On Polymorphic Sessions and Functions

A Tale of Two (Fully Abstract) Encodings

Bernardo Toninho and Nobuko Yoshida

Department of Computing, Imperial College London, United Kingdom

Abstract. 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.

1 Introduction

Dating back to Milner's seminal work [29], encodings of λ-calculus into π-calculus are seen as essential benchmarks to examine expressiveness of various extensions of the π-calculus. 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 wrt behavioural equivalence: the semantic distance between the λ-calculus and the π-calculus typically requires either restricting process behaviours [45] (e.g. via typed equivalences [5]) or enriching the λ-calculus with constants that allow for a suitable characterisation of the term equivalence induced by the behavioural equivalence on processes [43].

Encodings in π-calculi also gave rise to new typing disciplines: Session types [20,22], a typing system that is able to ensure deadlock-freedom for communication protocols between two or more parties [23], were originally motivated "from process encodings of various data structures in an asynchronous version of the π-calculus" [21]. Recently, a propositions-as-types correspondence between linear logic and session types [8,9,54] has produced several new developments and logically-motivated techniques [49,54,7,26] to augment both the theory and practice of session-based message-passing concurrency. Notably, parametric session polymorphism [7] (in the sense of Reynolds [41]) 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 [49]. While fully abstract encodings between λ-calculi and π-calculi have been proposed (e.g. [5,43]), 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 [2]).

In the spirit of Gentzen [14], 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 [28] in the polymorphic λ-calculus [1,19] (in a linear setting [6]). 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 [30].

We systematically extend our results to a session calculus with λ-term and process passing (the latter being the core calculus of [50], inspired by Benton's LNL [4]). 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 [46]. In addition, the encoding instantly provides a strong normalisation property of the higher-order session calculus.

Contributions and the outline of our paper are as follows:

§ **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.

§ **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.

§ **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.

§ **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. We demonstrate a use of these algebraic properties via examples.

§ **4.2,4.3** study term-passing and process-passing session calculi, extending our encodings to provide embeddings into the first-order session calculus. We show full abstraction and mutual inversion results, and derive strong normalisation of the higher-order session calculus from the encoding.

In order to introduce our encodings, we first overview Polyπ, its typing system and behavioural equivalence (§ **2**). We discuss related work and conclude with future work (§ **5**). Detailed proofs can be found in [52].

2 Polymorphic Session π-Calculus

This section summarises the polymorphic session π-calculus [7], dubbed Polyπ, arising as a process assignment to second-order linear logic [15], its typing system and behavioural equivalences.

2.1 Processes and Typing

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

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

$x\langle y\rangle.P$ 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

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

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

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

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

Fig. 1. Labelled Transition System.

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 [46].

In the remainder of this work we write $\equiv$ for a standard π-calculus structural congruence extended with the clause $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$. In order to streamline the presentation of observational equivalence [36,7], we write $\equiv_!$ for structural congruence extended with the so-called sharpened replication axioms [46], which capture basic equivalences of replicated processes (and are present in the proof dynamics of the exponential of linear logic). 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 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 $x(y)$; the binary choice actions ($x.\mathsf{inl}$, $\overline{x.\mathsf{inl}}$, $x.\mathsf{inr}$, and $\overline{x.\mathsf{inr}}$); and output and input actions of types ($\overline{x\langle A\rangle}$ and $x(A)$).

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), (lout), (lin), (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 $\xrightarrow{\tau}$ and $\rightarrow^+$ for the transitive closure of $\xrightarrow{\tau}$. Given $\alpha \neq \tau$, notation $\overset{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ and $\overset{\tau}{\Longrightarrow}$ stands for $\Longrightarrow$.

Typing System. The typing rules of Polyπ are given in Fig. 2, following [7]. 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 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

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

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

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

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

Fig. 2. Typing Rules (Abridged – See [52] for all rules).

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 $\cdot;\cdot;\cdot \vdash P :: T$.

As in [8,9,36,54], the typing discipline enforces that channel outputs always have as object a *fresh* name, in the style of the internal mobility π-calculus [44]. We clarify a few of the key rules: Rule $\forall\mathsf{R}$ 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 $\forall\mathsf{L}$), a process must output a type B which then warrants the use of the session as type $A\{B/X\}$. Rule $\multimap\mathsf{R}$ 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 $\otimes\mathsf{R}$) 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. Linear composition is captured by rule cut which 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 $z{:}C$. As shown in [7], typing entails Subject Reduction, Global Progress, and Termination.

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 [41], defined as a contextual logical relation on typed processes [7]. 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 [7], which preserves *observable* actions, called barbs. Formally, *barbed congruence*, noted $\cong$, is the largest equivalence on well-typed processes that is τ-closed, barb preserving, and contextually closed under typed contexts; see [7] and [52] for the full definition.

Logical Equivalence ($\approx_{\mathrm{L}}$). 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 2.3).

The logical relation is defined using the candidates technique of Girard [16]. 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 2.1 (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 $\Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$, 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. We show only select cases (see [52] for the full definition).

Definition 2.2 (Logical Equivalence). **(Base Case)** Given a type A and mappings ω, ω', η, we define *logical equivalence*, noted $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$, 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:

- $P \approx_{\mathrm{L}} Q :: z{:}X[\eta : \omega \Leftrightarrow \omega']$ iff $(P, Q) \in \eta(X)(z)$
- $P \approx_{\mathrm{L}} Q :: z{:}A \multimap B[\eta : \omega \Leftrightarrow \omega']$ iff $\forall P', y.\ (P \xrightarrow{z(y)} P') \Rightarrow \exists Q'.Q \overset{z(y)}{\Longrightarrow} Q'$ s.t. $\forall R_1, R_2.R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'](\nu y)(P' \mid R_1) \approx_{\mathrm{L}} (\nu y)(Q' \mid R_2) :: z{:}B[\eta : \omega \Leftrightarrow \omega']$
- $P \approx_{\mathrm{L}} Q :: z{:}A \otimes B[\eta : \omega \Leftrightarrow \omega']$ iff $\forall P', y.\ (P \xrightarrow{\overline{(\nu y) z\langle y \rangle}} P') \Rightarrow \exists Q'.Q \overset{\overline{(\nu y) z\langle y \rangle}}{\Longrightarrow} Q'$ s.t. $\exists P_1, P_2, Q_1, Q_2.\ P' \equiv_! P_1 \mid P_2 \wedge Q' \equiv_! Q_1 \mid Q_2 \wedge P_1 \approx_{\mathrm{L}} Q_1 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \wedge P_2 \approx_{\mathrm{L}} Q_2 :: z{:}B[\eta : \omega \Leftrightarrow \omega']$
- $P \approx_{\mathrm{L}} Q :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega']$ iff $\forall B_1, B_2, P', \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \xrightarrow{z(B_1)} P')$ implies $\exists Q'.Q \overset{z(B_2)}{\Longrightarrow} Q',\ P' \approx_{\mathrm{L}} Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]]$

(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 $\Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$ (with $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$) is inductively defined as:

$$\Gamma; \Delta, y : A \vdash P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega'] \text{ iff } \forall R_1, R_2.\ \text{s.t. } R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'], \Gamma; \Delta \vdash (\nu y)(\hat{\omega}(P) \mid \hat{\omega}(R_1)) \approx_{\mathrm{L}} (\nu y)(\hat{\omega}'(Q) \mid \hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega']$$

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

For the sake of readability we often omit the $\eta : \omega \Leftrightarrow \omega'$ portion of $\approx_{\mathrm{L}}$, which is henceforth implicitly universally quantified. Thus, we write $\Omega; \Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A$ (or $P \approx_{\mathrm{L}} Q$) 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 [16]. 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 2.3 (Properties of Logical Equivalence [7]).

Parametricity: *If $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ then, for all $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$, we have $\Gamma; \Delta \vdash \hat{\omega}(P) \approx_{\mathrm{L}} \hat{\omega}'(P) :: z{:}A[\eta : \omega \Leftrightarrow \omega']$.*

Soundness: *If $\Omega; \Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A$ then $\mathcal{C}[P] \cong \mathcal{C}[Q] :: z{:}A$, for any closing $\mathcal{C}[-]$.*

Completeness: *If $\Omega; \Gamma; \Delta \vdash P \cong Q :: z{:}A$ then $\Omega; \Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A$.*

3 To Linear-F and Back

We now develop our mutually inverse and fully abstract encodings between Polyπ and a linear polymorphic λ-calculus [55] 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.

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

The syntax of types is that of the multiplicative and exponential fragments of second-order intuitionistic linear logic: $\lambda x{:}A.M$ 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 [3], 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 standard [7]. The operational semantics of the calculus are the expected call-by-name semantics with commuting conversions [27]. We write $\Downarrow$ for the 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 [5]).

3.1 Encoding Linear-F into Session π-Calculus

We define a translation from Linear-F to Polyπ generalising the one from [49], 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 [14], 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 [6] 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 Church encodings as specified above. The translation on λ-terms is given below:

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

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 $\multimap\mathsf{R}$ 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 $[\![N]\!]_z$.

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 $(\nu w)z\langle w\rangle.P$):

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

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

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

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

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

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

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

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.

Theorem 3.4 (Operational Correspondence).

- *If* $\Omega; \Gamma; \Delta \vdash M : A$ *and* $M \to N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $[\![N]\!]_z \approx_{\mathtt{L}} P$
- *If* $[\![M]\!]_z \to P$ *then* $M \to^+ N$ *and* $[\![N]\!]_z \approx_{\mathtt{L}} P$

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. 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).

$$\left(\!\!\left(\multimap\mathsf{R})\ \dfrac{\Delta, x{:}A \vdash P :: z{:}B}{\Delta \vdash z(x).P :: z{:}A \multimap B}\right)\!\!\right) \triangleq (\multimap I)\ \dfrac{\Delta, x{:}A \vdash (\!|P|\!)_{\Delta, x:A \vdash z:B} : B}{\Delta \vdash \lambda x{:}A.(\!|P|\!)_{\Delta, x:A \vdash z:B} : A \multimap B}$$

$$\left(\!\!\left(\dfrac{(\multimap\mathsf{L})\quad \Delta_1 \vdash P :: y{:}A \quad \Delta_2, x{:}B \vdash Q :: z{:}C}{\Delta_1, \Delta_2, x{:}A \multimap B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: z{:}C}\right)\!\!\right) \triangleq$$

$$\text{(SUBST)}\ \dfrac{\Delta_2, x{:}B \vdash (\!|Q|\!)_{\Delta_2, x:B \vdash z:C} : C \qquad \dfrac{(\multimap E)\quad x{:}A \multimap B \vdash x{:}A \multimap B \quad \Delta_1 \vdash (\!|P|\!)_{\Delta_1 \vdash y:A} : B}{\Delta_1, x{:}A \multimap B \vdash x\,(\!|P|\!)_{\Delta_1 \vdash y:A} : B}}{\Delta_1, \Delta_2, x{:}A \multimap B \vdash (\!|Q|\!)_{\Delta_2, x:B \vdash z:C}\{(x\,(\!|P|\!)_{\Delta_1 \vdash y:A})/x\} : C}$$

Fig. 3. Translation on Typing Derivations (Excerpt – See [52])

Definition 3.5 (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 (see Fig. 3 for an excerpt of the translation on typing derivations, where we write $(\!|P|\!)_{\Omega;\Gamma;\Delta \vdash z:A}$ to denote the translation of $\Omega; \Gamma; \Delta \vdash P :: z{:}A$. We omit Ω and Γ when unchanged).

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

For instance, the encoding of a process $z(x).P :: z{:}A \multimap B$, typed by rule $\multimap\mathsf{R}$, results in the corresponding $\multimap I$ introduction rule in the λ-calculus and thus is $\lambda x{:}A.(\!|P|\!)$. To encode the process $(\nu y)x\langle y\rangle.(P \mid Q)$, typed by rule $\multimap\mathsf{L}$, 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 $(\!|Q|\!)\{(x\,(\!|P|\!))/x\}$. The term $x\,(\!|P|\!)$ consists of the application of x (of function type) to the argument $(\!|P|\!)$, thus ensuring that the term resulting from the substitution is of the appropriate type. We note that, for instance, the encoding of rule $\otimes\mathsf{L}$ does not need to appeal to substitution – the λ-calculus let style rules can be mapped directly. Similarly, rule $\forall\mathsf{R}$ is mapped to type abstraction, whereas rule $\forall\mathsf{L}$ 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 $(\!|P|\!)$. The encoding of existential polymorphism 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 3.6. *If* $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ *then* $(\!|\Omega|\!);(\!|\Gamma|\!);(\!|\Delta|\!) \vdash (\!|P|\!) : A$.

Example 3.7 (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 $z{:}\forall X.\forall Y.X \multimap Y \multimap X \otimes Y \vdash Q :: r{:}\mathbf{1}$.
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 $(\nu z)(P \mid Q) \to^{+} \mathbf{0} = (\!|\langle\rangle|\!)$.

In general, the translation of Def. 3.5 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 $\multimap\mathsf{R}$ and $\multimap\mathsf{L}$, where the continuation process $w(z).\mathbf{0}$ offers a session $w{:}\mathbf{1} \multimap \mathbf{1}$ (and so must use rule $\mathbf{1}\mathsf{L}$ 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). Thus, to establish operational completeness we consider full β-reduction, denoted by $\to_\beta$, i.e. enabling β-reductions under binders.

Theorem 3.8 (Operational Completeness). *Let* $\Omega;\Gamma;\Delta \vdash P :: z{:}A$. *If* $P \to Q$ *then* $(\!|P|\!) \to_\beta^* (\!|Q|\!)$.

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 [36], 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 [5] and in [26], in a classical linear logic setting. For conciseness, we define extended reduction as a relation on *typed* processes modulo $\equiv$.

Definition 3.9 (Extended Reduction [5]). We define $\mapsto$ as the type preserving relations on typed processes modulo $\equiv$ generated by:

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

where $\mathcal{C}$ is a (typed) process context which does not capture the bound name y.

Theorem 3.10 (Operational Soundness). *Let $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ and $(\!|P|\!) \to M$, there exists Q such that $P \mapsto^* Q$ and $(\!|Q|\!) =_\alpha M$.*

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 3.11 (Inverse). *If $\Omega;\Gamma;\Delta \vdash M : A$ then $\Omega;\Gamma;\Delta \vdash (\!|[\![M]\!]_z|\!) \cong M : A$. Also, if $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ then $\Omega;\Gamma;\Delta \vdash [\![(\!|P|\!)]\!]_z \approx_{\mathtt{L}} P :: z{:}A$*

Corollary 3.12 (Fullness). *Let $\Omega;\Gamma;\Delta \vdash P :: z{:}A$. $\exists M$ s.t. $\Omega;\Gamma;\Delta \vdash M : A$ and $\Omega;\Gamma;\Delta \vdash [\![M]\!]_z \approx_{\mathtt{L}} P :: z{:}A$ Also, let $\Omega;\Gamma;\Delta \vdash M : A$. $\exists P$ s.t. $\Omega;\Gamma;\Delta \vdash P :: z{:}A$ 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 [18] and proved by showing the relation generated by $[\![M]\!]_z \approx_{\mathtt{L}} [\![N]\!]_z$ forms $\cong$; we then establish the *completeness* direction by contradiction, using fullness.

Theorem 3.13 (Full Abstraction). *$\Omega;\Gamma;\Delta \vdash M \cong N : A$ iff $\Omega;\Gamma;\Delta \vdash [\![M]\!]_z \approx_{\mathtt{L}} [\![N]\!]_z :: z{:}A$.*

We can straightforwardly combine the above full abstraction with Theorem 3.11 to obtain full abstraction of the $(\!|-|\!)$ translation.

Theorem 3.14 (Full Abstraction). *$\Omega;\Gamma;\Delta \vdash P \approx_{\mathtt{L}} Q :: z{:}A$ iff $\Omega;\Gamma;\Delta \vdash (\!|P|\!) \cong (\!|Q|\!) : A$.*

4 Applications of the Encodings

In this section we develop applications of the encodings of the previous sections. Taking advantage of full abstraction and mutual inversion, we apply non-trivial properties from the theory of the λ-calculus to our session-typed process setting.

In § 4.1 we study inductive and coinductive sessions, arising through encodings of initial F-algebras and final F-coalgebras in the polymorphic λ-calculus.

In § 4.2 we study encodings for an extension of the core session calculus with term passing, where terms are derived from a simply-typed λ-calculus. Using the development of § 4.2 as a stepping stone, we generalise the encodings to a *higher-order* session calculus (§ 4.3), where processes can send, receive and execute other processes.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.

4.1 Inductive and Coinductive Session Types

The study of polymorphism in the λ-calculus [1,19,40,6] 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 [28], 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 calculus can express fairly intricate communication behaviours, including generic protocols through both existential and universal polymorphism (i.e. protocols that are parametric in their sub-protocols). Using our fully abstract encodings between the two calculi, we show that session polymorphism is expressive enough to encode inductive and coinductive sessions, "importing" the results for the λ-calculus, which may then be instantiated to provide a session-typed formulation of the encodings of data structures in the π-calculus of [30].

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 [42,1] (we write $F(X)$ to denote that X occurs 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 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.

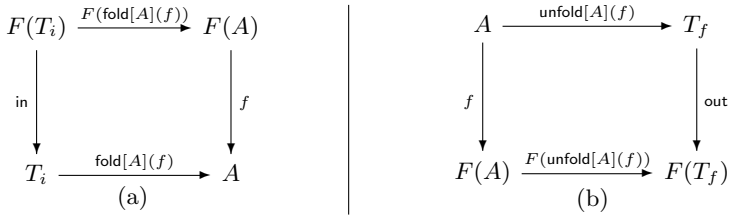

Fig. 4. Diagrams for Initial F-algebras and Final F-coalgebras

While the complete formal development of the representation of inductive and coinductive types in System F would lead us to far astray, we summarise here the key concepts as they apply to the λ-calculus (the interested reader can refer to [19] 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. 4(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, it is easy to see that:

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

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

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

such that the diagram in Fig. 4(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 [6] 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:

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

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 $(\nu y)x\langle y\rangle.P$):

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

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

$$\mathsf{foldP}(A)_{y_1,y_2} \triangleq (\nu x)([\![\mathsf{fold}]\!]_x \mid x\langle A\rangle.\overline{x}\langle v\rangle.(\overline{u}\langle y\rangle.[y \leftrightarrow v] \mid \overline{x}\langle z\rangle.([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2])))$$
$$\mathsf{unfoldP}(A)_{y_1,y_2} \triangleq (\nu x)([\![\mathsf{unfold}]\!]_x \mid x\langle A\rangle.\overline{x}\langle v\rangle.(\overline{u}\langle y\rangle.[y \leftrightarrow v] \mid \overline{x}\langle z\rangle.([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2])))$$

where $\mathsf{foldP}(A)_{y_1,y_2}$ 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, $\mathsf{unfoldP}(A)_{y_1,y_2}$ 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). *$\forall Q$ such that $X; u{:}F(X) \multimap X; y_1{:}T_i \vdash Q :: y_2{:}X$ we have $X; u{:}F(X) \multimap X; y_1{:}T_i \vdash Q \approx_{\mathrm{L}} \mathsf{foldP}(X)_{y_1,y_2} :: y_2{:}X$*

Theorem 4.2 (Universality of $\mathsf{unfoldP}$). *$\forall Q$ and F-coalgebra A s.t $\cdot; \cdot; y_1{:}A \vdash Q :: y_2{:}T_f$ we have that $\cdot; u{:}F(A) \multimap A; y_1{:}A \vdash Q \approx_{\mathrm{L}} \mathsf{unfoldP}(A)_{y_1,y_2} :: y_2 :: T_f$.*

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:

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

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:

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

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:

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

We note that the resulting encoding is reminiscent of the encoding of lists of [30] (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):

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

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:

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

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:

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

4.2 Communicating Values – Sess$\pi\lambda$

We now 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$\pi\lambda$.

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

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.

To First-Order Processes We now introduce our encoding for Sess$\pi\lambda$, defined inductively on session types, processes, types and λ-terms (we omit the purely inductive cases on session types and processes for conciseness). As before, the encoding on processes is defined on *typing derivations*, where we indicate the typing rule at the root of the typing derivation.

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

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

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

The encoding addresses the non-linear usage of data elements in processes by encoding the types $\tau \wedge A$ and $\tau \supset A$ as $!{[\![\tau]\!]} \otimes [\![A]\!]$ and $!{[\![\tau]\!]} \multimap [\![A]\!]$, 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 $!{[\![\tau]\!]} \multimap [\![\sigma]\!]$, following Girard's embedding of intuitionistic logic in linear logic [15].

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 [49]. The λ-abstraction is translated as input. Since variables in a λ-abstraction may be used non-linearly, the case for variables and application

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

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:

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

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

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:

$$\begin{aligned}&(\nu z)(z(x).z\langle x\rangle.z\langle(\lambda y{:}\sigma.x)\rangle.\mathbf{0} \mid z\langle\lambda w{:}\tau_0.w\rangle.P)\\ &\qquad\qquad \rightarrow (\nu z)(z\langle\lambda w{:}\tau_0.w\rangle.z\langle(\lambda y{:}\sigma.\lambda w{:}\tau_0.w)\rangle.\mathbf{0} \mid P) \qquad (1)\end{aligned}$$

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.

Lemma 4.5 (Compositionality). *Let $\Psi, x{:}\tau \vdash M : \sigma$ and $\Psi \vdash N : \tau$. We have that $[\![M\{N/x\}]\!]_z \approx_{\mathtt{L}} (\nu x)([\![M]\!]_z \mid !x(y).[\![N]\!]_y)$*

Revisiting the process to the left of the arrow in Equation 1 we have:

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

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

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

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 establish operational correspondence for the encoding.

Theorem 4.6 (Operational Correspondence).

1. *If $\Psi \vdash M : \tau$ and $[\![M]\!]_z \rightarrow Q$ then $M \rightarrow^+ N$ such that $[\![N]\!]_z \approx_{\mathtt{L}} Q$*

2. *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $[\![P]\!] \to Q$ *then* $P \to^+ P'$ *such that* $[\![P']\!] \approx_{\mathrm{L}} Q$
3. *If* $\Psi \vdash M : \tau$ *and* $M \to N$ *then* $[\![M]\!]_z \Longrightarrow P$ *such that* $P \approx_{\mathrm{L}} [\![N]\!]_z$
4. *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \to Q$ *then* $[\![P]\!] \to^+ R$ *with* $R \approx_{\mathrm{L}} [\![Q]\!]$

The process equivalence in Theorem 4.6 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 processes to linear λ-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 § 3.2.

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

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

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

The treatment of non-linear components of processes is identical to our previous encoding: non-linear functions $\tau \to \sigma$ 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 $\wedge\mathsf{R}$) 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:

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

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 3.6, 3.8, and 3.10). The full development can be found in [52].

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

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 [3,31]. Moreover, the extensional flavour of $\approx_{\mathtt{L}}$ includes η-like principles at the process level.

Theorem 4.8. *Let* $\cdot \vdash M : \tau$ *and* $\cdot \vdash N : \tau$. $(\!|M|\!) =_{\beta\eta} (\!|N|\!)$ *iff* $[\![M]\!]_z \approx_{\mathtt{L}} [\![N]\!]_z$. *Also, let* $\cdot \vdash P :: z{:}A$ *and* $\cdot \vdash Q :: z{:}A$. *We have that* $[\![P]\!] \approx_{\mathtt{L}} [\![Q]\!]$ *iff* $(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$.

We establish full abstraction for the encoding of λ-terms into processes (Theorem 4.8) 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 4.8 uses the same strategy of Theorem 3.14, appealing to the inverse theorems.

4.3 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 [50], 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$\pi\lambda^+$.

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

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` [37,50]. 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 [50]. 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 $\vdash Q :: d{:}\mathsf{Nat} \wedge \mathbf{1}$):

$$P \triangleq (\nu c)(c\langle\{d \leftarrow Q\}\rangle.c(x).\mathbf{0} \mid c(y).d \leftarrow y; d(n).c\langle n\rangle.\mathbf{0}) \tag{2}$$

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:

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

Given the seminal work of Sangiorgi [46], 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 § 4.2 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.

$$\llbracket\{\overline{x_j{:}A_j} \vdash z{:}A\}\rrbracket \quad \triangleq \overline{\llbracket A_j\rrbracket} \multimap \llbracket A\rrbracket$$

$$\llbracket\{x \leftarrow P \rightarrow \overline{y_i}\}\rrbracket_z \triangleq z(y_0).\ldots.z(y_n).\llbracket P\{z/x\}\rrbracket \quad (z \notin \mathit{fn}(P))$$

$$\llbracket x \leftarrow M \leftarrow \overline{y_i}; Q\rrbracket \triangleq (\nu x)(\llbracket M\rrbracket_x \mid \overline{x}\langle a_0\rangle.([a_0 \leftrightarrow y_0] \mid \cdots \mid x\langle a_n\rangle.([a_n \leftrightarrow y_n] \mid \llbracket Q\rrbracket)\ldots))$$

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 $\llbracket A_0\rrbracket \multimap \ldots \multimap \llbracket A_n\rrbracket \multimap \llbracket A\rrbracket$, which we write as $\overline{\llbracket A_j\rrbracket} \multimap \llbracket A\rrbracket$. 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 2 is thus:

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

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. The full development is omitted but can be found in [52].

We encode Sess$\pi\lambda^+$ into λ-terms, extending § 4.2 with:

$$(\!|\{\overline{x_i{:}A_i} \vdash z{:}A\}|\!) \triangleq \overline{(\!|A_i|\!)} \multimap (\!|A|\!) \qquad (\!|x \leftarrow M \leftarrow \overline{y_i}; Q|\!) \triangleq (\!|Q|\!)\{((\!|M|\!)\,\overline{y_i})/x\} \qquad (\!|\{x \leftarrow P \leftarrow \overline{w_i}\}|\!) \triangleq \lambda w_0.\ldots.\lambda w_n.(\!|P|\!)$$

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. 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.

Theorem 4.9 (Strong Normalisation). *Let $\Psi; \Gamma; \Delta \vdash P :: z{:}A$. There is no infinite reduction sequence starting from P.*

Theorem 4.10 (Inverse Encodings). *If $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ then $[\![(\!|P|\!)]\!]_z \approx_{\mathtt{L}} [\![P]\!]$. Also, if $\Psi \vdash M : \tau$ then $(\!|[\![M]\!]_z|\!) =_\beta (\!|M|\!)$.*

Theorem 4.11. *Let $\vdash M : \tau$, $\vdash N : \tau$, $\vdash P :: z{:}A$ and $\vdash Q :: z{:}A$. $(\!|M|\!) =_{\beta\eta} (\!|N|\!)$ iff $[\![M]\!]_z \approx_{\mathtt{L}} [\![N]\!]_z$ and $[\![P]\!] \approx_{\mathtt{L}} [\![Q]\!]$ iff $(\!|P|\!) =_{\beta\eta} (\!|Q|\!)$.*

5 Related Work and Concluding Remarks

Process Encodings of Functions. Toninho et al. [49] study encodings of the simply-typed λ-calculus in a logically motivated session π-calculus, via encodings to the linear λ-calculus. Our work differs since they do not study polymorphism nor reverse encodings; and we provide deeper insights through applications of the encodings. Full abstraction or inverse properties are not studied.

Sangiorgi [43] 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 [45], 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 [47] 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 [47] in our typed setting.

Wadler [54] 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 [26], 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 [25]. Mapping to the terminology used in our work [17], 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. Berger et al. [5] study an encoding of System F in a polymorphic linear π-calculus, showing it to be fully abstract based on game semantics techniques. Their typing system and proofs are more complex due to the fine-grained constraints from game semantics. Moreover, they do not study a reverse encoding.

Orchard and Yoshida [33] 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 [7] studies parametric session polymorphism for the intuitionistic setting, developing a behavioural equivalence that captures parametricity, which is used (denoted as $\approx_{\mathtt{L}}$) in our paper. The work [39] 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 [41]. Their typing discipline (originally from [53], 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 [38]) 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.

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 [28], with their strong definability in System F appearing in [1] and [19]. The definability of inductive and coinductive types using parametricity also appears in [40] in the context of a logic for parametric polymorphism and later in [6] in a linear variant of such a logic. The work of [55] 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 [51] and [26]. 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.

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. 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 linear logic-based articulation of name-passing interactions originally proposed by [8] (and studied extensively thereafter e.g. [50,51,36,9,54,7,25]) provides a clear and applicable tool for message-passing concurrency. 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 [13,12,36,7,5]. Future work includes extensions to the classical linear logic-based framework, including multiparty session types [10,11]. Encodings of session π-calculi to the λ-calculus have been used to implement session primitives in functional languages such as Haskell (see a recent survey [32]), OCaml [34,35,24] and Scala [48]. Following this line of work, 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. In particular, we wish to further study the meaning of functors, natural transformations and related constructions in a session-typed setting, both from a more fundamental viewpoint but also in terms of programming patterns.

Acknowledgements. The authors would like to thank the reviewers for their comments, suggestions and pointers to related works. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1.

References

1. Bainbridge, E.S., Freyd, P.J., Scedrov, A., Scott, P.J.: Functorial polymorphism. Theor. Comput. Sci. 70(1), 35–64 (1990)
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. In: ICFP (2017)
3. Barber, A.: Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, School of Informatics, University of Edinburgh (1996)
4. Benton, N.: A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In: CSL. pp. 121–135 (1994)
5. Berger, M., Honda, K., Yoshida, N.: Genericity and the pi-calculus. Acta Inf. 42(2-3), 83–141 (2005)
6. Birkedal, L., Møgelberg, R.E., Petersen, R.L.: Linear Abadi and Plotkin Logic. Logical Methods in Computer Science 2(5) (2006)
7. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: ESOP 2013. pp. 330–349 (2013)
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR 2010. pp. 222–236 (2010)
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science 26(3), 367–423 (2016)
10. Carbone, M., Lindley, S., Montesi, F., Schuermann, C., Wadler, P.: Coherence generalises duality: a logical explanation of multiparty session types. In: CONCUR'16. vol. 59, pp. 33:1–33:15. Sch. Dag. (2016)
11. Carbone, M., Montesi, F., Schurmann, C., Yoshida, N.: Multiparty session types as coherence proofs. In: CONCUR 2015. vol. 42, pp. 412–426. Sch. Dag. (2015)
12. Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Mobile processes and termination. In: Semantics and Algebraic Specification. pp. 250–273 (2009)
13. Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Termination in higher-order concurrent calculi. J. Log. Algebr. Program. 79(7), 550–577 (2010)
14. Gentzen, G.: Untersuchungen über das logische schließen. Mathematische Zeitschrift 39, 176–210 (1935)
15. Girard, J.: Linear logic. Theor. Comput. Sci. 50, 1–102 (1987)
16. Girard, J., Lafont, Y., Taylor, P.: Proofs and Types. C. U. P. (1989)
17. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. Inf. Comput. 208(9), 1031–1053 (2010)
18. Gorla, D., Nestmann, U.: Full abstraction for expressiveness: history, myths and facts. Mathematical Structures in Computer Science 26(4), 639–654 (2016)
19. Hasegawa, R.: Categorical data types in parametric polymorphism. Mathematical Structures in Computer Science 4(1), 71–109 (1994)
20. Honda, K.: Types for dyadic interaction. In: CONCUR'93. pp. 509–523 (1993)
21. Honda, K.: Session types and distributed computing. In: ICALP (2012)
22. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: ESOP'98. pp. 22–138 (1998)
23. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL'08. pp. 273–284 (2008)
24. Imai, K., Yoshida, N., Yuen, S.: Session-ocaml: a session-based library with polarities and lenses . In: COORDINATION. LNCS, vol. 10319, pp. 99–118 (2017)
25. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: ESOP'15. pp. 560–584 (2015)
26. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: ICFP 2016. pp. 434–447 (2016)

27. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. T. C. S. 228(1-2), 175–210 (1999)
28. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: LICS'87. pp. 30–36 (1987)
29. Milner, R.: Functions as processes. Math. Struct. in C.S. 2(2), 119–141 (1992)
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Inf. Comput. 100(1), 1–77 (1992)
31. Ohta, Y., Hasegawa, M.: A terminating and confluent linear lambda calculus. In: RTA'06. pp. 166–180 (2006)
32. Orchard, D., Yoshida, N.: Session types with linearity in Haskell. In: Behavioural Types: from Theory to Tools. River Publishers (2017)
33. Orchard, D.A., Yoshida, N.: Effects as sessions, sessions as effects. In: POPL 2016. pp. 568–581 (2016)
34. Padovani, L.: A Simple Library Implementation of Binary Sessions. JFP 27 (2016)
35. Padovani, L.: Context-Free Session Type Inference. In: ESOP 2017 (2017)
36. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: ESOP. pp. 539–558 (2012)
37. Pfenning, F., Griffith, D.: Polarized substructural session types. In: FoSSaCS. pp. 3–22 (2015)
38. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
39. Pierce, B.C., Sangiorgi, D.: Behavioral equivalence in the polymorphic pi-calculus. J. ACM 47(3), 531–584 (2000)
40. Plotkin, G.D., Abadi, M.: A logic for parametric polymorphism. In: TLCA '93. pp. 361–375 (1993)
41. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
42. Reynolds, J.C., Plotkin, G.D.: On functors expressible in the polymorphic typed lambda calculus. Inf. Comput. 105(1), 1–29 (1993)
43. Sangiorgi, D.: An investigation into functions as processes. In: MFPS (1993)
44. Sangiorgi, D.: Pi-calculus, internal mobility, and agent-passing calculi. Theor. Comput. Sci. 167(1&2), 235–274 (1996)
45. Sangiorgi, D.: Lazy functions and mobile processes. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 691–720 (2000)
46. Sangiorgi, D., Walker, D.: The pi-calculus: A theory of mobile processes (2001)
47. Sangiorgi, D., Xu, X.: Trees from functions as processes. In: CONCUR (2014)
48. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In: ECOOP'17 (2017)
49. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: FOSSACS 2012. pp. 346–360 (2012)
50. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP. pp. 350–369 (2013)
51. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: TGC 2014. pp. 159–175 (2014)
52. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings (long version). CoRR abs/1711.00878 (2017)
53. Turner, D.: The polymorphic pi-calculus: Theory and implementation. Tech. Rep. ECS-LFCS-96-345, School of Informatics, University of Edinburgh (1996)
54. Wadler, P.: Propositions as sessions. J. Funct. Program. 24(2-3), 384–418 (2014)
55. Zhao, J., Zhang, Q., Zdancewic, S.: Relational parametricity for a polymorphic linear lambda calculus. In: APLAS. pp. 344–359 (2010)