Polymorphic Session Processes as Morphisms

Bernardo Toninho[1] and Nobuko Yoshida[2]

[1] Universidade Nova de Lisboa and NOVA-LINCS, Portugal
[2] Imperial College London, United Kingdom

Abstract. The study of expressiveness of concurrent processes via session types opens a connection between linear logic and mobile processes, grounded in the rigorous logical background of propositions-as-types. One such study includes a notion of parametric session polymorphism, which connects session typed processes with rich higher-order functional computations. This work proposes a novel and non-trivial application of session parametricity – an encoding of inductive and coinductive session types, justified via the theory of initial algebras and final co-algebras using a processes-as-morphisms viewpoint. The correctness of the encoding (i.e. universality) relies crucially on parametricity and the associated relational lifting of sessions.

Keywords: Expressiveness, Session Types, π-calculus, F-Algebra, Linear Logic

1 Introduction

The study of expressiveness of the π-calculus by Palamidessi [20,19] opened a new field of process calculi, linking the areas of distributed computation and distributed algorithms with the π-calculus. Around the same period, *session types* [13] were introduced as a typing discipline that is able to ensure communication safety and deadlock-freedom of communication protocols among two or more parties [14]. In particular, a tight connection between session types and linear logic (a *propositons-as-session types* correspondence) [6,7] has produced several new developments and logically-motivated techniques [27,32,15] to augment both the theory and practice of session-based message-passing concurrency. Notably, parametric session polymorphism [5] (in the sense of Reynolds [24]) has been proposed and a corresponding Abstraction theorem has been established. Despite its practical significance [22], the expressiveness of parametric session polymorphism has been relatively unexplored.

In this paper, we study the expressiveness of parametric session polymorphism in a logical setting and the induced parametric equivalence on session-typed processes. More precisely, we study a notion of *processes-as-morphisms* and *session type operators-as-functors* (in the categorical sense) to develop an encoding of inductive and coinductive session types in the style of initial algebras and final coalgebras in System F [11,1,31,23] but in a process setting, introducing new interesting reasoning techniques such as a relational lifting of sessions, crucial for the correctness of the encoding of coinductive session types.

To develop the encoding of inductive and coinductive types (§ 3.3 and § 3.4) we introduce the notion of a functorial action on typed processes (§ 3.2), showing it satisfies

functoriality up-to parametric equivalence (Theorem 3.3), and enabling us to represent the necessary morphisms as session-typed processes (Theorems 3.7 and 3.9). Crucially, to show the universality of our encoding, i.e. the uniqueness clauses of initiality (Theorem 3.8) and finality (Theorem 3.12), we rely on fundamental properties of parametricity applied to a process setting, such as relational liftings of sessions.

Our results are yet another testament to the strength of propositions-as-sessions: by relying on a rigorous logical background we are able to produce rigorous results such as universality in a session-typed process setting using elegant logical techniques, which give rise to novel reasoning techniques in a process setting such as relational liftings (used in the arguably non-trivial proof of Lemma 3.10). Just as the categorical view point reveals new approaches to functional programming, this work aims to be a stepping stone towards a general notion of algebraic programming [2,3] in a concurrent setting, exploiting the algebraic structures that naturally arise via propositions-as-sessions in order to provide new abstraction mechanisms for concurrent programming. By barely scratching the surface, we can already study a concurrent notion of functor, initial algebra and final co-algebra. The study of further abstraction techniques such as natural transformations is the obvious continuation of this work.

Contributions We summarise the key contributions of this work:

- We develop a notion of functorial action of a type operator on session-typed processes, satisfying the necessary functor laws;
- We study encodings of inductive session types as initial algebras (§ 3.3) of a given type operator and, dually, coinductive session types as final coalgebras (§ 3.4).
- Using parametricity, we show our encodings satisfy the necessary initiality (Theorems 3.7 and 3.8) and finality (Theorems 3.9 and 3.12) properties, and are thus correct wrt the semantics of inductive and coinductive types.

We conclude with discussion of related work (§ 4). The technical report [30] includes the omitted definitions and detailed proofs.

2 Polymorphic Session π-Calculus

This section summarises the polymorphic session π-calculus dubbed Polyπ [5], arising as a process assignment to second-order linear logic [9], its typing system and behavioural equivalences.

The calculus is a synchronous π-calculus with binary guarded choice, input-guarded replication, channel links and prefixes for type input and output.

2.1 Processes and Typing

Definition 2.1 (Process and Type Syntax). 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:*

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

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

Session types denote the communication behaviour that takes place along a given channel between communicating processes. Our 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, signaling appropriately to the communicating partner; $!A$ denotes a session offering an unbounded (but finite) number of behaviours of type A; $\forall X.A$ denotes a polymorphic session that receives a type B and behaves uniformly as $A\{B/X\}$; dually, $\exists X.A$ denotes an existentially typed session, which emits a type B and behaves as $A\{B/X\}$.

Operational Semantics The operational semantics of our calculus are presented as a standard labelled transition system (Fig. 1) modulo $\equiv_\alpha$ congruence, in the style of the *early* system for the π-calculus [26]. In the remainder of this work we write $\equiv$ for the standard π-calculus structural congruence, extended with the clause $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$. We write $\equiv_!$ for structural congruence extended with the so-called sharpened replication axioms [26].

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

Definition 2.2 (Labeled Transition System). The relation *labeled transition* ($P \xrightarrow{\alpha} Q$) is defined by the rules in Fig. 1, subject to the side conditions: in rule (res), we require $y \notin \mathit{fn}(\alpha)$; in rule ($\mathsf{par}$), we require $\mathit{bn}(\alpha) \cap \mathit{fn}(R) = \emptyset$; in rule ($\mathsf{close}$), we require $y \notin \mathit{fn}(Q)$. We omit the symmetric versions of rules (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 $\xrightarrow{\tau}$. Given $\alpha \neq \tau$, notation $\overset{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ and $\overset{\tau}{\Longrightarrow}$ stands for $\Longrightarrow$.

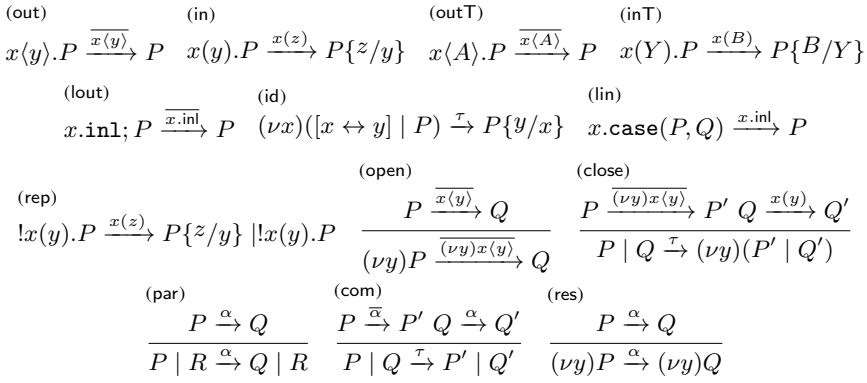

Fig. 1: π-calculus Labeled Transition System.

Typing System The typing system $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ is given in Fig. 2. The judgement means 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. $fv(A) \subseteq \Omega$). The rules are defined up-to structural congruence $\equiv$. We often write T for the right-hand side typing $z{:}A$, $\cdot$ for the empty context and Δ, Δ' for the union of contexts Δ and Δ', only defined when Δ and Δ' are disjoint. We write $\cdot \vdash P :: T$ to denote that P is *closed*.

As in [6,7,21,32], the typing discipline enforces that outputs always have as object a *fresh* name, in the style of the internal mobility π-calculus [25]. The typing rules of Fig. 2 can be divided into three classes: *right rules* (marked with R), which explicate how a process can *offer* a session of a given type; *left rules* (marked with L), which define how a process can *use* or interact with a session of a given type; and *judgmental rules* which define basic logical principles such as composition of proofs (rule cut) and identity (rule id). Rule $\forall$R defines the meaning of (impredicative) universal quantification over session types, stating that a session of type $\forall X.A$ inputs a type and then behaves uniformly as A; dually, to use such a session (rule $\forall$L), a process must output a type B which then warrants the use of the session as type $A\{B/X\}$. Rule $\multimap$R types session input, where a session of type $A \multimap B$ inputs a session of type A which is used to produce a session of type B; dually, one uses such a session (rule $\multimap$L) 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 x, which is then used as a type B.

The typing system ensures strong correctness properties on typed processes. As shown in [5], typing entails Subject Reduction, Global Progress, and Termination.

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

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

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

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

Fig. 2: Type System (Selected Rules)

2.2 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 [24], defined as a contextual logical relation on typed processes [5]. 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 necessarily instantiated with *the same*, but rather *related* types.

Typed Barbed Congruence ($\cong$) We use the typed contextual congruence from [5], 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 [5] for the full definition.

Logical Equivalence The definition of logical equivalence is no more than a typed contextual bisimulation with the following 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 specified session type. We can then show that such an equivalence satisfies the necessary fundamental properties (Theorem 2.4).

Formally, the logical relation is defined using the candidates technique of Girard [8,10]. In this setting, an *equivalence candidate* is an equivalence relation on well-typed processes satisfying basic closure conditions: an equivalence candidate must be compatible with barbed congruence and closed under forward and backward reduction.

We write $\mathcal{R} :: z{:}A \Leftrightarrow B$ for such a candidate relation, such that $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ further requires that $\cdot \vdash P :: z{:}A$ and $\cdot \vdash Q :: z{:}B$.

To define the logical relation we rely on some auxiliary notation. We write $\omega : \Omega$ to denote that a type substitution ω assigns a closed type to the type variables in Ω. Given two substitutions $\omega : \Omega$ and $\omega' : \Omega$, we define a candidate assignment η between ω and ω' as a mapping of 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. 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 substitution ω to P.

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

Definition 2.3 (Logical Equivalence). ***(Base Case)*** *Given a type A and mappings ω, ω', η, we define* logical equivalence*, noted $P \approx_{\text{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$, as the smallest 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:*

$$
\begin{aligned}
P \approx_{\text{L}} Q :: z{:}X[\eta : \omega \Leftrightarrow \omega'] \;\; &\textit{iff} \;\; (P, Q) \in \eta(X)(z) \\
P \approx_{\text{L}} Q :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega'] \;\; &\textit{iff} \;\; \forall B_1, B_2, P', \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\;\; (P \xrightarrow{z(B_1)} P') \; \textit{implies} \\
&\quad \exists Q'.Q \overset{z(B_2)}{\Longrightarrow} Q',\; P' \approx_{\text{L}} Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]] \\
P \approx_{\text{L}} Q :: z{:}\exists X.A[\eta : \omega \Leftrightarrow \omega'] \;\; &\textit{iff} \;\; \exists B_1, B_2, \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\;\; (P \xrightarrow{\overline{z\langle B_1 \rangle}} P') \; \textit{implies} \\
&\quad \exists Q'.Q \overset{\overline{z\langle B_2 \rangle}}{\Longrightarrow} Q',\; P' \approx_{\text{L}} Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]]
\end{aligned}
$$

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

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

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

It is instructive to inspect the clauses for type input and output ($\forall X.A$ and $\exists X.A$, respectively): in the former, 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 [10]. Dually, for type output we require the existence of a pair of output types and candidate relation such that the continuation processes are related. The power of this style of logical relation comes from 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 2.4 (Properties of Logical Equivalence [5]).
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* $\Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A$*.*

3 F-Algebras in Polymorphic Sessions

This section shows how to interpret diagrams and type operators in Polyπ (§ 3.2), which are then used to represent inductive (§ 3.3) and coinductive (§ 3.4) session types.

In order to present our encoding of inductive and coinductive sessions through session polymorphism (§ 3.3 and § 3.4), we first summarise the standard interpretation of inductive and coinductive types as algebras of a type operator (§ 3.1). We then show how to interpret diagrams, type operators (and their associated functorial action) at the level of the session calculus (§ 3.2), which enables us to carry out our main development.

3.1 Inductive and Coinductive Session Types

The study of polymorphism in the λ-calculus [1,11,23,4] has shown that parametric polymorphism is expressive enough to encode both inductive and coinductive types in a very precise sense, via a faithful representation of initial and final (co)algebras [16] which does not require extending the syntax nor the semantics of the calculus.

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). The introduction of recursive behaviours in the logical-based session typing framework has been addressed through the introduction of explicit inductive and coinductive session types [28,15] and the corresponding process constructs, preserving the good properties of the framework such as strong normalisation and absence of deadlocks.

Given the logical foundation of the polymorphic session calculus it is natural to wonder if polymorphic sessions are powerful enough to represent inductive and coinductive behaviours in a systematic way.

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

An algebra of a type operator F is a pairing of a type X and a function $f : F(X) \to X$. For an algebra (X, f) to be initial, it must be the case that for any other algebra (X', f') there must exist a *unique* function $h : X \to X'$ such that $h \circ f = f' \circ$

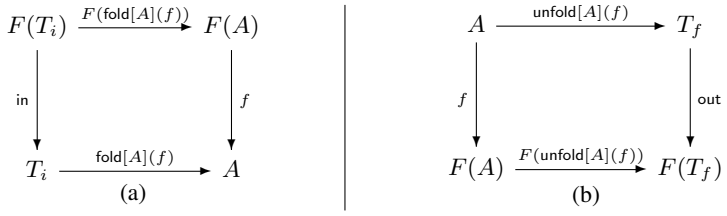

Fig. 3: Diagrams for initial F-algebras and final F-coalgebras

$F(h)$, where $F(h)$ stands for the functorial action of F applied to h. 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 type), 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 faithfully 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 the product type. Final coalgebras consist of a pair (X, f) where X is a type and $f : X \to F(X)$ such that for any other coalgebra (X', f') there exists a *unique* $h : X' \to X$ satisfying $f \circ h = F(h) \circ f'$. In the remainder of this section we assume the positivity requirement on F mentioned above. While the complete formal development of the representation of inductive and coinductive types in System F would lead us to far astray, we summarise here the key concepts as they apply to the λ-calculus (the interested reader can refer to [11] 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. 3(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 uniqueness property is often dubbed *initiality* or *universality*.

The construction of final F-coalgebras and their justification as *greatest* fixed points is dual. Assuming the existence of 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. 3(b) commutes and unfold is unique up to parametricity.

3.2 Diagrams, Functors and Functorial Action

In order to represent diagrams in our session framework, we have that the nodes are *session types* and the arrows are session-typed *processes* that are able to map a session of a given type to one of another. However, given that typing assigns session types to *channels* and not processes, a session transformer from A to B is parameterised by the channel that offers the A behaviour and the channel along which B is to be offered.

Thus, with a pair of types $x{:}A$ and $z{:}B$ and a process P of type $x{:}A \vdash P :: z{:}B$, we can schematically represent the arrow from $x{:}A$ to $z{:}B$ as $x{:}A \xrightarrow{P} z{:}B$. Given another process Q such that $z{:}B \vdash Q :: y{:}A$, we can represent arrow composition as the diagram of Fig. 4, denoting that we can transform a session $x{:}A$ into $y{:}A$ by either "following the diagram" along the P and Q arrows or we may compose (i.e. cut) the two arrows outright to form $(\nu z)(P \mid Q)$.

We note that the diagram in Fig. 4 commutes up-to structural congruence, since composition of P and Q is structurally equivalent to $(\nu x)(P \mid Q)$. This is in contrast with the diagram in Fig. 5, where we must show that $(\nu z)(P \mid Q) = [x \leftrightarrow y]$, for which $\equiv$ is insufficient, but we can use $\approx_{\mathtt{L}}$ of § 2.2 as $(\nu z)(P \mid Q) \approx_{\mathtt{L}} [x \leftrightarrow y]$. Hence, unless otherwise stated, we assume that equality in diagrams is $\approx_{\mathtt{L}}$.

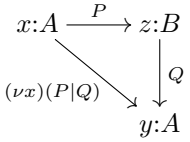

Fig. 4

Type Operators. Type operators, or *functors*, can be seen as transformations on (session) types, in the sense that given some type A and a type operator $X.B$ you can apply the operator to A to produce the type $B\{A/X\}$. For instance, the type operator $X.\mathbf{1} \multimap (X \otimes \mathbf{1})$ can be applied to some well-formed type A to produce the session type $\mathbf{1} \multimap (A \otimes \mathbf{1})$.

As is the case in functional generic programming, a key feature of type operators as functors is that they act as type-directed transformations in a very precise sense: given a type that has the same "shape" of the type operator, with some instantiation of the type variable (e.g. for the operator $X.\mathbf{1} \multimap (X \otimes \mathbf{1})$, the type $\mathbf{1} \multimap (A \otimes \mathbf{1})$, for some A), the functorial action of the type operator allows us to *automatically* transform a program (in our case, a session) of type $\mathbf{1} \multimap (A \otimes \mathbf{1})$ into one of, say, type $\mathbf{1} \multimap (B \otimes \mathbf{1})$ by simply providing a process that coerces a session of type A into one of type B – the functorial action takes care of all the additional boilerplate.

In a functional setting, the functorial action of a type operator is formalised as a structure preserving map, defined inductively on the structure of the type operator, such that $\mathsf{map}_{X.F(X)}(\lambda x.M)(N)$ takes a function $\lambda x.M$ of type $A \to B$ and a (to be transformed) term N of type $F(A)$, producing a term of type $F(B)$. In our session-typed setting, the natural representation $\mathsf{map}_{X.F(X)}$ must consist of a type directed *process transformation* that given a session $F(A)$ and a way to transform sessions of type A into sessions of type B produces a session $F(B)$. That is, the map construction acts as a *modular* adapter that given a process that transforms protocols of type A to protocols of type B, can perform the adaptation from A to B compositionally in any protocol where A occurs as a *sub-protocol*.

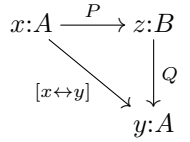

Fig. 5

Given that our typing judgment is of the form $\Omega; \Gamma; \Delta \vdash P :: z{:}A$, where $\{z\} \cap dom(\Delta) \cap dom(\Gamma) = \emptyset$, denoting that P *offers* a session of type A along channel z when composed with processes that offer the (linear) session channels specified in Δ and the (shared) session channels specified in Γ, our session transformers must be *parameterized* with the appropriate channel names. To account for the full generality of positive type operators and their associated *covariant* and *contravariant* functorial actions we define the two map processes, $\mathsf{map}^{+}_{z:X.A}(Q_{x,y})(z')$ and $\mathsf{map}^{-}_{z:X.A}(Q_{x,y})(z')$. The former applies to covariant functors, taking a session z' of type $A\{B'/X\}$ into $A\{B/X\}$ (offered along z) via the transformation process $Q_{x,y}$ (typed as $x{:}B' \vdash Q :: y{:}B$). The latter applies to contravariant functors, taking a session z' of type $A\{B/X\}$ into $A\{B'/X\}$.

Definition 3.1 (Functorial Action of a Type Operator). *Given a type operator $X.A$ and process $x{:}B' \vdash Q_{x,y} :: y{:}B$, we define its covariant and contravariant functorial action $\mathsf{map}^{+}_{z:X.A}(Q_{x,y})(z')$ and $\mathsf{map}^{-}_{z:X.A}(Q_{x,y})(z')$ inductively on the structure of $X.A$, where map^{+} applies to positive operators and map^{-} to negative ones, as follows (we write $\mathsf{map}^{\dagger}$ for either map^{+} or map^{-} and $\overline{\dagger}$ to stand for the reverse of $\dagger$):*

$$
\begin{array}{ll}
\mathsf{map}^{+}_{z:X.Y}(Q_{x,y})(z') & \triangleq [z' \leftrightarrow z] \quad (Y \neq X) \\
\mathsf{map}^{\dagger}_{z:X.\mathbf{1}}(Q_{x,y})(z') & \triangleq [z' \leftrightarrow z] \\
\mathsf{map}^{+}_{z:X.X}(Q_{x,y})(z') & \triangleq (\nu y)(Q\{z'/x\} \mid [y \leftrightarrow z]) \\
\mathsf{map}^{\dagger}_{z:X.A\otimes B}(Q_{x,y})(z') & \triangleq z'(a').\overline{z}\langle a\rangle.(\mathsf{map}^{\dagger}_{a:X.A}(Q_{x,y})(a') \mid \mathsf{map}^{\dagger}_{z:X.B}(Q_{x,y})(z')) \\
\mathsf{map}^{\dagger}_{z:X.A\multimap B}(Q_{x,y})(z') & \triangleq z(a).\overline{z}'\langle a'\rangle.(\mathsf{map}^{\overline{\dagger}}_{a:X.A}(Q_{x,y})(a') \mid \mathsf{map}^{\dagger}_{z:X.B}(Q_{x,y})(z')) \\
\mathsf{map}^{\dagger}_{z:X.\forall Y.A}(Q_{x,y})(z') & \triangleq z(Y).z'\langle Y\rangle.\mathsf{map}^{\dagger}_{z:X.A}(Q_{x,y})(z') \\
\mathsf{map}^{\dagger}_{z:X.\exists Y.A}(Q_{x,y})(z') & \triangleq z'(Y).z'\langle Y\rangle.\mathsf{map}^{\dagger}_{z:X.A}(Q_{x,y})(z')
\end{array}
$$

Lemma 3.2. *For all type operators $X.F(X)$, well-formed types B, B' and $x{:}B' \vdash Q :: y{:}B$:*

- *If $X.F(X)$ is positive and $\Omega; \Gamma; \Delta \vdash P :: w{:}F(B')$ then $\Omega; \Gamma; \Delta \vdash (\nu w)(P \mid \mathsf{map}^{+}_{z:X.F(X)}(Q_{x,y})(w)) :: z{:}F(B)$.*
- *If $X.F(X)$ is negative and $\Omega; \Gamma; \Delta \vdash P :: w{:}F(B)$, then $\Omega; \Gamma; \Delta \vdash (\nu w)(P \mid \mathsf{map}^{-}_{z:X.F(X)}(Q_{x,y})(w)) :: z{:}F(B')$.*

Functor Laws. We prove that our development of functors is *canonical* by showing that the *functor laws* are preserved by map. Specifically, map preserves identity and composition up to $\approx_{\mathtt{L}}$.

Theorem 3.3 (Functor Laws). *For any type operator $X.A$:*
Identity: *If $\Omega; \Gamma; \Delta \vdash P :: z' : A\{B/X\}$ then*

$$\Omega; \Gamma; \Delta \vdash (\nu z')(P \mid \mathsf{map}^{\dagger}_{z:X.A}([x \leftrightarrow y])(z')) \approx_{\mathtt{L}} (\nu z')(P \mid [z' \leftrightarrow z]) :: z{:}A\{B/X\}$$

Associativity: *Let $\Omega; \Gamma; x{:}B_1 \vdash P :: a{:}B_2$, $\Omega; \Gamma; a{:}B_2 \vdash Q :: y{:}B_3$ and $\Omega; \Gamma; \Delta \vdash R :: z'{:}A_0$. If $X.A$ is positive then:*

$$\begin{array}{c}\Omega; \Gamma; z'{:}A\{B_1/X\} \vdash \mathsf{map}^{+}_{z:X.A}((\nu a)(P \mid Q)_{x,y})(z') \approx \\ (\nu z'')(\mathsf{map}^{+}_{z'':X.A}(P_{x,a})(z') \mid \mathsf{map}^{+}_{z:X.A}(Q_{a,y})(z'')) :: z{:}A\{B_3/X\}\end{array}$$

If $X.A$ is negative then

$$\Omega; \Gamma; z'{:}A\{B_3/X\} \vdash \mathsf{map}^{-}_{z:X.A}((\nu a)(P \mid Q)_{x,y})(z') \approx (\nu z'')(\mathsf{map}^{-}_{z'':X.A}(Q_{a,y})(z') \mid \mathsf{map}^{-}_{z:X.A}(P_{x,y})(z'')) :: z{:}A\{B_1/X\}$$

Henceforth we write map to stand for map^{+}.

3.3 Encoding Inductive Types

An inductive type $\mu_i X.F(X)$, where $X.F(X)$ is a positive type operator is encoded as $[\![\mu_i X.F(X)]\!] \triangleq \forall X.!(F(X) \multimap X) \multimap X$. As discussed in § 3.1, inductive types can be understood as initial F-algebras. Let $T_i = [\![\mu_i X.F(X)]\!]$. We must show that the F-algebra $(T_i, \alpha_{x,y})$, with $x{:}F(T_i) \vdash \alpha_{x,y} :: y{:}T_i$, is initial.

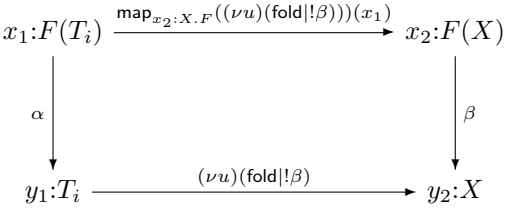

Fig. 6

Thus, we must show that the diagram of Fig. 6 commutes and that fold is unique (we write β_{x_2,y_2} for a process $x_2{:}F(X) \vdash \beta_{x_2,y_2} :: y_2{:}X$ and $!\beta$ for $!u(y_2).y_2(x_2).\beta_{x_2,y_2}$). We define fold_{y_1,y_2} and $\alpha_{x,y}$ as follows:

$$\begin{aligned} \mathsf{fold}_{y_1,y_2} &\triangleq y_1\langle X\rangle.\overline{y_1}\langle b\rangle.(!b(w).\overline{u}\langle c\rangle.[c \leftrightarrow w] \mid [y_1 \leftrightarrow y_2]) \\ \alpha_{x_1,y_1} &\triangleq y_1(X).y_1(u).\overline{u}\langle f\rangle.\overline{f}\langle b\rangle(\mathsf{map}_{b:X.F}(\mathsf{fold}_{y_1,y_2})(x_1) \mid [f \leftrightarrow y_1]) \end{aligned}$$

where $X; u{:}F(X) \multimap X; y_1{:}T_i \vdash \mathsf{fold}_{y_1,y_2} :: y_2{:}X$ and $\cdot;\cdot; x_1{:}F(T_i) \vdash \alpha_{x_1,y_1} :: y_1{:}\forall X.!(F(X) \multimap X) \multimap X$.

To show that Fig. 6 commutes, we prove the observational equivalence stated in Theorem 3.7, using the three lemmas below.

Lemma 3.4. *Let $X.A$ be a positive type operator. If $\Omega; \Gamma, u{:}B; x{:}B_1 \vdash P :: y{:}B_2$ and $\Omega; \Gamma; \cdot \vdash Q :: w{:}B$ then:*

$$\Omega; \Gamma; z'{:}A\{B_1/X\} \vdash \mathsf{map}_{z:X.A}((\nu u)(!u(w).Q \mid P))(z') \approx_{\mathtt{L}} (\nu u)(!u(w).Q \mid \mathsf{map}_{z:X.A}(P)(z')) :: z{:}A\{B_2/X\}$$

Lemma 3.5. *For any $\Omega; \Gamma; \cdot \vdash P :: x{:}A$ we have that:*

$$\Omega; \Gamma; \cdot \vdash (\nu u)(!b(w).\overline{u}\langle c\rangle.[c \leftrightarrow w] \mid !u(y_2).P) \approx_{\mathtt{L}} !b(y_2).P :: b{:}!A$$

Lemma 3.6. *Let $\Omega; \Gamma; \cdot \vdash U_1 \approx_{\mathtt{L}} U_2 :: x{:}A$ and $\Omega; \Gamma; \cdot \vdash R_1 \approx_{\mathtt{L}} R_2 :: u{:}!A$ then: (1) $\Omega; \Gamma; \cdot \vdash !u(x).U_1 \approx_{\mathtt{L}} R_1 :: u{:}!A$; (2) $\Omega; \Gamma; \cdot \vdash !u(x).U_2 \approx_{\mathtt{L}} R_2 :: u{:}!A$*

Theorem 3.7 (Weak Initiality). *Let $X.F(X)$ be a positive type operator, then for any type X and morphism β such that $x_2{:}F(X) \vdash \beta_{x_2,y_2} :: y_2{:}X$ we have that:*

$$X;\cdot; x_1{:}F(T_i) \vdash (\nu y_1)(\alpha_{x_1,y_1} \mid (\nu u)(\mathsf{fold}_{y_1,y_2} \mid !\beta)) \approx_{\mathtt{L}} (\nu x_2)(\mathsf{map}_{x_2:X.F}((\nu u)(\mathsf{fold}_{y_1,y_2} \mid !\beta))(x_1) \mid \beta_{x_2,y_2}) :: y_2{:}X$$

Initiality. Having shown that the diagram of Fig. 6 commutes, we have that the F-algebra (T_i, α) is weakly initial in the sense that there exists a morphism (constructable with fold) from it to any other such F-algebra. We now show that the algebra is indeed initial. In other words, we show that fold is the *only* morphism (up-to $\approx_{\mathrm{L}}$) that makes the diagram of Fig. 6 commute.

Theorem 3.8 (Initiality of T_i). *Let $X.F$ be a positive type operator. For any F-algebra (X, β), we have that for all H such that $X; u{:}F(X) \multimap X; y_1{:}T_i \vdash H :: y_2{:}X$ the following holds: $X; u{:}F(X) \multimap X; y_1{:}T_i \vdash H \approx_{\mathrm{L}} \mathsf{fold}_{y_1,y_2} :: y_2{:}X$.*

Proof. We only sketch the key elements of the proof. The proof of initiality requires showing an equivalence at the (open) type variable X. In particular, we must show that for all pairs of closed types (and respective admissible relations) that may instantiate X, fold and H are equivalent. By making crucial use of parametricity (i.e. that fold and H are equivalent to themselves for any such admissible relations) we can construct an admissible relation that allows us to discharge the main proof obligation.

3.4 Encoding Coinductive Types

We perform a similar development to that of § 3.3 but for coinductive types. A coinductive type $\mu_f X.F(X)$, where $X.F(X)$ is a positive type operator is encoded as: $[\![\mu_f X.F(X)]\!] \triangleq \exists X.!(X \multimap F(X)) \otimes X$. Coinductive types are interpreted as final F-coalgebras. Let $T_f = [\![\mu_f X.F(X)]\!]$, we must show that the F-coalgebra $(T_f, \alpha_{x,y})$, with $x{:}T_i \vdash \alpha_{x,y} :: y{:}F(T_i)$ is final. As before, we show that the diagram of Fig. 7 commutes and that unfold is unique (we write β for a process $x_1{:}X \vdash \beta_{x_1,y_1} :: y_1{:}F(X)$ and $!\beta$ as before). We define $\mathsf{unfold}_{x_1,x_2}$ and $\alpha_{x,y}$ as:

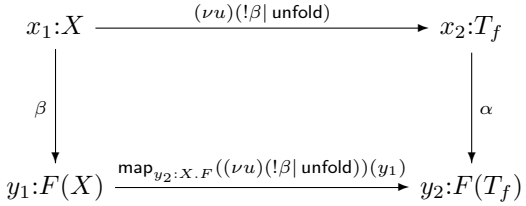

Fig. 7

$$\begin{aligned}\mathsf{unfold}_{x_1,x_2} &\triangleq x_2\langle X\rangle.\overline{x_2}\langle c\rangle.(!c(v).\overline{u}\langle c'\rangle.[c' \leftrightarrow v] \mid [x_1 \leftrightarrow x_2])\\ \alpha_{x_2,y_2} &\triangleq x_2(X).x_2(u).\overline{u}\langle c\rangle.\overline{c}\langle c'\rangle.([x_2 \leftrightarrow c'] \mid \mathsf{map}_{y_2:X.F}(\mathsf{unfold}_{x_1,x_2})(c))\end{aligned}$$

where $X; u{:}X \multimap F(X); x_1{:}X \vdash \mathsf{unfold}_{x_1,x_2} :: x_2{:}T_f$ and $\cdot;\cdot; x_2{:}T_f \vdash \alpha_{x_2,y_2} :: y_2{:}F(T_f)$. We show that diagram above commutes by proving Lemma 3.9.

Theorem 3.9 (Weak Finality). *Let $X.F(X)$ be a positive type operator. For any type X and morphism β such that $x_1{:}X \vdash \beta_{x_1,y_1} :: y_1{:}F(X)$ we have that: $X; \cdot; x_1{:}X \vdash$*

$$\begin{aligned}&(\nu x_2)((\nu u)(!\beta \mid \mathsf{unfold}_{x_1,x_2}) \mid \alpha_{x_2,y_2})\\ &\quad\approx_{\mathrm{L}} (\nu y_1)(\beta_{x_1,y_1} \mid \mathsf{map}_{y_2:X.F}((\nu u)(!\beta \mid \mathsf{unfold}))(y_1)) :: y_2{:}F(T_f)\end{aligned}$$

Finality To show the universality of $(T_f, \alpha_{x,y})$ we establish a dual property to Theorem 3.8, proving that unfold is the *unique* morphism for which the diagram of Fig. 7 commutes. However, unlike in the development of Theorem 3.8, the existential flavour of the encoding requires a slightly more intricate formal argument.

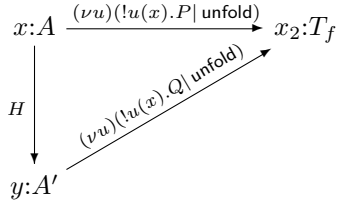

Fig. 8

We first establish the following two equivalences: Lemma 3.10 states that using unfold to produce a session morphism from A to T_f is equivalent to first using a coalgebra morphism H from A to A' and then using unfold at type A' to ultimately produce a session of type T_f. The proof of this lemma is more challenging than the development of § 3.3 due to the encoding using an existentially quantified type (i.e., parametricity provides a less general property). The proof showcases an instance of a particularly powerful style of reasoning that is not usually available in a process setting. We establish the result by performing the *relational lifting* of *process* H, viewed as a relation between A and A', which is the *graph relation* of H (i.e. $(d, d') \in \mathcal{R}$ iff $d_y \cong (\nu x)(d'_x \mid H_{x,y})$); Lemma 3.11 shows that unfolding the coalgebra morphism α is an identity at type T_f. The proof of Lemma 3.10 is the most intricate constracting a graph relation between A and A' with process H. The corresponding diagram is given in Fig. 8. These two lemmas conclude the finality result (Theorem 3.12).

Lemma 3.10. *Let $X.F$ be a positive type operator. Let $x{:}A \vdash H :: y{:}A'$ be a morphism of coalgebras from $P :: x{:}A \multimap F(A)$ to $Q :: x{:}A' \multimap F(A')$. The equivalence holds:*

$$\begin{aligned} x{:}A \vdash (\nu y)(H \mid (\nu u)(!u(x).Q \mid \mathsf{unfold}_{y,x_2}\{A'/X\})) \\ \approx_{\mathtt{L}} (\nu u)(!u(x).P \mid \mathsf{unfold}_{x,x_2}\{A/X\}) :: x_2{:}T_f \end{aligned}$$

Proof. Since we must establish an equivalence at an existential type, we need to exhibit one particular type instantiation of the existential variable (and respective admissible relation). Noting that reasoning directly from the parametricity of H and unfold does not enable us to fulfill this proof obligation, we proceed by lifting the morphism H to the relational level, showing that the two processes are equivalent when we consider the *graph* relation of the coalgebra morphism explained above (relating processes that are observationally equivalent when one of them is composed with H). By appealing to this lifting via parametricity we conclude using the coalgebraic properties of H.

Lemma 3.11. $x{:}T_f \vdash (\nu u)(!u(x).x(y).\alpha_{y,x} \mid \mathsf{unfold}_{x,z}\{T_f/X\}) \approx_{\mathtt{L}} [x \leftrightarrow z] :: z{:}T_f$

Theorem 3.12 (Finality of T_f). *Let $X.F$ be a positive type operator, $x{:}A \vdash P_{x,y} :: y{:}F(A)$ an F-coalgebra and $x{:}A \vdash H_{x,y} :: y{:}T_f$ be a coalgebra map from P to α:*

$$\cdot;\cdot; x{:}A \vdash H_{x,x_2} \approx_{\mathtt{L}} (\nu u)(!u(y).y(x).P_{x,y} \mid \mathsf{unfold}\{A/X\}) :: x_2{:}T_f$$

Proof. The proof combines the relational lifting of the coalgebra morphism H (Lemma 3.10) and Lemma 3.11 to discharge the main proof obligation.

Remark 3.13 (Relational Lifting). The development of the encoding of coinductive types enables us to exhibit the full expressive power of parametric equivalence for typed processes, allowing us to reason about process equivalences at *different* types. This is made explicit in the proof of Lemma 3.10, where we use parametric equivalence to perform a relational lifting of a process (i.e. a coalgebra process morphism between A and

A'): the two given processes are equivalent when the one on the left-hand side instantiates the existential type with A and the one on the right with A', given that the relation between the types is precisely the composition with the coalgebra process morphism on the right-hand side of the equivalence (i.e. the relational lifting of the morphism).

4 Conclusion

This work has explored a *processes-as-morphisms* approach to demonstrate the expressiveness of polymorphic mobile processes, showing an encoding of inductive and coinductive session types, justified via the theory of initial algebras and final co-algebras. Our work gives a direct account of encodability comparing to the method in [29] which *indirectly* proves an identical result using fully abstract encodings from/to linear System F [33]. Our work crucially uses a rigorous proof-theoretic correspondence between (intuitionistic) linear-logic and session types. The notion of "expressiveness" in our paper differs from the standard (positive and negative) encodability criteria defined by e.g. Palamidessi [19,20] for the π-calculus. Our hope is that this new algebraic encodability will potentially enable an exploration of algebraic constructs beyond initial and final co-algebras in a session programming setting. We wish to further study the meaning of functors and natural transformations in a logic-based session-typed setting, both from a more fundamental viewpoint but also in terms of practical programming patterns. This also relates to Miller's work on logic programming languages [17,12,18] where a proof-theoretic foundation based on (intuitionistic) logic gives a uniform, clear basis for (logic) programming languages.

Acknowledgements. The authors would like to thank Dominic Orchard and the anonymous reviewers for their comments and suggestions. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1 and NOVA LINCS (UID/CEC/04516/2019).

References

1. E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theor. Comput. Sci.*, 70(1):35–64, 1990.
2. R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, Inc., 1997.
3. R. S. Bird, O. de Moor, and P. F. Hoogendijk. Generic functional programming with types and relations. *J. Funct. Program.*, 6(1):1–28, 1996.
4. L. Birkedal, R. E. Møgelberg, and R. L. Petersen. Linear Abadi and Plotkin Logic. *Logical Methods in Computer Science*, 2(5), 2006.
5. L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP 2013*, pages 330–349, 2013.
6. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, pages 222–236, 2010.
7. L. Caires, F. Pfenning, and B. Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.
8. J. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In *Proc. of the 2nd Scandinavian Logic Symposium*, pages 63–92, 1971.

9. J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
10. J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
11. R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 1994.
12. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110:327–365, 1994.
13. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, pages 122–138, 1998.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.
15. S. Lindley and J. G. Morris. Talking bananas: structural recursion for session types. In *ICFP 2016*, pages 434–447, 2016.
16. N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.
17. D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 09 1991.
18. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
19. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous &pgr;-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 256–265, New York, NY, USA, 1997. ACM.
20. C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical. Structures in Comp. Sci.*, 13(5):685–719, Oct. 2003.
21. J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations for session-based concurrency. In *ESOP 2012*, pages 539–558, 2012.
22. B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3):531–584, 2000.
23. G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA '93*, pages 361–375, 1993.
24. J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
25. D. Sangiorgi. Pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.
26. D. Sangiorgi and D. Walker. *The pi-calculus: A theory of mobile processes*. Cambridge University Press, 2001.
27. B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FOSSACS 2012*, pages 346–360, 2012.
28. B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC 2014*, pages 159–175, 2014.
29. B. Toninho and N. Yoshida. On Polymorphic Sessions And Functions: A Tale of Two (Fully Abstract) Encodings. In *27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 827–855. Springer, 2018.
30. B. Toninho and N. Yoshida. Polymorphic session processes as morphisms. Technical Report 02/2019, Department of Computing, Imperial College London, July 2019. `https://www.doc.ic.ac.uk/research/technicalreports/2019/DTRS19-2.pdf`.
31. P. Wadler. Recursive types for free! Unpublished draft available at `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.
32. P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.
33. J. Zhao, Q. Zhang, and S. Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In *APLAS*, pages 344–359, 2010.