Behavioral Polymorphism and Parametricity in Session-Based Communication

Luís Caires[1], Jorge A. Pérez[1], Frank Pfenning[2], and Bernardo Toninho[1,2]

[1] CITI and Departamento de Informática, FCT, Universidade Nova de Lisboa
[2] Computer Science Department, Carnegie Mellon University

Abstract. We investigate a notion of behavioral genericity in the context of session type disciplines. To this end, we develop a logically motivated theory of parametric polymorphism, reminiscent of the Girard-Reynolds polymorphic λ-calculus, but casted in the setting of concurrent processes. In our theory, polymorphism accounts for the exchange of abstract communication protocols and dynamic instantiation of heterogeneous interfaces, as opposed to the exchange of data types and dynamic instantiation of individual message types. Our polymorphic session-typed process language satisfies strong forms of type preservation and global progress, is strongly normalizing, and enjoys a relational parametricity principle. Combined, our results confer strong correctness guarantees for communicating systems. In particular, parametricity is key to derive non-trivial results about internal protocol independence, a concurrent analogous of representation independence, and non-interference properties of modular, distributed systems.

1 Introduction

Modern distributed systems are typically conceived as decentralized collections of software artifacts which execute intricate communication protocols. These large-scale systems must meet strict correctness and trustworthiness requirements. Emerging technologies—such as service-oriented computing and subscription-based, cost-sharing platforms (e.g. cloud computing)—promise to be effective towards achieving these goals, while reducing costs and enhancing business agility. They also pose new challenges for system construction: communicating systems should behave properly even when deployed in open, highly dynamic environments, such as third-party infrastructures.

In this communication-oriented context, *genericity*—one of the fundamental principles in software engineering—turns out to be a most relevant concern. Indeed, genericity promotes modular protocol specifications, therefore facilitating system verification and evolution/maintenance. It allows for convenient represention of, for instance, families of protocols which differ only in the format of the exchanged messages (as in, e.g., protocols for file distribution which behave correctly independently of the transferred items). This "message genericity" is most useful and appears to be well-understood.

Nevertheless, and partly due to the widespread adoption of technologies such as those hinted at above, distributed systems nowadays exhibit fairly sophisticated incarnations of genericity, which often go well beyond message genericity. Indeed, systems are increasingly generic with respect to *arbitrary communication protocols*, which may be known and instantiated only at runtime. Here we refer to this kind of genericity as *behavioral genericity*; we find it to be a very common concept in several settings:

– *Critical web applications* (such as banking portals) are increasingly being deployed into service-oriented architectures. As such, upgrade actions (e.g., replacing a service provider) often involve the dynamic reconfiguration of communication interfaces/protocols. These changes should be transparent to clients. To this end, web applications should be conceived as generic with respect to such interfaces/protocols.
– *Online application stores* are infrastructures for the distribution of software applications. They should concurrently interact with (i) *developers* willing to add new (i.e. unknown) applications to the store and (ii) *clients* wishing to remotely execute/buy/download available applications. In order to operate securely and reliably, the store needs to be generic with respect to the behavior of clients and applications.
– *Cloud-based services* admit highly dynamic, flexible architectures. In fact, these services are *elastic*, for they acquire computing resources when demand is high, and release them when they are no longer needed. For such scaling policies to be effective, services need to be generic with respect to their underlying coordination protocols, as these may well depend on the system's architecture at a given time.

Many other distributed software systems exhibit forms of behavioral genericity in the context of disciplined, structured communications. Reasoning about these systems and their correctness is extremely hard, essentially because the required abstractions should enforce independence with respect to arbitrary complex behaviors, and not just over messages. Models and techniques for data/message genericity are thus simply inadequate for this task. This calls for novel reasoning techniques, which may effectively support the analysis of behavioral genericity in complex distributed protocols.

Here we rise to this challenge in the context of *session-based concurrency* [17,18], a foundational approach to communication correctness. In session-based concurrency, dialogues between participants are structured into *sessions*, the basic units of communication; interaction patterns are abstracted as *session types*, which are statically checked against specifications. Session types ensure protocols in which actions always occur in dual pairs: when one partner sends, the other receives; when one partner offers a selection, the other chooses; when a session terminates, no further interaction may occur.

In this paper, we develop a session types discipline able to cope with behavioral genericity. Our system includes *impredicative* universal and existential quantification over *sessions*: this results in *parametric polymorphism*—in the sense of the Girard-Reynolds polymorphic λ-calculus [23,13]—defined in a session-based, concurrent setting. In our theory, universal and existential quantification correspond to the input and output of a session type, respectively. As session types may describe arbitrarily complex communication protocols, our theory of polymorphic processes enables an expressive form of *abstract protocol communication*. As a key distinguishing feature, our developments follow naturally from the interpretation of session types as *intuitionistic* linear logic propositions given in [6,7]. This allows us to obtain central technical results for polymorphic, session-typed processes in a remarkably elegant way:

1. Polymorphic processes respect session typed specifications in a deadlock-free way. These two central—and non trivial—correctness guarantees follow from our *type preservation* and *global progress* results (Theorems 1 and 2).
2. Polymorphic processes never engage into infinite internal behavior. In fact, well-typed processes are *strongly normalizing* (Theorem 5). The proof of this important

(and arguably expected) result is via the reducibility candidates technique, by relying on an elegant generalization of the linear logical relations of [20].

3. Polymorphic processes enjoy a principle of *relational parametricity* in the context of a behavioral type theory (Theorem 8). In Section 6, we illustrate how parametricity allows us to formally justify properties of behavioral genericity and representation independence, which in our case means behavioral independence on representation protocols. Parametricity also enables a sound and complete characterization of *typed contextual equivalence* (Theorem 9).

To our knowledge, relational parametricity (in the sense of Reynolds [24]) has not been previously investigated in the context of a rich behavioral type theory for processes, such as session types. In the realm of concurrent processes, genericity via (existential) polymorphism was first investigated by Turner [27], in the context of a simply-typed π-calculus. Berger et al. [1,2] were the first to study a π-calculus with parametric polymorphism based on universal and existential quantification over types. In the setting of session types, support for genericity has been obtained mainly via *bounded polymorphism* [12,10,9], which extends session types with a form of (universal) quantification over types, controlled via subtyping. While useful to reason about protocols with message genericity, bounded polymorphism is insufficient to support behavioral genericity. Recently, Wadler [28] proposed a logic-based session type theory which includes the natural typing rules for second-order quantifiers and may support polymorphism of the kind we consider here; however, no analysis of behavioral genericity is identified. Our results thus provide substantial evidence of how a logically motivated approach offers appropriate, powerful tools for actually reasoning about behavioral genericity in complex protocols. In passing, we establish rather strong connections between well-known foundational results and polymorphically typed concurrent processes.

In the remainder of this introduction, we briefly describe the logical interpretation of [6] and illustrate the potential of our model of polymorphic sessions with an example. Our ongoing research program on logical foundations for session-based concurrency [6,26,21,7,20,8] builds upon an interpretation of intuitionistic linear logical propositions as session types, sequent proofs as π-calculus processes [25], and cut elimination as process communication. In the resulting Curry-Howard correspondence, well-typed processes enjoy strong forms of type preservation and global progress [6,7], and are strongly normalizing [20]. The interpretation endows channel names with types (logic propositions) that describe their session protocol. This way, e.g., an assignment $x{:}A{\multimap}B$ denotes a session x that first *inputs* a name of type A, and then behaves as type B on x; dually, $x{:}A \otimes B$ denotes a session x that first *outputs* a name of type A and then behaves as type B on x. Other constructors are given compatible interpretations; in particular, $!A$ is the type of a shared server offering sessions of type A. Given a *linear* environment Δ and an *unrestricted* environment Γ, a type judgment in our system is of the form $\Gamma; \Delta \vdash P :: z{:}C$, where Γ, Δ, and $z{:}C$ have pairwise disjoint domains. Such a judgment is intuitively read as: process P offers session C along channel z, provided it is placed in a context providing the sessions declared in Γ and Δ.

Here we uniformly extend the system of [6] with two new kinds of session types, $\forall X.A$ and $\exists X.A$, corresponding to impredicative universal and existential quantification over sessions. As mentioned above, they are interpreted as the input and output of

a session type, respectively. As an example, consider the polymorphic session type:

$$\mathsf{CloudServer} \triangleq \forall X.!(\mathsf{api} \multimap X) \multimap !X$$

which represents a simple interface for a *cloud-based application server*. In our theory, this is the session type of a system which first *inputs* an arbitrary type (say GMaps); then inputs a shared service of type $\mathsf{api} \multimap \mathsf{GMaps}$. Each instance of this service yields a session that when provided with the implementation of an API will provide a behavior of type GMaps; finally becoming a persistent (shared) server of type GMaps. Our application server is meant to interact with developers who, by building upon the services it offers, implement their own applications. In our framework, the dependency between the cloud server and applications may be expressed by the typing judgment

$$\cdot\,;\, x{:}\mathsf{CloudServer} \vdash \mathit{DrpBox} :: z{:}\mathsf{dbox} \tag{1}$$

Intuitively, (1) says that to offer behavior dbox on z, the file hosting service represented by process DrpBox relies on a linear behavior described by type CloudServer provided on x (no shared behaviors are required). The rôle of behavioral genericity should be clear from the following observation: to support interaction with developers such as DrpBox—which implement all kinds of behaviors, such as dbox above—any process realizing type CloudServer should necessarily be *generic* on such expected behaviors.

The above example illustrates how the combination of polymorphism and linearity enables very fine-grained specifications of interactive behavior via types. Indeed, as just discussed, impredicative quantification enforces that every cloud server implementation must be agnostic to the specific behavior of the actual applications it will provide, whereas linearity allows us to reason precisely about behavior and session usage (e.g., the only way the server can provide the behavior X is by making use of session $\mathsf{api} \multimap X$). In Section 3 we develop this example further, demonstrating how the expressiveness and flexbility of polymorphic session types is captured in process specifications. Then, in Section 6 we illustrate how to exploit parametricity, strong normalization, and other properties of well-typed processes to reason about such specifications. In fact, we show how by merely exploiting the shape of its (polymorphic) type, we are able to analyze the observable behavior of a generic cloud-based server.

For space reasons, most proofs are omitted. An associated technical report [5] gives full technical details, and reports further developments which connect our work with impredicative polymorphism in the functional setting via an encoding of System F.

2 Polymorphic Session Types

We consider a synchronous π-calculus [25] extended with binary guarded choice, channel links, and type input and output prefixes. The syntax of processes/types is as follows:

Definition 1 (Processes, Session Types). *Given an infinite set Λ of* names *(x, y, z, u, v), the set of* processes *(P, Q, R) and session types (A, B, C) is defined by*

$$\begin{array}{rl} P ::= & \overline{x}\langle y\rangle.P \mid x(y).P \mid !x(y).P \mid P \mid Q \mid (\boldsymbol{\nu} y)P \mid \mathbf{0} \\ & \mid \overline{x}\langle A\rangle.P \mid x(X).P \mid x.\mathtt{inl}; P \mid x.\mathtt{inr}; P \mid x.\mathtt{case}(P, Q) \mid [x \leftrightarrow z] \end{array}$$

$$A ::= \mathbf{1} \mid A \multimap B \mid A \otimes B \mid A \,\&\, B \mid A \oplus B \mid !A \mid X \mid \forall X.A \mid \exists X.A$$

The guarded choice mechanism and the channel link construct are as in [6,26,20]. Informally, channel links "re-implement" an ambient session on a different channel name, thus defining a renaming operation (see below). Moreover, channel links allow a simple interpretation of the identity rule. Polymorphism is represented by prefixes for input and output of types, denoting the exchange of abstract communication protocols.

We identify processes up to consistent renaming of bound names, writing $\equiv_\alpha$ for this congruence. We write $P\{^x/_y\}$ for the process obtained from P by capture avoiding substitution of x for y in P, and *fn*(P) for the free names of P. Session types are directly generated from the language of linear propositions. Structural congruence expresses basic identities on the structure of processes, reduction expresses internal behavior of processes, and labeled transitions define interaction with the environment.

Definition 2. Structural congruence *is the least congruence relation generated by the following laws:* $P \mid \mathbf{0} \equiv P$; $P \equiv_\alpha Q \Rightarrow P \equiv Q$; $P \mid Q \equiv Q \mid P$; $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$; $(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$; $x \notin fn(P) \Rightarrow P \mid (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \mid Q)$; $(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$; *and* $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$.

Definition 3. Reduction *$(P \to Q)$ is the binary relation on processes defined by:*

$$\overline{x}\langle y\rangle.Q \mid x(z).P \to Q \mid P\{^y/_z\} \qquad \overline{x}\langle A\rangle.Q \mid x(Y).P \to Q \mid P\{^A/_Y\}$$
$$\overline{x}\langle y\rangle.Q \mid !x(z).P \to Q \mid P\{^y/_z\} \mid !x(z).P \qquad x.\mathtt{inl}; P \mid x.\mathtt{case}(Q,R) \to P \mid Q$$
$$(\nu x)([x \leftrightarrow y] \mid P) \to P\{^y/_x\} \ \ (x \neq y) \qquad x.\mathtt{inr}; P \mid x.\mathtt{case}(Q,R) \to P \mid R$$
$$Q \to Q' \Rightarrow P \mid Q \to P \mid Q' \qquad P \to Q \Rightarrow (\boldsymbol{\nu} y)P \to (\boldsymbol{\nu} y)Q$$
$$P \equiv P',\ P' \to Q',\ Q' \equiv Q \Rightarrow P \to Q$$

A transition $P \xrightarrow{\alpha} Q$ denotes that P may evolve to Q by performing the action represented by label α. In general, 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 $\overline{(\nu z)x\langle z\rangle}$, respectively, and input action $x(y)$. Also, they include labels pertaining to the binary choice construct ($x.\mathsf{inl}$, $\overline{x.\mathsf{inl}}$, $x.\mathsf{inr}$, and $\overline{x.\mathsf{inr}}$), and labels describing output and input of types (denoted $\overline{x\langle A\rangle}$ and $x(A)$, respectively).

Definition 4 (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 $y \notin fn(\alpha)$; 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 rules* (par)*,* (com)*, and* (close).

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

Type System. Our type system assigns session types to process communication channels. Our session type language (cf. Definition 1) corresponds exactly to second-order linear logic, and our typing rules capture this correspondence in a precise way. We define two judgments: $\Omega; \Gamma; \Delta \vdash P :: x{:}A$ and $\Omega \vdash A$ type. Context Ω keeps track of type variables that can be introduced by the polymorphic type constructors; Γ records persistent sessions $u{:}B$, which can be invoked arbitrarily often along channel u; Δ maintains the sessions $x{:}B$ that can be used exactly once on channel x. When empty,

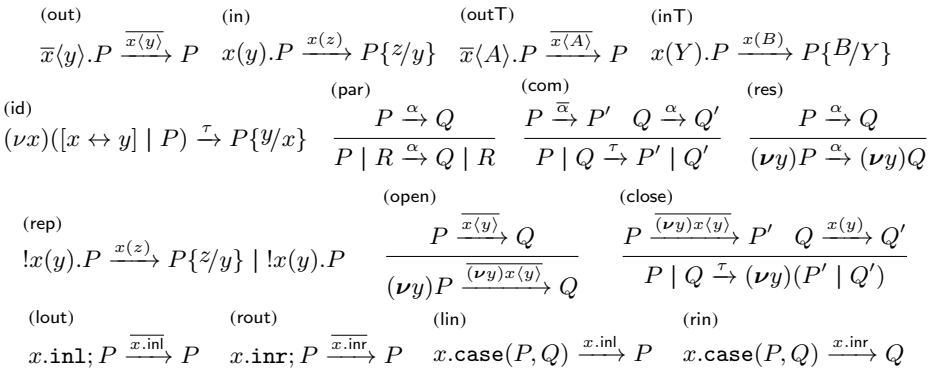

Fig. 1. π-calculus Labeled Transition System.

Γ, Δ, and Ω are often denoted by '$\cdot$'. Judgment $\Omega \vdash A$ type defines well-formedness of types: it denotes that A is a well-formed type with free variables registered in Ω. The rules for type well-formedness are straightforward (see [5]). Our main typing judgment thus states that process P implements a session of type A along channel x, provided it is composed with processes providing sessions linearly in Δ and persistently in Γ, such that the types occurring in the judgment are well-formed according to Ω.

The typing rules for our polymorphic session calculus are given in Fig. 2. We use T, S for right-hand-side singleton environments (e.g., $z{:}C$). Rules pertaining to the propositional fragment extend those introduced in [6] with context Ω. The rules in the last two rows of Fig. 2 explain how to *provide* and *use* sessions of a polymorphic type. More precisely, rule (T$\forall$R) describes the offering of a session of universal type $\forall X.A$ by inputing an arbitrary type, bound to X, and proceeding as A, which may bind the type variable X, regardless of what the actual received type is. Rule (T$\forall$L) says that the use of type $\forall X.A$ consists of the output of a type B—well-formed under type context Ω—which then warrants the use of the session as $A\{^B/_X\}$. The existential type is dual: providing an existentially typed session $\exists X.A$ (cf. rule (T$\exists$R)) is accomplished by outputting a well-formed type B and then providing a session of type $A\{^B/_X\}$. Using an existential session $\exists X.A$ (cf. rule (T$\exists$L)) implies inputing a type and then using the session as A, agnostic to what the actual received type can be. Note that in the presence of polymorphism the identity rule (Tid) (not present in [6,7], but used in [26,21,20]) is necessary, since it is the only way of typing a session with a type variable.

As usual, in the presence of type annotations in binders, type-checking is decidable in our system (these are omitted for readability). We consider π-calculus terms up to structural congruence, and so typability is closed under $\equiv$ by definition. The system enjoys the usual properties of equivariance, weakening, and contraction in Γ, as well as *name coverage* (free names of a process are bound by the contexts or the right-hand-side) and *regularity* (free variables of types are bound in the type variable context).

Correspondence with Second-Order Linear Logic. Our type system exhibits a tight correspondence with a sequent calculus presentation of intuitionistic second-order linear logic. Informally, if we erase the processes and channel names from the typing

$$\text{(Tid)}\ \frac{}{\Omega;\Gamma;x{:}A \vdash [x \leftrightarrow z] :: z{:}A} \qquad \text{(T1L)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: T}{\Omega;\Gamma;\Delta, x{:}\mathbf{1} \vdash P :: T} \qquad \text{(T1R)}\ \frac{}{\Omega;\Gamma;\cdot \vdash \mathbf{0} :: x{:}\mathbf{1}}$$

$$\text{(T}\otimes\text{L)}\ \frac{\Omega;\Gamma;\Delta, y{:}A, x{:}B \vdash P :: T}{\Omega;\Gamma;\Delta, x{:}A \otimes B \vdash x(y).P :: T} \qquad \text{(T}\otimes\text{R)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: y{:}A \quad \Omega;\Gamma;\Delta' \vdash Q :: x{:}B}{\Omega;\Gamma;\Delta,\Delta' \vdash (\nu y)\overline{x}\langle y\rangle.(P \mid Q) :: x{:}A \otimes B}$$

$$\text{(T}\multimap\text{L)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: y{:}A \quad \Omega;\Gamma;\Delta', x{:}B \vdash Q :: T}{\Omega;\Gamma;\Delta,\Delta', x{:}A\multimap B \vdash (\nu y)\overline{x}\langle y\rangle.(P \mid Q) :: T} \qquad \text{(T}\multimap\text{R)}\ \frac{\Omega;\Gamma;\Delta, y{:}A \vdash P :: x{:}B}{\Omega;\Gamma;\Delta \vdash x(y).P :: x{:}A\multimap B}$$

$$\text{(Tcut)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: x{:}A \quad \Omega;\Gamma;\Delta', x{:}A \vdash Q :: T}{\Omega;\Gamma;\Delta,\Delta' \vdash (\nu x)(P \mid Q) :: T} \qquad \text{(Tcut}^!\text{)}\ \frac{\Omega;\Gamma;\cdot \vdash P :: y{:}A \quad \Omega;\Gamma, u{:}A;\Delta \vdash Q :: T}{\Omega;\Gamma;\Delta \vdash (\nu u)(!u(y).P \mid Q) :: T}$$

$$\text{(T!L)}\ \frac{\Omega;\Gamma, u{:}A;\Delta \vdash P\{u/x\} :: T}{\Omega;\Gamma;\Delta, x{:}!A \vdash P :: T} \qquad \text{(Tcopy)}\ \frac{\Omega;\Gamma, u{:}A;\Delta, y{:}A \vdash P :: T}{\Omega;\Gamma, u{:}A;\Delta \vdash (\nu y)\overline{u}\langle y\rangle.P :: T} \qquad \text{(T!R)}\ \frac{\Omega;\Gamma;\cdot \vdash Q :: y{:}A}{\Omega;\Gamma;\cdot \vdash !x(y).Q :: x{:}!A}$$

$$\text{(T}\oplus\text{L)}\ \frac{\Omega;\Gamma;\Delta, x{:}A \vdash P :: T \quad \Omega;\Gamma;\Delta, x{:}B \vdash Q :: T}{\Omega;\Gamma;\Delta, x{:}A \oplus B \vdash x.\mathtt{case}(P,Q) :: T} \qquad \text{(T}\&\text{R)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: x{:}A \quad \Omega;\Gamma;\Delta \vdash Q :: x{:}B}{\Omega;\Gamma;\Delta \vdash x.\mathtt{case}(P,Q) :: x{:}A \,\&\, B}$$

$$\text{(T}\&\text{L}_1\text{)}\ \frac{\Omega;\Gamma;\Delta, x{:}A \vdash P :: T}{\Omega;\Gamma;\Delta, x{:}A \,\&\, B \vdash x.\mathtt{inl}; P :: T} \qquad \text{(T}\oplus\text{R}_1\text{)}\ \frac{\Omega;\Gamma;\Delta \vdash P :: x{:}A}{\Omega;\Gamma;\Delta \vdash x.\mathtt{inl}; P :: x{:}A \oplus B}$$

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

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

Fig. 2. The Type System. Rules (T&L$_2$)-(T$\oplus$R$_2$), analogous to (T&L$_1$)-(T$\oplus$R$_1$), are omitted.

derivations we obtain precisely sequent proofs in intuitionistic second-order linear logic. This correspondence (detailed in [5]) is made precise by defining a faithful proof term assignment for the sequent calculus and a typed extraction function that maps these proof terms to process typing derivations, as reported in [6] for the propositional case.

Notice that the correspondence goes beyond the mapping of proof inferences to typing derivations. We can show that process reductions can be mapped to proof conversions arising from the standard proof-theoretic cut elimination procedure. This induces a strong form of subject reduction on well-typed processes (see below). Furthermore, we can classify *all* proof conversions arising in this manner as reductions, structural congruences, or as observational equivalences on well-typed processes. See [6,20] for details of the correspondence of proof conversions and their process interpretation.

Subject Reduction and Progress. The deep logical foundations allow us to establish strong properties of process behavior through typing. We now discuss and state *subject reduction* and *global progress* for our system. Subject reduction (Theorem 1) follows

from a simulation between reductions in the typed π-calculus and proof conversions that arise naturally in proof theory. This ensures that our interpretation is not arbitrary, but rather captures the actual dynamics of proofs. Subject reduction, together with linear typing, ensures session fidelity; the proof follows closely that of [6,7], extending it with lemmas that characterize process/proof reductions at universal and existential types.

Theorem 1 (Subject Reduction/Type Preservation). *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *and* $P \to Q$ *then* $\Omega; \Gamma; \Delta \vdash Q :: z{:}A$.

As for global progress (Theorem 2), also in this case the proof is an orthogonal extension from that of [6,7], requiring a series of inversion lemmas and the following notion of *live* process. For any P, define $live(P)$ if and only if $P \equiv (\nu \widetilde{n})(\pi.Q \mid R)$, for some process R, a sequence of names $\widetilde{n}$, and a *non-replicated* guarded process $\pi.Q$.

Theorem 2 (Progress). *If* $\cdot; \cdot; \cdot \vdash P :: x{:}\mathbf{1}$ *and* $live(P)$ *then* $\exists Q$ *s.t.* $P \to Q$.

3 The Cloud Application Server, Revisited

To illustrate the expressiveness and flexibility that we obtain via polymorphic sessions, here we present concurrent specifications associated to the cloud-based application server described in the Introduction. Below, for the sake of clarity, we abbreviate bound outputs $(\nu y)\overline{x}\langle y\rangle$ as $\overline{x}\langle y\rangle$. Recall the type for the cloud-based application server: $\mathsf{CloudServer} \triangleq \forall X.!(\mathsf{api} \multimap X) \multimap !X$. Then, following the logic interpretation just introduced, a process which realizes type $\mathsf{CloudServer}$ on name x is the following:

$$CS_x \triangleq x(X).x(y).!x(w).\overline{y}\langle v\rangle.\overline{v}\langle a\rangle.(P_a \mid [w \leftrightarrow v])$$

where P_a is a process implementing the server API along channel a. Process CS_x expects a protocol description X (a session type) and a session y, which is a persistent implementation of X that requires the API provided by the server. CS_x will then create a replicated service that can provide the behavior X after delivering to y the API implementation that is represented by process P_a.

What does an application to be published in the cloud server look like? Let us assume a simple process, noted $Conv_w$, representing a *file conversion service* which, by using a suitable API, takes a file and generates its PDF version (e.g., performing OCR on images and generating the PDF of the text): $a{:}\mathsf{api} \vdash Conv_w :: w{:}\mathsf{file} \multimap (\mathsf{pdf} \otimes \mathbf{1})$.

In order to publish the conversion service into our application server, developers need to harmonize its requirements (as described by the left-hand side typing) with those of the server infrastructure CS_x. To this end, we define a "wrapper" process which contains $Conv_w$ and is compatible with CS_x (where $\mathsf{conv} \triangleq \mathsf{file} \multimap (\mathsf{pdf} \otimes \mathbf{1})$):

$$x{:}\mathsf{CloudServer} \vdash PubConv_z :: z{:}\,!\mathsf{conv}$$
$$PubConv_z \triangleq \overline{x}\langle \mathsf{conv}\rangle.\overline{x}\langle y\rangle.(!y(w).w(a).Conv_w \mid [x \leftrightarrow z])$$

Process $PubConv_z$ first sends protocol/type conv to the cloud server, followed by a session y that consists of a persistent service, that when given the API will produce a session of type conv. After these communication steps, the cloud server session now

provides the full behavior of conv along x, and so the client forwards x along the endpoint channel z, thus providing $!\mathsf{conv}$ along z by making use of the functionality provided by the server. By combining the above processes, we obtain:

$$\vdash (\nu x)(CS_x \mid PubConv_z) :: z{:}\,!\mathsf{conv} \tag{2}$$

representing the publication of our file conversion service in the cloud-based infrastructure. Behavioral genericity is in the fact that publishing *any* other service would require following exactly the same above procedure. Assume, for instance, a service $Maps_n$:

$$a{:}\mathsf{api} \vdash Maps_n :: n{:}\mathsf{addr} \multimap (\mathsf{AMaps} \,\&\, \mathsf{GMaps})$$

which when provided a value of type addr (representing an address), it offers a choice between map services AMaps (vector-based maps) and GMaps (raster-based maps). Let $\mathsf{maps} \triangleq \mathsf{addr} \multimap (\mathsf{AMaps} \,\&\, \mathsf{GMaps})$. Clearly, the behavior described by types conv and maps is very different. Still, their relationship with the server at x is exactly the same—they are equally independent. Indeed, by proceeding exactly as we showed above for process $Conv_w$, we can produce a wrapper process $PubMaps_z$ and then obtain:

$$\vdash (\nu x)(CS_x \mid PubMaps_z) :: z{:}\,!\mathsf{maps} \tag{3}$$

The parametric behavior of CS_x can be thus witnessed by comparing (2) and (3) above. In Section 6 we illustrate how to use parametricity to formally justify properties of behavioral genericity/representation independence for processes such as those above.

The above example can be extended to illustrate the interplay of behavioral genericity and concurrency. A more realistic cloud-based platform is one which is always available on a certain name u. This can be represented in our framework by stating

$$\cdot;\cdot \vdash\ !u(x).CS_x :: u{:}\,!\mathsf{CloudServer}$$

and by slightly modifying our assumptions on processes $PubConv_z$ and $PubMaps_z$, in such a way that they become two clients of the persistent server on u:

$$u{:}\mathsf{CloudServer}\,;\,\cdot \vdash \overline{u}\langle x\rangle.PubConv_z :: z{:}\,!\mathsf{conv}$$

(The client for $PubMaps_z$ is similar.) Our typing system ensures that interactions between the server $!u(x).CS_x$ and clients such as the two above will be consistent, safe, and finite. Moreover, these interactions exploit behavioral genericity without interfering with each other, and respecting resource usage policies declared by typing.

Above we have considered a very simple interface type for the cloud-based server. Our framework allows us to represent much richer interfaces. For instance, the type $\mathsf{CloudServerAds} \triangleq \forall X.!(\mathsf{api} \multimap X) \multimap !(X \,\&\, \mathsf{AdListings})$ captures a more sophisticated server which provides its API but forces the resulting system to feature an advertisement service. Type $\mathsf{AdListings}$ encodes a listing of advertisements that the application server "injects" into the service—this injection is represented with a choice $\&$, so as to model the ability of a client to choose to watch an advertisement. It is not difficult to extend this mechanism with further functionalities, such as providing the server developers/clients with an administrator service not exposed to the external clients.

Having illustrated the expressiveness of polymorphic session-typed processes, it is legitimate to investigate the correctness guarantees they enjoy. In the next section, we establish strong normalization, a desirable liveness property for mobile code. Then, in Section 5, we develop a theory of relational parametricity for session-typed processes.

4 Polymorphic Session-Typed Processes are Strongly Normalizing

In this section, we show that well-typed processes of our polymorphic language are (compositionally) strongly normalizing (terminating). Hence, in addition to adhering to the behavior prescribed by session types in a deadlock-free way (cf. Theorems 1 and 2), well-typed, polymorphic processes never engage into infinite computations (Theorem 5). This property is practically meaningful in the context of distributed computing, as it may be used to certify that mobile polymorphic code will not attempt, e.g., a denial-of-service attack by exhausting the resources of a remote service.

Our proof builds on the well-known reducibility candidates technique [14], and generalizes the linear logical relations for session typed processes given in [20] to the *impredicative* polymorphic setting. Technically, the proof is in two stages: we first define a logical predicate inductively on the linear type structure; then, we show that all well-typed processes are in the predicate.

Below, we say that a process P *terminates* (written $P\Downarrow$) if there is no infinite reduction sequence starting with P. The logical predicate uses the following extension to structural congruence with the so-called *sharpened replication axioms* [25].

Definition 5. *We write $\equiv_!$ for the least congruence relation on processes which results from extending structural congruence $\equiv$ (Def. 2) with the following axioms:*

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

Intuitively, $\equiv_!$ allows us to properly "split" processes: axioms (1) and (2) represent the distribution of shared servers among processes, while (3) formalizes the garbage collection of shared servers which can no longer be invoked by any process. It is worth noticing that $\equiv_!$ expresses sound behavioral equivalences in our typed setting.

We now define a notion of *reducibility candidate* at a given type: this is a predicate on well-typed processes which satisfies some crucial closure conditions. As in Girard's proof, the idea is that one of the particular candidates is the "true" logical predicate. Below and henceforth, $\cdot \vdash P :: z{:}A$ stands for a process P which is well-typed under the empty typing environment.

Definition 6 (Reducibility Candidate). *Given a type A and a name z, a* reducibility candidate at $z{:}A$*, written* $\mathsf{R}[z{:}A]$*, is a predicate on all processes P such that $\cdot \vdash P :: z{:}A$ and satisfy the following:*

(1) If $P \in \mathsf{R}[z{:}A]$ then $P\Downarrow$. *(2) If $P \in \mathsf{R}[z{:}A]$ and $P \Longrightarrow P'$ then $P' \in \mathsf{R}[z{:}A]$.*
(3) If for all P_i such that $P \Longrightarrow P_i$ we have $P_i \in \mathsf{R}[z{:}A]$ then $P \in \mathsf{R}[z{:}A]$.

As in the functional case, the properties required for our reducibility candidates are termination (1), closure under reduction (2), and closure under backward reduction (3).

The Logical Predicate. Intuitively, the logical predicate captures the terminating behavior of processes as induced by typing. This way, e.g., the meaning of a terminating process of type $z{:}\forall X.A$ is that after inputing an arbitrary type B, a terminating process of type $z{:}A\{^B/_X\}$ is obtained. As we consider impredicative polymorphism, the main technical issue is that $A\{^B/_X\}$ may be larger than $\forall X.A$, for any measure of size.

The logical predicate is defined inductively, and is parameterized by two mappings, ω and η. Given a context Ω, we write $\omega : \Omega$ to denote that ω is an assignment of closed types to variables in Ω. We write $\omega[X \mapsto A]$ to denote the extension of ω with a new mapping of X to A. We use a similar notation for extensions of η. We write $\hat{\omega}(P)$ (resp. $\hat{\omega}(A)$) to denote the application of the mapping ω to free type-variables in P (resp. in A). We write $\eta : \omega$ to denote that η is an assignment of functions taking names to reducibility candidates, to type variables in Ω (at the types in ω).

It is instructive to compare the key differences between our development and the notion of logical relation for functional languages with impredicative polymorphism, such as System F. In that context, types are assigned to terms and thus one maintains a mapping from type variables to reducibility candidates at the appropriate types. In our setting, since types are assigned to channel names, we need the ability to refer to reducibility candidates at a given type at channel names which are *yet to be determined*. Therefore, when we quantify over all types and all reducibility candidates at that type, intuitively, we need to "delay" the choice of the actual name along which the candidate must offer the session type. A reducibility candidate at type A which is "delayed" in this sense is denoted as $\mathsf{R}[-{:}A]$, where '$-$' stands for a name to be instantiated later on.

We thus define a sequent-indexed family of process predicates: a set of processes $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$ satisfying some conditions is assigned to any sequent of the form $\Omega; \Gamma; \Delta \vdash T$, provided both $\omega{:}\Omega$ and $\eta{:}\omega$. The predicate is defined inductively on the structure of the sequents: the base case considers sequents with an empty left-hand side typing (abbreviated $\mathcal{T}^{\omega}_{\eta}[T]$), whereas the inductive case considers arbitrary typing contexts and relies on principles for process composition (cf. rules (Tcut) and ($\mathsf{Tcut}^!$)).

Definition 7 (Logical Predicate - Base Case). *For any type A and name z, the logical predicate $\mathcal{T}^{\omega}_{\eta}[z{:}A]$ is inductively defined by the set of all processes P such that $\cdot \vdash \hat{\omega}(P) :: z{:}\hat{\omega}(A)$ and satisfy the conditions in Figure 3.*

Definition 8 (Logical Predicate - Inductive Case). *For any sequent $\Omega; \Gamma; \Delta \vdash T$ with a non-empty left hand side environment, we define $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$ (with $\omega : \Omega$ and $\eta : \omega$) as the set of processes inductively defined as follows:*

$$P \in \mathcal{T}^{\omega}_{\eta}[\Gamma; y{:}A, \Delta \vdash T] \textit{ iff } \forall R \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu y)(\hat{\omega}(R) \mid \hat{\omega}(P)) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$$
$$P \in \mathcal{T}^{\omega}_{\eta}[u{:}A, \Gamma; \Delta \vdash T] \textit{ iff } \forall R \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu u)(!u(y).\hat{\omega}(R) \,|\, \hat{\omega}(P)) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$$

Definitions 7 and 8 are the natural extension of the linear logical relations in [20] to the case of impredicative polymorphic types. Notice how the interpretation of the variable type includes the instantiation at name z of the reducibility candidate given

$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}X] \text{ iff } P \in \eta(X)(z)$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}\mathbf{1}] \text{ iff } \forall P'.(P \Longrightarrow P' \wedge P' \not\longrightarrow) \Rightarrow P' \equiv_! \mathbf{0}$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \multimap B] \text{ iff } \forall P' y.(P \stackrel{z(y)}{\Longrightarrow} P') \Rightarrow \forall Q \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu y)(P' \mid Q) \in \mathcal{T}^{\omega}_{\eta}[z{:}B]$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \otimes B] \text{ iff } \forall P' y.(P \stackrel{\overline{(\nu y) z\langle y\rangle}}{\Longrightarrow} P') \Rightarrow \exists P_1, P_2.(P' \equiv_! P_1 \mid P_2 \wedge P_1 \in \mathcal{T}^{\omega}_{\eta}[y{:}A] \wedge P_2 \in \mathcal{T}^{\omega}_{\eta}[z{:}B])$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}!A] \text{ iff } \forall P'.(P \Longrightarrow P') \Rightarrow \exists P_1.(P' \equiv_! !z(y).P_1 \wedge P_1 \in \mathcal{T}^{\omega}_{\eta}[y{:}A])$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A] \text{ iff } (\forall B, P', \mathsf{R}[-{:}B].\ (B\ \mathsf{type} \wedge P \stackrel{z(B)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[-:B]]}[z{:}A])$$
$$P \in \mathcal{T}^{\omega}_{\eta}[z{:}\exists X.A] \text{ iff } (\exists B, \mathsf{R}[-{:}B].(B\ \mathsf{type} \wedge P \stackrel{\overline{z\langle B\rangle}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[-:B]]}[z{:}A])$$

Fig. 3. Logical predicate (base case). Definitions for $\mathcal{T}^{\omega}_{\eta}[z{:}A \oplus B]$ and $\mathcal{T}^{\omega}_{\eta}[z{:}A \,\&\, B]$ are as expected; see [5] for details.

by $\eta(X)$. The clause for the universal $\forall X.A$ denotes that a terminating session of universal type must be able to input *any* type and then be terminating at the open type A, where the meaning of the type variable can be any possible candidate of appropriate type (which includes the actual logical predicate). The clause for the existential is dual.

Proving Strong Normalization. Using the above logical predicate, the proof of strong normalization of well-typed processes follows the one presented in [20]. Roughly, the idea is to define a notion of logical representatives of the dependencies specified in the left-hand side typing. Such representatives simplify reasoning, as they allow to move from predicates for sequents with non empty left-hand side typings to predicates with an empty left-hand side typing, provided processes have been appropriately closed.

The theorem below ensures that $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$ is indeed a reducibility candidate, and thus it implies termination.

Theorem 3 (The Logical Predicate is a Reducibility Candidate). *If $\Omega \vdash A$ type, $\omega : \Omega$, and $\eta : \omega$ then $\mathcal{T}^{\omega}_{\eta}[z{:}A]$ is a reducibility candidate at $z{:}\hat{\omega}(A)$.*

With the technical machinery appropriately defined, we can show the Fundamental Theorem, stating that all well-typed processes belong to the logical predicate.

Theorem 4 (Fundamental Theorem). *If $\Omega; \Gamma; \Delta \vdash P :: T$ then, for all $\omega : \Omega$ and $\eta : \omega$, we have that $\hat{\omega}(P) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$.*

We state the main result of this section, which follows as a consequence of the Fundamental Theorem above: all well-typed polymorphic processes terminate.

Theorem 5 (Strong Normalization). *If $\Omega; \Gamma; \Delta \vdash P{::}T$ then $\hat{\omega}(P)\Downarrow$, for every $\omega{:}\Omega$.*

5 Relational Parametricity for Session-Typed Processes

The cloud-based server given in Section 3 calls for the need for formally asserting that a server with type $u : !\mathsf{CloudServer}$ must behave "the same" independently of the arbitrary types of its clients. In general, the characterization of any well-behaved notion of

type genericity has been captured by some kind of parametricity property, in particular *relational parametricity*, as introduced by Reynolds [24]. The principle of relational parametricity allows us to formally support reasoning about non-trivial properties of processes, such as observational equivalence under changes of representation, which have important consequences on our setting, where types actually denote process behaviors, and the abstraction result implies observational equivalence of a composite system under change of some internal (representation) *protocol* (not just *data*) types.

In this section we thus establish for the first time a relational parametricity result for a session-typed process calculus, based on our underlying logically founded approach.

We first introduce a form of logical equivalence, noted $\approx_{\mathrm{L}}$, which formalizes a relational parametricity principle (Theorem 8) along the lines of Reynolds' abstraction theorem [24] (see [16]). Logical equivalence also allows us to characterize *barbed congruence*, noted $\cong$, in a sound and complete way (Theorem 9). Notice that while $\approx_{\mathrm{L}}$ corresponds to the natural extension of $\mathcal{T}_{\eta}^{\omega}[\Gamma; \Delta \vdash T]$ (cf. Definition 8) to the binary setting, $\cong$ represents the form of contextual equivalence typically used in concurrency.

Barbed Congruence. We begin by introducing barbed congruence. It is defined as the largest equivalence relation on typed processes that is (i) closed under internal actions; (ii) preserves barbs— arguably the most basic observable on the behavior of processes; and is (iii) *contextual*, i.e., preserved by every admissible process context. We make these three desiderata precise, defining first a suitable notion of *type-respecting relations* in our setting. Below, we use $\mathcal{S}$ to range over sequents of the form $\Omega; \Gamma; \Delta \vdash T$.

Definition 9 (Type-respecting relations). *A (binary)* type-respecting relation *over processes, written $\{\mathcal{R}_{\mathcal{S}}\}_{\mathcal{S}}$, is defined as a family of relations over processes indexed by $\mathcal{S}$. We often write $\mathcal{R}$ to refer to the whole family. Also, $\Omega; \Gamma; \Delta \vdash P \,\mathcal{R}\, Q :: T$ stands for*

$$\textit{(i)}\ \Omega; \Gamma; \Delta \vdash P :: T \textit{ and } \Omega; \Gamma; \Delta \vdash Q :: T \quad \textit{and} \quad \textit{(ii)}\ (P, Q) \in \mathcal{R}_{\,\Omega;\Gamma;\Delta\vdash T}.$$

We omit the definitions of reflexivity, transitivity, and symmetry for type-respecting relations; we will say that a type-respecting relation that enjoys the three properties is an equivalence. In what follows, we will often omit the adjective "type-respecting".

We now define τ-closedness, barb preservation, and contextuality.

Definition 10 (τ-closed). *Relation $\mathcal{R}$ is* τ-closed *if $\Omega; \Gamma; \Delta \vdash P\mathcal{R}Q :: T$ and $P \to P'$ imply there exists a Q' such that $Q \Longrightarrow Q'$ and $\Omega; \Gamma; \Delta \vdash P'\mathcal{R}Q' :: T$.*

The following definition of observability predicates, or barbs, extends standard presentations with observables for labeled choice and selection, and type input and output:

Definition 11 (Barbs). *Let $O_x = \{\overline{x}, x, \overline{x.\mathsf{inl}}, \overline{x.\mathsf{inr}}, x.\mathsf{inl}, x.\mathsf{inr}\}$ be the set of* basic observables *under name x. Given a well-typed process P, we write: (i) $\mathsf{barb}(P, \overline{x})$, if $P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P'$; (ii) $\mathsf{barb}(P, \overline{x})$, if $P \xrightarrow{\overline{x\langle A\rangle}} P'$, for some A, P'; (iii) $\mathsf{barb}(P, x)$, if $P \xrightarrow{x(A)} P'$, for some A, P'; (iv) $\mathsf{barb}(P, x)$, if $P \xrightarrow{x(y)} P'$, for some y, P'; (v) $\mathsf{barb}(P, \alpha)$, if $P \xrightarrow{\alpha} P'$, for some P' and $\alpha \in O_x \setminus \{x, \overline{x}\}$. Given some $o \in O_x$, we write $\mathsf{wbarb}(P, o)$ if there exists a P' such that $P \Longrightarrow P'$ and $\mathsf{barb}(P', o)$ holds.*

Definition 12 (Barb preserving relation). *Relation $\mathcal{R}$ is a* barb preserving *if, for every name x, $\Omega;\Gamma;\Delta \vdash P\mathcal{R}Q :: T$ and $\mathsf{barb}(P,o)$ imply $\mathsf{wbarb}(Q,o)$, for any $o \in O_x$.*

In an untyped setting, a relation is said to be *contextual* if it is closed under any well-formed process context C (i.e., a process with a hole). In our case, contexts are typed, and the set of well-formed process contexts (i.e., processes with a typed hole) can be mechanically derived from the typing rules, by exhaustively considering all possibilities for typed holes. This way, e.g., rules (Tcut) and (Tcut$^!$) are the basis for defining parallel contexts. The operation of "filling in" the hole of a context with a process can be handled by an additional typing rule available to contexts, which checks that the type of the process matches that of the hole. For space reasons, we refrain from reporting the complete formal definition of typed process contexts; see [5] for details. Based on these intuitions, we define a contextual relation as follows:

Definition 13 (Contextuality). *Relation $\mathcal{R}$ is* contextual *if $\Omega;\Gamma;\Delta \vdash P\mathcal{R}Q :: T$ implies $\Omega;\Gamma;\Delta' \vdash C[P]\mathcal{R}C[Q] :: T'$, for every Δ', T' and typed context C.*

Definition 14 (Barbed Congruence). Barbed congruence*, noted $\cong$, is the largest equivalence on well-typed processes that is τ-closed, barb preserving, and contextual.*

Logical Equivalence. We now define our notion of logical equivalence for well-typed processes: it arises as a natural extension of the logical predicate of Definition 8 to the relational setting. We begin by defining the crucial notion of equivalence candidate: an equivalence relation on well-typed processes satisfying certain basic closure conditions.

Definition 15 (Equivalence Candidate). *Let A, B be types. 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 conditions:*

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 $P_0 \Longrightarrow P$, we have $(P_0,Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$. Similarly for Q: If $(P,Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ then, for all Q_0 such that $Q_0 \Longrightarrow Q$ then $(P,Q_0) \in \mathcal{R} :: z{:}A \Leftrightarrow B$.*

We often write $(P,Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ as $P\,\mathcal{R}\,Q :: z{:}A \Leftrightarrow B$.

While item (1) says that equivalence candidates are closed with respect to $\cong$, item (2) can be shown to be redundant. As in our definition of logical predicate, we require some auxiliary notation. We recall that $\omega : \Omega$ denotes a type substitution ω that assigns a closed type to type variables in Ω. Given two type substitutions $\omega : \Omega$ and $\omega' : \Omega$, we define an equivalence candidate assignment η between ω and ω' as a mapping of a delayed (in the sense of the mapping η of Section 4) equivalence candidate $\eta(X) :: -{:}\omega(X) \Leftrightarrow \omega'(X)$ to the type variables in Ω. We write $\eta(X)(z)$ for the instantiation of the (delayed) equivalence candidate with the name z. We write $\eta : \omega \Leftrightarrow \omega'$ to denote that η is a (delayed) equivalence candidate assignment between ω and ω'.

We define a sequent-indexed family of process relations, that is, a set of pairs of processes (P,Q), written $\Gamma;\Delta \vdash P \approx_{\mathtt{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$, satisfying some conditions,

$$
\begin{array}{rl}
P \approx_{\mathrm{L}} Q :: z{:}X[\eta : \omega \Leftrightarrow \omega'] & \text{iff } (P,Q) \in \eta(X)(z) \\
P \approx_{\mathrm{L}} Q :: z{:}\mathbf{1}[\eta : \omega \Leftrightarrow \omega'] & \text{iff } \forall P', Q'.\ (P \Longrightarrow P' \wedge P' \not\longrightarrow \wedge\ Q \Longrightarrow Q' \wedge Q' \not\longrightarrow) \Rightarrow \\
 & \quad (P' \equiv_! \mathbf{0} \wedge Q' \equiv_! \mathbf{0}) \\
P \approx_{\mathrm{L}} Q :: z{:}A \multimap B[\eta : \omega \Leftrightarrow \omega'] & \text{iff } \forall P', y.\ (P \xrightarrow{z(y)} P') \Rightarrow \exists Q'.Q \stackrel{z(y)}{\Longrightarrow} Q'\ s.t. \\
 & \quad \forall R_1, R_2.\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \\
 & \quad (\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'] & \text{iff } \forall P', y.\ (P \xrightarrow{\overline{(\nu y) z\langle y \rangle}} P') \Rightarrow \exists Q'.Q \stackrel{\overline{(\nu y) z\langle y \rangle}}{\Longrightarrow} Q'\ s.t. \\
 & \quad \forall R_1, R_2, n.\ y{:}A \vdash R_1 \approx_{\mathrm{L}} R_2 :: n{:}\mathbf{1}[\eta : \omega \Leftrightarrow \omega'] \\
 & \quad (\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[\eta : \omega \Leftrightarrow \omega'] & \text{iff } \forall P'.\ (P \xrightarrow{z(y)} P') \Rightarrow \exists Q'.Q \stackrel{z(y)}{\Longrightarrow} Q' \wedge \\
 & \quad \forall R_1, R_2, n.\ y{:}A \vdash R_1 \approx_{\mathrm{L}} R_2 :: n{:}\mathbf{1}[\eta : \omega \Leftrightarrow \omega'] \\
 & \quad (\nu y)(P' \mid R_1) \approx_{\mathrm{L}} (\nu y)(Q' \mid R_2) :: z{:}!A[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathrm{L}} Q :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff } \forall B_1, B_2, P', \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \xrightarrow{z(B_1)} P') \Rightarrow \\
 & \quad \exists Q'.Q \stackrel{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]] \\
P \approx_{\mathrm{L}} Q :: z{:}\exists X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff } \exists B_1, B_2, \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \xrightarrow{\overline{z\langle B \rangle}} P') \Rightarrow \\
 & \quad \exists Q'.Q \stackrel{\overline{z\langle B \rangle}}{\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]]
\end{array}
$$

Fig. 4. Logical equivalence (base case). Definitions for $P \approx_{\mathrm{L}} Q :: z{:}A \,\&\, B[\eta : \omega \Leftrightarrow \omega']$ and $P \approx_{\mathrm{L}} Q :: z{:}A \oplus B[\eta : \omega \Leftrightarrow \omega']$ are as expected; see [5] for details.

is assigned to any sequent of the form $\Omega; \Gamma; \Delta \vdash T$, with $\omega : \Omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$. As in the definition of the logical predicate, logical equivalence is defined inductively on the structure of the sequents: the base case considers empty left-hand side typings, whereas the inductive case which considers arbitrary typing contexts.

Definition 16 (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 largest 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 in Figure 4.*

Definition 17 (Logical Equivalence - Inductive Case). *Let Γ, Δ be non empty typing environments. Given the sequent $\Omega; \Gamma; \Delta \vdash T$, 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:*

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

This way, logical equivalence turns out to be a generalization of the logical predicate $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \vdash T]$ (Definition 7) to the binary setting. The key difference lies in the defi-

nition of candidate (here called equivalence candidate), which instead of guaranteeing termination, enforces closure under barbed congruence.

Theorem 6 below is the binary analog of Theorem 4 (Fundamental Theorem). Its proof is similar: we establish that logical equivalence is one of the equivalence candidates, and then show that well-typed processes are logically equivalent to themselves.

Theorem 6 (Logical Equivalence is an Equivalence Candidate). *The relation* $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$ *is an equivalence candidate at* $z{:}\hat{\omega}(A)$ *and* $z{:}\hat{\omega'}(A)$.

The final ingredient for our desired parametricity result is the following theorem:

Theorem 7 (Compositionality). *Let* B *be any type. Also, let* $\mathcal{R} :: -{:}\hat{\omega}(B) \Leftrightarrow \hat{\omega'}(B)$ *stand for logical equivalence (cf. Definition 16).*
Then, $P \approx_{\mathrm{L}} Q :: z{:}A\{{}^{B}/_{X}\}[\eta : \omega \Leftrightarrow \omega']$ *if and only if*

$$P \approx_{\mathrm{L}} Q :: z{:}A[\eta[X \mapsto \mathcal{R}] \ : \ \omega[X \mapsto \hat{\omega}(B)] \Leftrightarrow \omega'[X \mapsto \hat{\omega'}(B)]]$$

We now state the main result of the section; its proof depends on a backward closure property, and on Theorems 6 and 7.

Theorem 8 (Relational Parametricity). *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then, for all* $\omega, \omega' : \Omega$ *and* $\eta : \omega \Leftrightarrow \omega'$*, we have* $\Gamma; \Delta \vdash \hat{\omega}(P) \approx_{\mathrm{L}} \hat{\omega'}(P) :: z{:}A[\eta : \omega \Leftrightarrow \omega']$.

Remarkably, by appealing to parametricity and contextuality of logical equivalence, we can show that $\approx_{\mathrm{L}}$ and $\cong$ coincide. This result establishes a definitive connection between the usual barb-based notion of observational equivalence from concurrency theory, and the logical equivalence induced by our logical relational semantics (see [5]).

Theorem 9 (Logical Equivalence and Barbed Congruence coincide). *Relations* $\approx_{\mathrm{L}}$ *and* $\cong$ *coincide for well-typed processes. More precisely:*

1. *If* $\Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$ *holds for any* $\omega, \omega' : \Omega$ *and* $\eta : \omega \Leftrightarrow \omega'$*, then* $\Omega; \Gamma; \Delta \vdash P \cong Q :: z{:}A$
2. *If* $\Omega; \Gamma; \Delta \vdash P \cong Q :: z{:}A$ *then* $\Gamma; \Delta \vdash P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$ *for some* $\omega, \omega' : \Omega$ *and* $\eta : \omega \Leftrightarrow \omega'$.

6 Using Parametricity to Reason About the Cloud Server

Here we illustrate a simple application of our parametricity result for reasoning about concurrent polymorphic processes. We are interested in studying a restaurant finding system; such an application is expected to rely on some maps application, to be uploaded to a cloud server. In our example, we would like to consider two different implementations of the system, each one relying on a different maps service. We assume that the two implementations will comply with the expected specification for the restaurant service, even if each one uses a different maps service (denoted by closed types AMaps and GMaps). This assumption may be precisely expressed by the judgment

$$s{:}!(\mathsf{api} \multimap \circ X) \multimap \circ !X \vdash C_1 \approx_{\mathrm{L}} C_2 :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2] \tag{4}$$

where $\eta_r(X) = \mathcal{R}$, $\omega_1(X) = \mathsf{AMaps}$, and $\omega_2(X) = \mathsf{GMaps}$, where $\mathcal{R}$ is an equivalence candidate that relates AMaps and GMaps, i.e., $\mathcal{R} : \mathsf{AMaps} \Leftrightarrow \mathsf{GMaps}$. The type of the restaurant finding application is denoted rest; it does not involve type variable X. Also, we assume X does not occur in the implementations C_1, C_2. Intuitively, the above captures the fact that C_1 and C_2 are similar "up to" the relation $\mathcal{R}$.

By exploiting the shape of type $\mathsf{CloudServer}$, we can ensure that *any* process S such that $\cdot \vdash S :: s{:}\mathsf{CloudServer}$ behaves uniformly, offering the same generic behavior to its clients. That is to say, once the server is instantiated with an uploaded application, the behavior of the resulting system will depend only on the type provided by the application. Recall the polymorphic type of our cloud server: $\mathsf{CloudServer} \triangleq \forall X.!(\mathsf{api} \multimap X) \multimap !X$. Based on the form of this type and combining inversion on typing and strong normalization (Theorem 5), there is a process $SBody$ such that

$$S \xrightarrow{s(X)} SBody \qquad X; \cdot; \cdot \vdash SBody :: s{:}!(\mathsf{api} \multimap X) \multimap !X \tag{5}$$

hold. By parametricity (Theorem 8) on (5), we obtain

$$\cdot \vdash \hat{\omega}(SBody) \approx_{\mathrm{L}} \hat{\omega}'(SBody) :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta : \omega \Leftrightarrow \omega']$$

for any ω, ω', and η. In particular, it holds for the η_r, ω_1 and ω_2 defined above:

$$\cdot \vdash \hat{\omega}_1(SBody) \approx_{\mathrm{L}} \hat{\omega}_2(SBody) :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta_r : \omega_1 \Leftrightarrow \omega_2] \tag{6}$$

By Definition 17, the formal relationship between C_1 and C_2 given by (4) implies

$$\cdot \vdash (\nu s)(\hat{\omega}_1(R_1) \mid C_1) \approx_{\mathrm{L}} (\nu s)(\hat{\omega}_2(R_2) \mid C_2) :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2]$$

for any R_1, R_2 such that $R_1 \approx_{\mathrm{L}} R_2 :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta_r : \omega_1 \Leftrightarrow \omega_2]$. In particular, it holds for the two processes related in (6) above. Combining these two facts, we have:

$$\cdot \vdash (\nu s)(\hat{\omega}_1(SBody) \mid C_1) \approx_{\mathrm{L}} (\nu s)(\hat{\omega}_2(SBody) \mid C_2) :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2]$$

Since rest does not involve X, using Theorem 7 we actually have:

$$\cdot \vdash (\nu s)(\hat{\omega}_1(SBody) \mid C_1) \approx_{\mathrm{L}} (\nu s)(\hat{\omega}_2(SBody) \mid C_2) :: z{:}\mathsf{rest}[\emptyset : \emptyset \Leftrightarrow \emptyset] \tag{7}$$

Now, given (7), and using backward closure of $\approx_{\mathrm{L}}$ under reductions (possible because of Theorem 6 and Definition 15), we obtain:

$$\cdot \vdash (\nu s)(S \mid \overline{s}\langle \mathsf{AMaps} \rangle.C_1) \approx_{\mathrm{L}} (\nu s)(S \mid \overline{s}\langle \mathsf{GMaps} \rangle.C_2) :: z{:}\mathsf{rest}[\emptyset : \emptyset \Leftrightarrow \emptyset]$$

Then, using Theorem 9, we finally have

$$\cdot \vdash (\nu s)(S \mid \overline{s}\langle \mathsf{AMaps} \rangle.C_1) \cong (\nu s)(S \mid \overline{s}\langle \mathsf{GMaps} \rangle.C_2) :: z{:}\mathsf{rest}$$

This simple, yet illustrative example shows how one may use our parametricity results to reason about the observable behavior of concurrent systems that interact under a polymorphic behavioral type discpline.

7 Related Work

To our knowledge, our work is the first to establish a relational parametricity principle (in the sense of Reynolds [24]) in the context of a rich behavioral type theory for concurrent processes. Combined with parametricity, our type preservation, progress, and strong normalization results therefore improve upon previous works on polymorphism for session types ([12,10,4,28,9,15], see below) by providing general, logic-based foundations for the analysis of behavioral genericity in structured communications.

By extending the notion of subtyping in [11], Gay [12] studied a form of bounded polymorphism associated to branch/choice types: each branch is quantified by a type variable with upper and lower bounds. Forms of unbounded polymorphism can be enabled via special types Bot and Top. Dezani et al. [10] study bounded polymorphism for a session-typed, object-oriented language. Bono and Padovani [3,4] rely on unbounded polymorphism in a session types variant that is used to ensure correct (copyless) message-passing programs. Dardha et al. [9] develop an encoding of session types into linear/variant types; it can be extended to handle session types with existential parametric polymorphism (as in the π-calculus [25,27], see below) and bounded polymorphism (as in [12]). Goto et al. [15] develop a model of session polymorphism, in which session types are modeled as labeled transition systems which may incorporate deductive principles; polymorphism relies upon suitable deductions over transitions.

Most related to our developments are works by Berger et al. [1,2] and Wadler [28]. Berger et al. [1,2] were the first to propose a polymorphically typed π-calculus with universal and existential quantification. Their system is not based on session types but results from combining so-called action types with linearity and duality principles. In their setting, enforcing resource usage disciplines entails a dedicated treatment for issues such as, e.g., sequentiality/causality in communications and type composition; in contrast, in the context of session-typed interactions, our logic-based approach offers general principles for handling such issues (e.g., typed process composition via cut). As in our case, they prove strong normalization of well-typed processes using reducibility candidates; however, due to the differences on typing, the proofs in [1,2] cannot be compared to our developments. In particular, our application of the reducibility candidates technique generalizes the linear logical relations we defined in [20]. While in [1] a parametricity result is stated, the journal paper [2] develops a behavioral theory based on generic transitions together with a fully abstract embedding of System F. Here again detailed comparisons with our proofs are difficult, because of the different typing disciplines considered in each case. Wadler [28] proposed an interpretation of session types as classical linear logic (along the lines of [7]). His system supports the kind of parametric polymorphism we develop here. However, the focus of [28] is not on the theory of parametric polymorphism. In particular, it does not address proof techniques for behavioral genericity nor establishes a relational parametricity principle, as we do here.

In a broader context—and loosely related to our work—Turner [27] studied impredicative, existential polymorphism for a simply-typed π-calculus (roughly, the discipline in which types describe the objects names can carry). In processes, polymorphism is expressed as explicit type parameters in input/output prefixes. Sangiorgi and Pierce [22] proposed a behavioral theory for Turner's framework. Neither of these works address strong normalization nor study relational parametricity. Building upon [22], Jeffrey and

Rathke [19] show that weak bisimulation is fully abstract for observational equivalence for an asynchronous polymorphic π-calculus. Recently, Zhao et al. [30] studied linearity and polymorphism for (variants of) System F. They prove relational parametricity via logical relations for open terms, but no concurrent interpretation is considered.

8 Concluding Remarks

In this paper, we have presented a systematic study of behavioral genericity for concurrent processes. Our study is in the context of session types—a rich behavioral type theory able to precisely describe complex communication protocols. Our work naturally generalizes recent discoveries on the correspondence between linear logic propositions and session types [6,7,20]. Previous works on genericity for concurrent processes appeal to various forms of polymorphism. In contrast to most of such works, and by developing a theory of impredicative, parametric polymorphism, we are able to formally connect the concept of behavioral parametricity with the well-known principle of relational parametricity, as introduced by Reynolds [24]. Since in our framework polymorphism accounts for the exchange of abstract protocols, relational parametricity enables us to effectively analyze concurrent systems which are parametric on arbitrarily complex communication disciplines. In addition to enjoying a relational parametricity principle, well-typed processes in our system respect session types in a deadlock-free way and are strongly normalizing. This unique combination of results confers very strong correctness guarantees for communicating systems. As a running example, we have illustrated how to specify and reason about a simple polymorphic cloud-based application server. In future work we would like to explore generalizations of our relational parametricity result so as to address security concerns (along the lines of [29]).

Acknowledgments. This research was supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, under grants INTERFACES NGN-44 / 2009 and SFRH / BD / 33763 / 2009, and CITI; and by the Army Research Office under Award No. W911NF-09-1-0273. We thank the anonymous reviewers for their useful comments.

References

1. Berger, M., Honda, K., Yoshida, N.: Genericity and the pi-calculus. In: Proc. of FoSSaCS. LNCS, vol. 2620, pp. 103–119. Springer (2003)
2. Berger, M., Honda, K., Yoshida, N.: Genericity and the pi-calculus. Acta Inf. 42(2-3), 83–141 (2005)
3. Bono, V., Padovani, L.: Polymorphic endpoint types for copyless message passing. In: Proc. of ICE'11. EPTCS, vol. 59, pp. 52–67 (2011)
4. Bono, V., Padovani, L.: Typing copyless message passing. Logical Methods in Computer Science 8(1) (2012)
5. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Relational parametricity for polymorphic session types. Tech. rep., CMU-CS-12-108, Carnegie Mellon Univ. (Apr 2012)
6. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR'2010. LNCS, vol. 6269, pp. 222–236. Springer (2010)

7. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types (2012), under Revision - http://www.cs.cmu.edu/~fp/papers/sessions12.pdf
8. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: TLDI'12. pp. 1–12. ACM, New York, NY, USA (2012)
9. Dardha, O., Giachino, E., Sangiorgi, D.: Session Types Revisited. In: PPDP. pp. 139–150. ACM (2012)
10. Dezani-Ciancaglini, M., Giachino, E., Drossopoulou, S., Yoshida, N.: Bounded session types for object oriented languages. In: FMCO'06. LNCS, vol. 4709, pp. 207–245. Springer (2007)
11. Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. 42, 191–225 (2005)
12. Gay, S.J.: Bounded polymorphism in session types. Math. Struc. in Comp. Sci. 18(5), 895–930 (2008)
13. Girard, J.Y.: 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. pp. 63–92. North-Holland Publishing Co. (1971)
14. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and Types (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press (1989)
15. Goto, M., Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: An Extensible Approach to Session Polymorphism (2012), http://fpl.cs.depaul.edu/projects/xpol/
16. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press (2012)
17. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993)
18. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP'98. LNCS, vol. 1381, pp. 122–138. Springer (1998)
19. Jeffrey, A., Rathke, J.: Full abstraction for polymorphic pi-calculus. Theor. Comput. Sci. 390(2-3), 171–196 (2008)
20. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: Proc. of ESOP. LNCS, vol. 7211, pp. 539–558. Springer (2012)
21. Pfenning, F., Caires, L., Toninho, B.: Proof-carrying code in a session-typed process calculus. In: Proc. of CPP '11. LNCS, vol. 7086, pp. 21–36. Springer (2011)
22. Pierce, B.C., Sangiorgi, D.: Behavioral equivalence in the polymorphic pi-calculus. J. ACM 47(3), 531–584 (2000)
23. Reynolds, J.C.: Towards a theory of type structure. In: Programming Symposium, Proceedings Colloque sur la Programmation. pp. 408–423. Springer-Verlag, London, UK, UK (1974)
24. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing 83. pp. 513–523. Elsevier Science Publishers B. V. (1983)
25. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2001)
26. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Proc. of PPDP '11. pp. 161–172. ACM, New York, NY, USA (2011)
27. Turner, D.: The polymorphic pi-calculus: Theory and implementation. Tech. rep., ECS-LFCS-96-345, Univ. of Edinburgh (1996)
28. Wadler, P.: Propositions as sessions. In: Thiemann, P., Findler, R.B. (eds.) ICFP. pp. 273–286. ACM (2012)
29. Washburn, G., Weirich, S.: Generalizing parametricity using information-flow. In: LICS. pp. 62–71. IEEE Computer Society (2005)
30. Zhao, J., Zhang, Q., Zdancewic, S.: Relational parametricity for a polymorphic linear lambda calculus. In: APLAS. LNCS, vol. 6461, pp. 344–359. Springer (2010)