis = \llparenthesis P_0\rrparenthesis\{(x\,!\llparenthesis M\rrparenthesis)/x\}$ by definition
$\llbracket \llparenthesis P_0\rrparenthesis\{(x\,!\llparenthesis M\rrparenthesis)/x\}\rrbracket_z = (\nu a)(\llbracket x\,!\llparenthesis M\rrparenthesis\rrbracket_a \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z\{a/x\})$ by Lemma 3.4
$= (\nu a)((\nu b)(\llbracket x\rrbracket_b \mid \overline{b}\langle c\rangle.(\llbracket !\llparenthesis M\rrparenthesis\rrbracket_c \mid [b \leftrightarrow a]) \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z\{a/x\})$ by definition
$= (\nu a)((\nu b)([x \leftrightarrow b] \mid \overline{b}\langle c\rangle.(!c(w).\llbracket \llparenthesis M\rrparenthesis\rrbracket_w \mid [b \leftrightarrow a]) \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z\{a/x\}))$ by definition
$\longrightarrow (\nu a)(\overline{x}\langle c\rangle.(!c(w).\llbracket \llparenthesis M\rrparenthesis\rrbracket_w \mid [x \leftrightarrow a]) \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z\{a/x\})$ by reduction semantics
$\approx_{\mathsf{L}} \overline{x}\langle c\rangle.(!c(w).\llbracket \llparenthesis M\rrparenthesis\rrbracket_w \mid \llbracket \llparenthesis P_0\rrparenthesis\rrbracket_z)$ by commuting conversion and reduction
$\approx_{\mathsf{L}} \llbracket P\rrbracket = \overline{x}\langle y\rangle.(!y(u).\llbracket M\rrbracket_u \mid \llbracket P_0\rrbracket)$ by i.h. and congruence

□

Theorem A.4 (Inverse Encodings – λ-terms). *If $\Psi \vdash M : \tau$ then $\llparenthesis \llbracket M\rrbracket_z\rrparenthesis =_\beta \llparenthesis M\rrparenthesis$*

Proof. By induction on typing.

Case: Variable

$\llbracket M\rrbracket_z = \overline{x}\langle y\rangle.[y \leftrightarrow z]$ by definition
$\llparenthesis \overline{x}\langle y\rangle.[y \leftrightarrow z]\rrparenthesis = x$ by definition

Case: λ-abstraction

$\llbracket \lambda x{:}\tau_0.M_0\rrbracket_z = z(x).\llbracket M_0\rrbracket_z$ by definition
$\llparenthesis z(x).\llbracket M_0\rrbracket_z\rrparenthesis = \lambda x{:}!\llparenthesis \tau_0\rrparenthesis.\mathsf{let}\ !x = x\ \mathsf{in}\ \llparenthesis \llbracket M_0\rrbracket_z\rrparenthesis$ by definition
$=_\beta \llparenthesis \lambda x{:}\tau_0.M_0\rrparenthesis = \lambda x{:}!\llparenthesis \tau_0\rrparenthesis.\mathsf{let}\ !x = x\ \mathsf{in}\ \llparenthesis M_0\rrparenthesis$ by i.h. and congruence

Case: Application

$\llbracket M_0\, M_1\rrbracket_z = (\nu y)(\llbracket M_0\rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_1\rrbracket_w \mid [y \leftrightarrow z])$ by definition
$\llparenthesis (\nu y)(\llbracket M_0\rrbracket_y \mid \overline{y}\langle x\rangle.(!x(w).\llbracket M_1\rrbracket_w \mid [y \leftrightarrow z])\rrparenthesis = \llparenthesis \overline{y}\langle x\rangle.(!x(w).\llbracket M_1\rrbracket_w \mid [y \leftrightarrow z])\rrparenthesis\{\llparenthesis \llbracket M_0\rrbracket_y\rrparenthesis/y\}$
by definition
$= \llparenthesis \llbracket M_0\rrbracket_y\rrparenthesis\ !\llparenthesis \llbracket M_1\rrbracket_w\rrparenthesis$ by definition

$=_\beta \llparenthesis M_0\, M_1 \rrparenthesis = \llparenthesis M_0 \rrparenthesis\, !\llparenthesis M_1 \rrparenthesis$ by i.h. and congruence

□

Lemma 5.10. *Let* $\cdot \vdash M : \tau$ *and* $\cdot \vdash V : \tau$ *with* $V \not\to$. $\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket V \rrbracket_z$ *iff* $\llparenthesis M \rrparenthesis \to^*_{\beta\eta} \llparenthesis V \rrparenthesis$

Proof.

$(\Leftarrow)$

$\llparenthesis M \rrparenthesis \to^*_{\beta\eta} \llparenthesis V \rrparenthesis$ by assumption

If $\llparenthesis M \rrparenthesis = \llparenthesis V \rrparenthesis$ then $\llbracket V \rrbracket_z \approx_{\mathsf{L}} \llbracket V \rrbracket_z$ by reflexivity

If $\llparenthesis M \rrparenthesis \to^+_{\beta\eta} \llparenthesis V \rrparenthesis$ then $\llbracket M \rrbracket_z \Longrightarrow P \approx_{\mathsf{L}} \llbracket V \rrbracket_z$ by Lemma 5.4

$\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket V \rrbracket_z$ by closure under reduction

$(\Rightarrow)$

$V =_\alpha \lambda x{:}\tau_0.V_0$ by inversion

$\llparenthesis V \rrparenthesis = \lambda x{:}!\llparenthesis \tau_0 \rrparenthesis.\mathsf{let}\ !x = x \mathsf{\ in\ } \llparenthesis V_0 \rrparenthesis$ by definition

$\llbracket V \rrbracket_z = z(x).\llbracket V_0 \rrbracket_z$ by definition

$M : \tau_0 \to \tau_1$ by inversion

$\llparenthesis M \rrparenthesis \to^*_{\beta\eta} V' \not\to$ by strong normalisation

We proceed by induction on the length n of the (strong) reduction:

Subcase: $n = 0$

$\llparenthesis M \rrparenthesis = \lambda x{:}\tau_0.M_0$ by inversion

$M_0 = V_0$ by uniqueness of normal forms

Subcase: $n = n' + 1$

$\llparenthesis M \rrparenthesis \to_{\beta\eta} M'$ by assumption

$\llbracket M \rrbracket_z \Longrightarrow P \approx_{\mathsf{L}} \llbracket M' \rrbracket_z$ by Lemma 5.4

$\llbracket M' \rrbracket_z \approx_{\mathsf{L}} \llbracket V \rrbracket_z$ by closure under reduction

$\llparenthesis M' \rrparenthesis \to^*_{\beta\eta} \llparenthesis V \rrparenthesis$ by i.h.

$\llparenthesis M \rrparenthesis \to^*_{\beta\eta} \llparenthesis V \rrparenthesis$ by transitive closure

□

Theorem 5.11 (Full Abstraction).
Let:

(a) $\cdot \vdash M : \tau$ *and* $\cdot \vdash N : \tau$;
(b) $\cdot \vdash P :: z{:}A$ *and* $\cdot \vdash Q :: z{:}A$.

We have that $\llparenthesis M \rrparenthesis =_{\beta\eta} \llparenthesis N \rrparenthesis$ *iff* $\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket N \rrbracket_z$ *and* $\llbracket P \rrbracket \approx_{\mathsf{L}} \llbracket Q \rrbracket$ *iff* $\llparenthesis P \rrparenthesis =_{\beta\eta} \llparenthesis Q \rrparenthesis$.

We establish the proof of the two statements separately.

Theorem A.5. *Let* $\cdot \vdash M : \tau$ *and* $\cdot \vdash N : \tau$. *We have that* $\llparenthesis M \rrparenthesis =_{\beta\eta} \llparenthesis N \rrparenthesis$ *iff* $\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket N \rrbracket_z$

Proof.

Completeness $(\Rightarrow)$

$\llparenthesis M \rrparenthesis =_{\beta\eta} \llparenthesis N \rrparenthesis$ iff $\exists S.\llparenthesis M \rrparenthesis \to^*_{\beta\eta} S$ and $\llparenthesis N \rrparenthesis \to^*_{\beta\eta} S$

Assume $\to^*$ is of length 0, then: $\llparenthesis M \rrparenthesis =_\alpha \llparenthesis N \rrparenthesis$, $\llbracket M \rrbracket_z \equiv \llbracket N \rrbracket_z$ and thus $\llbracket M \rrbracket \approx_{\mathsf{L}} \llbracket N \rrbracket_z$

Assume $\to^+$ is of some length > 0:

$\llparenthesis M \rrparenthesis \to^+_{\beta\eta} S$ and $\llparenthesis N \rrparenthesis \to^+_{\beta\eta} S$, for some S by assumption

$\llbracket M \rrbracket_z \to^+ P \approx_{\mathsf{L}} \llbracket S \rrbracket_z$ and $\llbracket N \rrbracket_z \to^+ Q \approx_{\mathsf{L}} \llbracket S \rrbracket_z$ by Theorem 5.4

$\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket S \rrbracket_z$ and $\llbracket N \rrbracket_z \approx_{\mathsf{L}} \llbracket S \rrbracket_z$ by closure under reduction

$\llbracket M \rrbracket_z \approx_{\mathsf{L}} \llbracket N \rrbracket_z$ by transitivity

Soundness $(\Leftarrow)$

$[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$ by assumption
Suffices to show: $\exists S.(\!|M|\!) \longrightarrow^*_{\beta\eta} S$ and $(\!|N|\!) \longrightarrow^*_{\beta\eta} S$
$(\!|N|\!) \longrightarrow^*_{\beta\eta} S' \not\longrightarrow$ by strong normalisation
We proceed by induction on the length n of the reduction:
Subcase: $n = 0$
$[\![M]\!]_z \approx_{\mathsf{L}} [\![S']\!]_z$ by assumption
$(\!|M|\!) \longrightarrow^*_{\beta\eta} (\!|N|\!)$ by Lemma 5.10
Subcase: $n = n' + 1$
$(\!|N|\!) \longrightarrow_{\beta\eta} S'$ by assumption
$[\![N]\!]_z \longrightarrow P \approx_{\mathsf{L}} [\![S']\!]_z$ by Theorem 5.4
$[\![M]\!]_z \approx_{\mathsf{L}} [\![S']\!]_z$ by closure under reduction
$(\!|M|\!) =_{\beta\eta} (\!|S'|\!)$ by i.h.
$(\!|M|\!) =_{\beta\eta} (\!|N|\!)$ by transitivity

□

Theorem A.6. *Let* $\cdot \vdash P :: z{:}A$ *and* $\cdot \vdash Q :: z{:}A$. *We have that* $[\![P]\!] \approx_{\mathsf{L}} [\![Q]\!]$ *iff* $(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$

Proof.

$(\Leftarrow)$
Let $M = (\!|P|\!)$ and $N = (\!|Q|\!)$:
$[\![M]\!]_z \approx_{\mathsf{L}} [\![N]\!]_z$ by Theorem A.5 $(\Rightarrow)$
$[\![M]\!]_z = [\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} [\![P]\!]$ and $[\![N]\!]_z = [\![(\!|Q|\!)]\!]_z \approx_{\mathsf{L}} [\![Q]\!]$ by Theorem 5.9
$[\![P]\!] \approx_{\mathsf{L}} [\![Q]\!]$ by compatibility of logical equivalence
$(\Rightarrow)$
$[\![(\!|P|\!)]\!]_z \approx_{\mathsf{L}} [\![(\!|Q|\!)]\!]_z$ by Theorem 3.12 and compatibility of logical equivalence
$(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$ by Theorem A.5 $(\Leftarrow)$

□

A.4 Proofs of § 5.2 – Higher-Order Session Processes

A.4.1 Proofs for Encoding of λ into Sess$\pi\lambda^+$.

Theorem 5.13 (Operational Soundness – $[\![-]\!]_z$).
(1) *If* $\Psi \vdash M : \tau$ *and* $[\![M]\!]_z \longrightarrow Q$ *then* $M \longrightarrow^+ N$ *such that* $[\![N]\!]_z \approx_{\mathsf{L}} Q$
(2) *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $[\![P]\!] \longrightarrow Q$ *then* $P \longrightarrow^+ P'$ *such that* $[\![P']\!] \approx_{\mathsf{L}} Q$

Proof. By induction on the given reduction.

Case: $(\nu x)(P_0 \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1)\ldots)) \longrightarrow (\nu x)(P_0' \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1)\ldots))$
$P = x \leftarrow M_0 \leftarrow \overline{y_i}; P_2$ with $[\![M_0]\!]_x = P_0$ and $[\![P_1]\!] = P_2$ by inversion
$M_0 \longrightarrow^+ M_0'$ with $[\![M_0']\!]_x \approx_{\mathsf{L}} P_0'$ by i.h.
$(x \leftarrow M_0 \leftarrow \overline{y_i}; P_2) \longrightarrow^+ (x \leftarrow M_0' \leftarrow \overline{y_i}; P_2)$ by reduction semantics
$[\![x \leftarrow M_0' \leftarrow \overline{y}; P_2]\!] = (\nu x)([\![M_0]\!]_x \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1)\ldots))$
by definition
$\approx_{\mathsf{L}} (\nu x)(P_0' \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1)$ by congruence

Case: $(\nu x)(x(a_0).\ldots.x(a_n).P_0 \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1) \longrightarrow (\nu x, a_0)(x(a_1).\ldots.x(a_n).P_0 \mid [a_0 \leftrightarrow y_0] \mid x\langle a_1\rangle.([a_1 \leftrightarrow y_1] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid P_1) = Q$

$P = x \leftarrow \{x \leftarrow P_2 \leftarrow \overline{a_i}\} \leftarrow \overline{y_i}; P_3$ with $\llbracket P_3 \rrbracket = P_1$ and $\llbracket P_2 \rrbracket = P_0$ — by inversion
$x \leftarrow \{x \leftarrow P_2 \leftarrow \overline{a_i}\} \leftarrow \overline{y_i}; P_3 \rightarrow (\nu x)(P_2\{\overline{y_i}/\overline{a_i}\} \mid P_3)$ — by reduction semantics
$Q \rightarrow^+ (\nu x)(P_0\{\overline{y_i}/\overline{a_i}\} \mid P_1) = (\nu x)(\llbracket P_2 \rrbracket\{\overline{y_i}/\overline{a_i}\} \mid \llbracket P_3 \rrbracket)$ by reduction semantics and definition

□

Theorem 5.14 (Operational Completeness – $\llbracket - \rrbracket_z$).

(1) *If* $\Psi \vdash M : \tau$ *and* $M \rightarrow N$ *then* $\llbracket M \rrbracket_z \Longrightarrow P$ *such that* $P \approx_{\mathsf{L}} \llbracket N \rrbracket_z$
(2) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \rightarrow Q$ *then* $\llbracket P \rrbracket \rightarrow^+ R$ *with* $R \approx_{\mathsf{L}} \llbracket Q \rrbracket$

Proof. By induction on the reduction semantics.

Case: $x \leftarrow M \leftarrow \overline{y_i}; Q \rightarrow x \leftarrow M' \leftarrow \overline{y_i}; Q$ from $M \rightarrow M'$

$\llbracket x \leftarrow M \leftarrow \overline{y_i}; Q \rrbracket = (\nu x)(\llbracket M \rrbracket_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid \llbracket Q \rrbracket) \dots))$
by definition
$\llbracket M \rrbracket_x \Longrightarrow R_0$ with $R_0 \approx_{\mathsf{L}} \llbracket M' \rrbracket_x$ — by i.h.
$\llbracket x \leftarrow M \leftarrow \overline{y_i}; Q \rrbracket \Longrightarrow (\nu x)(R_0 \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid \llbracket Q \rrbracket) \dots))$
by reduction semantics
$\approx_{\mathsf{L}} \llbracket x \leftarrow M \leftarrow \overline{y_i}; Q \rrbracket = (\nu x)(\llbracket M \rrbracket_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid \llbracket Q \rrbracket) \dots))$
by congruence

Case: $x \leftarrow \{x \leftarrow P_0 \leftarrow \overline{w_i}\} \leftarrow \overline{y_i}; Q \rightarrow (\nu x)(P_0\{\overline{y_i}/\overline{w_i}\} \mid Q)$

$\llbracket x \leftarrow \{x \leftarrow P_0 \leftarrow \overline{w_i}\} \leftarrow \overline{y_i}; Q \rrbracket =$
$(\nu x)(x(w_0).\dots.x(w_n).\llbracket P_0 \rrbracket \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid \llbracket Q \rrbracket) \dots))$
by definition
$\rightarrow^+ (\nu x)(\llbracket P_0 \rrbracket\{\overline{y_i}/\overline{w_i}\} \mid \llbracket Q \rrbracket)$ — by reduction semantics
$\approx_{\mathsf{L}} (\nu x)(\llbracket P_0\{\overline{y_i}/\overline{w_i}\} \rrbracket \mid \llbracket Q \rrbracket)$

□

A.4.2 Proofs for Encoding of Sess$\pi\lambda^+$ into λ.

Theorem 5.16 (Operational Soundness – $\llparenthesis - \rrparenthesis$).

(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $\llparenthesis P \rrparenthesis \rightarrow M$ *then* $P \mapsto^* Q$ *such that* $M =_\alpha \llparenthesis Q \rrparenthesis$
(2) *If* $\Psi \vdash M : \tau$ *and* $\llparenthesis M \rrparenthesis \rightarrow N$ *then* $M \rightarrow_\beta^+ M'$ *such that* $N =_\alpha \llparenthesis M' \rrparenthesis$

Proof. By induction on the given reduction.

Case: $\llparenthesis P_0 \rrparenthesis\{(\llparenthesis M \rrparenthesis\, \overline{y_i})/x\} \rightarrow N\{(\llparenthesis M \rrparenthesis\, \overline{y_i})/x\}$

$P = x \leftarrow M \leftarrow \overline{y_i}; P_0$ — by inversion
$P_0 \mapsto^* R$ with $N =_\alpha \llparenthesis R \rrparenthesis$ — by i.h.
$P \mapsto^* x \leftarrow M \leftarrow \overline{y_i}; R$ — by definition of $\mapsto$
$\llparenthesis x \leftarrow M \leftarrow \overline{y_i}; R \rrparenthesis = \llparenthesis R \rrparenthesis\{(\llparenthesis M \rrparenthesis\, \overline{y_i})/x\}$ — by definition
$=_\alpha N\{(\llparenthesis M \rrparenthesis\, \overline{y_i})/x\}$ — by congruence

Case: $\llparenthesis P_0 \rrparenthesis\{(\llparenthesis M \rrparenthesis\, \overline{y_i})/x\} \rightarrow \llparenthesis P_0 \rrparenthesis\{M'/x\}$

$P = x \leftarrow M \leftarrow \overline{y_i}; P_0$ — by inversion
Subcase: $\llparenthesis M \rrparenthesis\, \overline{y_i} \rightarrow N\, \overline{y_i}$
$M \rightarrow_\beta^+ M''$ with $N =_\alpha \llparenthesis M'' \rrparenthesis$ — by i.h.
$P \mapsto^+ x \leftarrow M'' \leftarrow \overline{y_i}; P_0$ — by reduction semantics
$\llparenthesis x \leftarrow M'' \leftarrow \overline{y_i}; P_0 \rrparenthesis = \llparenthesis P_0 \rrparenthesis\{(\llparenthesis M'' \rrparenthesis\, \overline{y_i})/x\}$ — by definition
$=_\alpha \llparenthesis P_0 \rrparenthesis\{M'/x\}$ — by congruence
Subcase: $\llparenthesis M \rrparenthesis\, \overline{y_i} \rightarrow (\lambda y_1.\dots.y_n.M_0)\, y_1\, \dots\, y_n$

$M = \{x \leftarrow Q \leftarrow \overline{y_i}\}$ with $\llparenthesis Q \rrparenthesis = M_0$ — by inversion
$P = x \leftarrow \{x \leftarrow Q \leftarrow \overline{y_i}\} \leftarrow \overline{y_i}; P_0$ — by inversion
$P \longrightarrow (\nu x)(Q \mid P_0)$ — by reduction semantics
$\llparenthesis (\nu x)(Q \mid P_0) \rrparenthesis = \llparenthesis P_0 \rrparenthesis \{\llparenthesis Q \rrparenthesis / x\}$ — by definition
$(\lambda y_1. \dots .y_n.M_0)\, y_1 \ \dots \ y_n \longrightarrow^+ M_0$ — by operational semantics

□

Theorem 5.17 (Operational Completeness – $\llparenthesis - \rrparenthesis$).
(1) *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $\llparenthesis P \rrparenthesis \longrightarrow^*_\beta \llparenthesis Q \rrparenthesis$
(2) *If* $\Psi \vdash M : \tau$ *and* $M \longrightarrow N$ *then* $\llparenthesis M \rrparenthesis \longrightarrow^+ \llparenthesis N \rrparenthesis$

Proof. By induction on the given reduction

Case: $(x \leftarrow M \leftarrow \overline{y_i}; P_0) \longrightarrow (x \leftarrow M' \leftarrow \overline{y_i}; P_0)$ with $M \longrightarrow M'$

$\llparenthesis x \leftarrow M \leftarrow \overline{y_i}; P_0 \rrparenthesis = \llparenthesis P_0 \rrparenthesis \{ \llparenthesis M \rrparenthesis\, \overline{y_i} / x\}$ — by definition
$\llparenthesis M \rrparenthesis \longrightarrow^* \llparenthesis M' \rrparenthesis$ — by i.h.
$\llparenthesis x \leftarrow M' \leftarrow \overline{y_i}; P_0 \rrparenthesis = \llparenthesis P_0 \rrparenthesis \{ \llparenthesis M' \rrparenthesis\, \overline{y_i} / x\}$ — by definition
$\llparenthesis P_0 \rrparenthesis \{ \llparenthesis M \rrparenthesis\, \overline{y_i} / x\} \longrightarrow^*_\beta \llparenthesis P_0 \rrparenthesis \{ \llparenthesis M' \rrparenthesis\, \overline{y_i} / x\}$ — by congruence

Case: $(x \leftarrow \{x \leftarrow Q \leftarrow \overline{y_i}\} \leftarrow \overline{y_i}; P_0) \longrightarrow (\nu x)(Q \mid P_0)$

$\llparenthesis x \leftarrow \{x \leftarrow Q \leftarrow \overline{y_i}\} \leftarrow \overline{y_i}; P_0 \rrparenthesis = \llparenthesis P_0 \rrparenthesis \{((\lambda y_0. \dots .\lambda y_n. \llparenthesis Q \rrparenthesis)\, y_0 \ \dots y_n)/x\}$ — by definition
$\longrightarrow^+_\beta \llparenthesis P_0 \rrparenthesis \{\llparenthesis Q \rrparenthesis / x\}$ — by congruence and transitivity
$\llparenthesis (\nu x)(Q \mid P_0) \rrparenthesis = \llparenthesis P_0 \rrparenthesis \{\llparenthesis Q \rrparenthesis / x\}$ — by definition

□

A.4.3 *Proofs of Inverse Theorem and Full Abstraction for Sess$\pi\lambda^+$.*

Theorem 5.18 (Inverse Encodings). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![\llparenthesis P \rrparenthesis]\!]_z \approx_{\mathsf{L}} [\![P]\!]$. *Also, if* $\Psi \vdash M : \tau$ *then* $\llparenthesis [\![M]\!]_z \rrparenthesis =_\beta \llparenthesis M \rrparenthesis$.

We prove each case as a separate theorem.

Theorem A.7 (Inverse Encodings – Processes). *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $[\![\llparenthesis P \rrparenthesis]\!]_z \approx_{\mathsf{L}} [\![P]\!]$

Proof. By induction on the given typing derivation. We show the new cases.

Case: Rule $\{\}E$

$P = x \leftarrow M \leftarrow \overline{y}; Q$ — by inversion
$\llparenthesis P \rrparenthesis = \llparenthesis Q \rrparenthesis \{ (\llparenthesis M \rrparenthesis\, \overline{y})/x\}$ — by definition
$[\![\llparenthesis Q \rrparenthesis \{ (\llparenthesis M \rrparenthesis\, \overline{y})/x\}]\!]_z = (\nu a)([\![\llparenthesis M \rrparenthesis\, \overline{y}]\!]_a \mid [\![\llparenthesis Q \rrparenthesis]\!]_z \{a/x\})$ — by Lemma 5.2
$= (\nu a, x)([\![\llparenthesis M \rrparenthesis]\!]_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid [\![\llparenthesis Q \rrparenthesis]\!] \{a/x\}) \dots))$ — by definition
$\equiv (\nu x)([\![\llparenthesis M \rrparenthesis]\!]_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid [\![\llparenthesis Q \rrparenthesis]\!]) \dots))$
$[\![P]\!] = (\nu x)([\![M]\!]_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid [\![Q]\!]) \dots))$ — by definition
$\approx_{\mathsf{L}} (\nu x)([\![\llparenthesis M \rrparenthesis]\!]_x \mid \overline{x}\langle a_0 \rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n \rangle.([a_n \leftrightarrow y_n] \mid [\![\llparenthesis Q \rrparenthesis]\!]) \dots))$ — by i.h.

□

Theorem A.8 (Inverse Encodings – λ-terms). *If* $\Psi \vdash M : \tau$ *then* $\llparenthesis [\![M]\!]_z \rrparenthesis =_\beta \llparenthesis M \rrparenthesis$

Proof. By induction on the given typing derivation. We show the new cases.

Case: Rule $\{\}I$

$M = \{x \leftarrow P \leftarrow \overline{y_i}\}$ by inversion
$[\![M]\!]_z = z(y_0).\ldots.z(y_n).[\![P\{z/x\}]\!]$ by definition
$(\!|z(y_0).\ldots.z(y_n).[\![P\{z/x\}]\!]|\!) = \lambda y_0.\ldots.\lambda y_n.(\!|[\![P\{z/x\}]\!]|\!)$ by definition
$[\![M]\!] = \lambda y_0.\ldots.\lambda y_n.(\!|P|\!)$ by definition
$=_\beta \lambda y_0.\ldots.\lambda y_n.(\!|[\![P\{z/x\}]\!]|\!)$ by i.h.

□

A.5 Strong Normalisation for Higher-Order Sessions

Theorem 5.21 (Operational Completeness). *If $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ and $P \longrightarrow Q$ then $(\!|P|\!)^+ \longrightarrow_\beta^+ (\!|Q|\!)^+$*

Proof.

Case: $(\nu u)(!u(x).P_0 \mid \overline{u}\langle x\rangle.P_1) \longrightarrow (\nu u)(!u(x).P_0 \mid (\nu x)(P_0 \mid P_1))$

$(\!|(\nu u)(!u(x).P_0 \mid \overline{u}\langle x\rangle.P_1)|\!)^+ = \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_1|\!)^+\{u/x\}\{(\!|P_0|\!)^+/u\}$
$= \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_1|\!)^+\{(\!|P_0|\!)^+/x\}\{(\!|P_0|\!)^+/u\}$ by definition
$\longrightarrow (\!|P_1|\!)^+\{(\!|P_0|\!)^+/x\}\{(\!|P_0|\!)^+/u\}$ by operational semantics
$(\!|(\nu u)(!u(x).P_0 \mid (\nu x)(P_0 \mid P_1))|\!)^+ = (\!|P_1|\!)^+\{(\!|P_0|\!)^+/x\}\{(\!|P_0|\!)^+/u\}$ by definition

Other cases are unchanged.

□

Theorem 5.22 (Operational Soundness). *If $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ and $(\!|P|\!)^+ \longrightarrow M$ then $P \mapsto^* Q$ such that $(\!|Q|\!) \longrightarrow^* M$.*

Proof.

Case: $(\!|P|\!)^+ = \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_0|\!)^+\{u/x\}$ with $(\!|P|\!)^+ \longrightarrow (\!|P_0|\!)^+\{u/x\}$

$(\!|P|\!)^+ = \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_0|\!)^+\{u/x\} \longrightarrow (\!|P_0|\!)^+\{u/x\}$ by operational semantics, as needed.

Remaining cases are fundamentally unchanged.

□

Theorem 5.23 (Inverse). *If $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ then $[\![(\!|P|\!)^+]\!]_z \approx_{\mathsf{L}} [\![P]\!]$*

Proof.

Case: copy rule

$(\!|P|\!)^+ = \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_0|\!)^+\{u/x\}$ by definition
$[\![\mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ (\!|P_0|\!)^+\{u/x\}]\!]_z = (\nu y)(\mathbf{0} \mid [\![(\!|P_0|\!)^+\{u/x\}]\!]_z)$ by definition
$\equiv [\![(\!|P_0|\!)^+\{u/x\}]\!]_z$ by structural congruence
$\approx_{\mathsf{L}} (\nu x)(\overline{u}\langle w\rangle.[w \leftrightarrow x] \mid [\![(\!|P_0|\!)^+]\!]_z)$ by compositionality
$\approx_{\mathsf{L}} [\![P]\!]$ by i.h. + congruence + definition of $\approx_{\mathsf{L}}$ for open processes

□

Lemma A.9. *If $\Psi \vdash M : \tau$ then $(\!|[\![M]\!]_z|\!)^+ =_\beta (\!|M|\!)^+$*

Proof.

Case: uvar rule

$[\![u]\!]_z = (\nu x)u\langle x\rangle.[x \leftrightarrow z]$ by definition
$(\!|(\nu x)u\langle x\rangle.[x \leftrightarrow z]|\!)^+ = \mathsf{let}\ \mathbf{1} = \langle\rangle\ \mathsf{in}\ u =_\beta u$

□