Genericity and the π-Calculus

Martin Berger[⋆] Kohei Honda[⋆] Nobuko Yoshida[⋆⋆]

Abstract. We introduce a second-order polymorphic π-calculus based on duality principles. The calculus and its behavioural theories cleanly capture some of the core elements of significant technical development on polymorphic calculi in the past. This allows precise embedding of generic sequential functions as well as seamless integration with imperative constructs such as state and concurrency. Two behavioural theories are presented and studied, one based on a second-order logical relation and the other based on a polymorphic labelled transition system. The former gives a sound and complete characterisation of the contextual congruence, while the latter offers a tractable reasoning tool for a wide range of generic behaviours. The applicability of these theories is demonstrated through non-trivial reasoning examples and a fully abstract embedding of System F, the second-order polymorphic λ-calculus.

1 Introduction

Genericity is a useful concept in software engineering which allows encapsulation of design decisions such that data-structures and algorithms can be changed more independently. It arises in two distinct but closely related forms: one, which we may refer to as *universal*, aids generic manipulation of data, as in lists, queues, trees or stacks. The other *existential* form facilitates hiding of structure from the outside, asking for it to be treated generically. In both cases, genericity partitions programs into parts that depend on the precise nature of the data under manipulation and parts that do not, supporting principled code reuse and precise type-checking. For example, C++ evolved from C by adding genericity in the form of *templates* (universal) and *objects* (existential).

It is known that key aspects of genericity for sequential functional computation are captured by second-order polymorphism where type variables, in addition to program variables, can be abstracted and instantiated. In particular, the two forms of genericity mentioned above are accounted for by the two forms of quantification coming from logic, $\forall$ and $\exists$. Basic formalisms incorporating genericity include System F (the second-order λ-calculus) [8, 28] and ML [18]. Centring on these and related formalisms, a rich body of studies on type disciplines, semantics and proof principles for genericity has been accumulated.

The present work aims to offer a π-calculus based starting point for repositioning and generalising the preceding functional account of genericity in the broader realm of interaction. We are partly motivated by the lack of a general

⋆ Department of Computer Science, Queen Mary, University of London, London, U.K.
⋆⋆ Department of Computing, Imperial College, London, U.K.

mathematical basis of genericity that also covers state, concurrency and nondeterminism. For example, the status of two fundamental concepts for reasoning about generic computation, relational parametricity [28] and its dual simulation principle [1, 19, 27], is only well-understood for pure functions. But a mathematical basis of diverse forms of generic computation is important when we wish to reason about software made up from many components with distinct behavioural properties, from purely functional behaviour to programs with side effects to distributed computing, all of which may exhibit certain forms of genericity.

The π-calculus is a small syntax for communicating processes in which we can precisely represent many classes of computational behaviours, from purely sequential functions to those of distributed systems [5, 7, 17, 32, 33]. Can we find a uniform account of genericity for diverse classes of computational behaviour using the π-calculus? This work presents our initial results in this direction, concentrating on a polymorphic variant of the linear/affine π-calculus with state [7, 12, 32, 33]. It turns out that the duality principle in the linear/affine type structure naturally extends to second-order quantification, leading to a powerful theory of polymorphism that allows precise embedding of existing polymorphic functional calculi and unifies some of the significant technical elements of the known theories of genericity.

Summary of Contributions. The following summarises the main technical contributions of the present paper.

1. Introduction of the polymorphic linear/affine π-calculus based on duality principles, as well as its consistent extension to state and concurrency. One of the central syntactic results is strong normalisability for linear polymorphic processes.
2. Theory of behavioural equivalences based on a generic labelled transition system applicable to both sequential and concurrent polymorphic processes. We apply the theory to non-trivial reasoning examples as well as to a fully abstract embedding of System F in the linear polymorphic π-calculus.
3. A sound and complete characterisation of the contextual congruence by a second-order logical relation for linear/affine polymorphic processes, leading to relational parametricity and a simulation principle for extensional equality. The theory offers a tractable reasoning tool for generic processes as we demonstrate through examples.

Related Work. Originally the second-order polymorphism for the λ-calculus was discovered by Girard [8] and Reynolds [28] with a main focus on universal abstraction. Later Mitchell and Plotkin [19, 27] relates its dual form, existential abstraction, to data hiding. Exploiting a duality principle, the present theory unifies these two uses of polymorphism, data-hiding and parametricity, into a single framework, both in operation and in typing. The unification accompanies new reasoning techniques such as generic labelled transition.

Turner [30] is the first to study (impredicative) polymorphism in the π-calculus, giving a type-preserving encoding of System F. His type discipline is incorporated into Pict [23]. Vasconcelos [31] studies a predicative polymorphic typing discipline and shows that it can type-check interesting polymorphic processes while allowing tractable type inference. Our use of a duality principle (whose origin can be traced back to Linear Logic [9]) is the main difference from those previous approaches.

Pierce and Sangiorgi [22] study a behavioural equivalence for Turner's calculus and observe that existential types can reduce the number of transitions by prohibiting interactions at hidden channels. Lazic, Nowak and Roscoe [15] show that when programs manipulate data abstractly (called *data independence*), a transition system with a parametricity property can be used for reasoning, leading to efficient model checking techniques. The generic labelled transition unifies, and in some cases strengthens, these ideas as dual aspects of a single framework. The use of duality also leads to lean and simple constructions.

Pitts studies contextual congruences in PCF-like polymorphic functional calculi and characterises them via syntactic logical relations [25, 26], cf. [24]. His work has inspired constructions and proof techniques for our corresponding characterisations. The present relational theory for the π-calculus treats several elements of Pitts' theories (for example call-by-name and call-by-value) in a uniform framework. Duality also substantially simplifies the constructions.

Recently, several studies of the semantics of polymorphism based on games and other intensional models have appeared. Hughes [13] presents game semantics for polymorphism in which strategies pass arenas to represent type passing and proves full abstraction for System F. His model is somewhat complex due to its direct representation of type instantiation. Murawski and Ong [21] substantially simplify Hughes approach, but do not obtain full abstraction for impredicative polymorphism. Abramsky and Lenisa [3, 4] give a fully abstract model for predicative polymorphism using interaction combinators. Treatment of impredicative polymorphism is left as an open issue. In view of the relationship between π-calculi and game semantics [7, 11, 14], it would be interesting to use typed processes from the present work to construct game-based categories.

Structure of the paper. Section 2 informally illustrates key ideas with examples. Section 3 introduces the syntax and typing rules. Section 4 gives a sound and complete characterisation of a contextual congruence by a second-order logical relation. Section 5 studies a generic labelled transition and the induced equivalence. Section 6 discusses non-trivial applications of two behavioural theories, including a fully abstract embedding of System F. The full technical development of the presented material is found in [6].

Acknowledgement. We thank anonymous referees for their helpful suggestions. The first two authors are partially supported by EPSRC grant GR/N/37633. The third author is partially supported by EPSRC grant GR/R33465/01.

2 Generic Processes, Informally

This section introduces key ideas with simple examples. We start with the following small polymorphic process (which is essentially a process encoding of the polymorphic identity), using the standard syntax of the (asynchronous) π-calculus.

$$\vdash !x(yz).\overline{z}\langle y\rangle \triangleright x : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$$

In this process $\overline{z}\langle y\rangle$ is an output of y along the channel z and $!x(yz).\overline{z}\langle y\rangle$ is a replicated input, repeatedly receiving two names y and z at x. After having received y and z, it sends y along z.

The typing $x : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ assigns $\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ to x. X is a type variable: $\overline{\mathrm{X}}$ indicates the dual of X. $(\mathrm{X})^{\uparrow}$ sends a name of type X exactly once, while $(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ indicates the behaviour of receiving two names at a replicated input channel, one used as $\overline{\mathrm{X}}$ and the other as $(\mathrm{X})^{\uparrow}$. Finally, $\forall \mathrm{X}$ universally abstracts X, saying X can be any type. Here $\forall \mathrm{X}$ binds X and its dual simultaneously. The operational content of typing a channel with a type variable is to enforce that y cannot be used as an interaction point (which would require a concrete type). Hence y with a variable $\overline{\mathrm{X}}$ only appears as a value in a message.

Next we consider the process which is dual to the above agent. Let $\mathsf{t}\langle y\rangle \stackrel{\mathrm{def}}{=} !y(a_1a_2z).\overline{z}\langle a_1\rangle$, $\mathsf{not}\langle cw\rangle \stackrel{\mathrm{def}}{=} !c(a_1a_2z).\overline{w}\langle a_2a_1z\rangle$ and $\mathbb{B} \stackrel{\mathrm{def}}{=} \forall \mathrm{X}.(\overline{\mathrm{X}}\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ (which are, respectively, truth, negation and the polymorphic boolean type).

$$\vdash \overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \triangleright x : \exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}, e : (\mathbb{B})^{\uparrow} \qquad (1)$$

This process sends y and z (respectively representing the truth and the continuation) via x, where $\overline{x}(yz)P$ stands for $(\boldsymbol{\nu}\, yz)(\overline{x}\langle yz\rangle|P)$. Then it receives a single name at z and sends its negation via e. To understand the typing, let's look at the situation before existential abstraction:

$$\vdash \overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \triangleright x : (\mathbb{B}(\overline{\mathbb{B}})^{\downarrow})^{?}, e : (\mathbb{B})^{\uparrow} \qquad (2)$$

We now abstract $\mathbb{B}$ and its dual at x simultaneously, obtaining $\exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}$ ($\exists \mathrm{X}$ binds both X and $\overline{\mathrm{X}}$). Thus existential abstraction hides the concrete type $\mathbb{B}$.

The types $\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ and $\exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}$ are dual to each other and indicate that composition of two processes is possible. When composed, the process interacts as follows. Below and henceforth we write $\mathsf{id}\langle x\rangle$ for $!x(yz).\overline{z}\langle y\rangle$.

$$\begin{array}{rcl}\mathsf{id}\langle x\rangle \mid \overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) & \longrightarrow & \mathsf{id}\langle x\rangle \mid (\boldsymbol{\nu}\, yz)(\overline{z}\langle y\rangle|\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle)\\ & \longrightarrow & \mathsf{id}\langle x\rangle \mid (\boldsymbol{\nu}\, y)(\mathsf{t}\langle y\rangle|\overline{e}(c)\mathsf{not}\langle cy\rangle)\\ & \approx & \mathsf{id}\langle x\rangle \mid \overline{e}(c)\mathsf{f}\langle c\rangle\end{array}$$

Here $\mathsf{f}\langle c\rangle \stackrel{\mathrm{def}}{=} !c(xyz).\overline{z}\langle y\rangle$ (representing falsity) and $\approx$ is the standard weak bisimilarity. As this interaction indicates, a universally abstracted name, after its receipt from the environment, can only be used to be sent back to the environment as a free name. The dual existential side can then count on such behaviour of the interacting party: in the above case, the process on the right-hand side can expect that, via z, it would receive the name y as a free name which it has exported in the initial reduction, as it indeed does in the second transition.

This duality plays the key role in defining generic labelled transitions, which induce behavioural equivalences more abstract (larger) than non-generic ones and which are applicable to the reasoning over a wide range of generic behaviours. We use an example of a generic transition sequence of the process in (1).

$$\overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \overset{\overline{x}(yz)}{\longrightarrow} \overset{z\langle y\rangle}{\longrightarrow} \mathsf{t}\langle y\rangle|\overline{e}(c)\mathsf{not}\langle cy\rangle$$

A crucial point in this transition is that it does *not* allow a bound input in the second action, because the protocol at existentially abstracted names is opaque. The induced name substitution then opens a channel for internal communication. In contrast, the process in (2), different from (1) only in type, has the following transition sequence.

$$\overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \overset{\overline{x}(yz)}{\longrightarrow} \overset{z(w)}{\longrightarrow} \mathsf{t}\langle y\rangle|\overline{e}(c)\mathsf{not}\langle cw\rangle.$$

Note that we have a bound input in the second action; the transition sequence is now completely controlled by type information, without sending/receiving concrete values. Here the duality principle dictates existential/universal type variables correspond to free name passing, while concrete types (which rigorously specify protocols of interaction by their type structure) correspond to bound name passing.

This way, the duality in the type structure is precisely reflected in the duality in behaviour. This duality principle is also essential in the construction of the second-order logical relations, for proving the strong normalisability of linear polymorphic processes and for various embedding results.

3 A Polymorphic π-Calculus

3.1 Processes

In this section we formally introduce a polymorphic version of the affine π-calculus [7] and its extensions to linearity [32, 33] and state [12].

Let $x, y, \ldots$ range over a countable set $\mathcal{N}$ of *names*. $\vec{y}$ is a vector of names. Then *processes*, ranged over by $P, Q, R, \ldots$, are given by the following grammar.

$$P \;::=\; x(\vec{y}).P \mid !x(\vec{y}).P \mid \overline{x}\langle\vec{y}\rangle \mid P|Q \mid (\boldsymbol{\nu}\, x)P \mid \mathbf{0}$$

Names in round parenthesis act as binders, based on which we define the alpha equality $\equiv_\alpha$ in the standard way. We briefly illustrate each construct. $x(\vec{y}).P$ inputs via x with a continuation P. Its replicated counterpart is $!x(\vec{y}).P$. $\overline{x}\langle\vec{y}\rangle$ outputs $\vec{y}$ along x. In each of these agents, the initial free occurrence is *subject*, while each carried name in an input/output is *object*. The parallel composition of P and Q is $P|Q$ and $(\boldsymbol{\nu}\, x)P$ makes x private to P. $\mathbf{0}$ is the inaction, indicating the lack of behaviour. The structural equality $\equiv$ is standard [17] (without the replication rule $!P \equiv P|!P$), which we omit. The reduction relation $\longrightarrow$ are generated from:

$$x(\vec{y}).P \,|\, \overline{x}\langle\vec{z}\rangle \;\longrightarrow\; P\{\vec{z}/\vec{y}\} \qquad\qquad !x(\vec{y}).P \,|\, \overline{x}\langle\vec{z}\rangle \;\longrightarrow\; !x(\vec{y}).P \,|\, P\{\vec{z}/\vec{y}\}$$

closing under parallel composition and hiding, taking processes modulo $\equiv$.

3.2 Types

Below we introduce polymorphic extensions of different classes of first-order type disciplines, starting from the affine sequential polymorphic typing [7]. Following [7, 32, 33] we use the set of *action modes*, which are:

$\downarrow$ Affine input (at most once),	$\uparrow$ Affine output (at most once),
$\mathbf{!}$ Server at replicated input,	$\mathbf{?}$ Client requests to $\mathbf{!}$,

as well as $\updownarrow$, which indicates non-composability at affine channels. $\downarrow, \mathbf{!}$ are *input modes*, while $\uparrow, \mathbf{?}$ are *output modes*. Input/output modes are together called *directed modes*. $p, p', \ldots$ (resp. p_{I}, resp. p_{O}) denote directed (resp. input, resp. output) modes. We define $\overline{p}$, the *dual* of p, by: $\overline{\downarrow} = \uparrow$, $\overline{\mathbf{!}} = \mathbf{?}$ and $\overline{\overline{p}} = p$.

Let X, X', ... range over a countable set of *type variables*. We fix a bijection $\overline{\mathrm{X}}$ which is self-inverse (i.e. $\overline{\overline{\mathrm{X}}} = \mathrm{X}$) and irreflexive (i.e. $\overline{\mathrm{X}} \neq \mathrm{X}$). Each X is assigned a directed action mode p, written X^p, so that the mode of $\overline{\mathrm{X}}$ is always dual to that of X. Channel types are given as follows:

$$\tau ::= \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} \mid \langle \tau_{\mathrm{I}}, \tau'_{\mathrm{O}} \rangle \quad \tau_{\mathrm{I}} ::= \mathrm{X}^{p_{\mathrm{I}}} \mid (\vec{\tau}_{\mathrm{O}})^{p_{\mathrm{I}}} \mid \forall \mathrm{X}.\tau_{\mathrm{I}} \mid \exists \mathrm{X}.\tau_{\mathrm{I}} \quad \tau_{\mathrm{O}} ::= \mathrm{X}^{p_{\mathrm{O}}} \mid (\vec{\tau}_{\mathrm{I}})^{p_{\mathrm{O}}} \mid \forall \mathrm{X}.\tau_{\mathrm{O}} \mid \exists \mathrm{X}.\tau_{\mathrm{O}}$$

τ_{I} and τ_{O} are called *input type* and *output type*, respectively, which are together called *directed types*. Note quantification is given only on directed types. For each directed τ, the *dual of* τ, $\overline{\tau}$, is the result of dualising all action modes, type variables and quantifiers in τ. In $\langle \tau, \tau' \rangle$, we always assume $\tau' = \overline{\tau}$. Following [7, 12, 32, 33], we assume the sequentiality constraint on channel types, i.e. $\downarrow$-type carries only $\mathbf{?}$-types while a $\mathbf{!}$-type carries $\mathbf{?}$-types and a unique $\uparrow$-type, dually for $\uparrow / \mathbf{?}$. We set $\mathsf{md}(\mathrm{X}^p) = p$, $\mathsf{md}((\vec{\tau})^p) = p$ and $\mathsf{md}(\forall \mathrm{X}.\tau) = \mathsf{md}(\exists \mathrm{X}.\tau) = \mathsf{md}(\tau)$, as well as $\mathsf{md}(\langle \tau, \tau' \rangle) = \mathbf{!}$ if $\mathsf{md}(\tau) = \mathbf{!}$ and $\mathsf{md}(\langle \tau, \tau' \rangle) = \updownarrow$ if $\mathsf{md}(\tau) = \downarrow$. We often write τ^p if $\mathsf{md}(\tau) = p$.

Quantifications bind type variables in pairs, so that both X and $\overline{\mathrm{X}}$ in τ are bound in $\forall \mathrm{X}.\tau$ and $\exists \mathrm{X}.\tau$. This extends to type substitution (which should always respect action modes), e.g. $(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{\mathbf{!}}[\tau/\mathrm{X}]$ is $(\overline{\tau}(\tau)^{\uparrow})^{\mathbf{!}}$. $\mathsf{ftv}(\tau)$ is the set of free type variables in τ, automatically including their duals. τ is *closed* if $\mathsf{ftv}(\tau) = \emptyset$.

3.3 Typing

We present a polymorphic type discipline based on implicit typing. The full version [6] explores different presentations and variants, including explicitly typed ones. The sequents have the form $\vdash_{\phi} P \triangleright A$, where A is an *action type*, a finite map from names to channel types, and ϕ is an *IO-mode*, which is either I or O. In $\vdash_{\phi} P \triangleright A$, A assign types to free names in P, while ϕ indicates either P has an active thread (O) or not (I). We use the following operations and relations:

- $\odot$ on IO-modes is a partial operation given by: $\mathrm{I} \odot \mathrm{I} = \mathrm{I}$ and $\mathrm{I} \odot \mathrm{O} = \mathrm{O} \odot \mathrm{I} = \mathrm{O}$ (note $\mathrm{O} \odot \mathrm{O}$ is not defined). When $\phi_1 \odot \phi_2$ is defined we write $\phi_1 \asymp \phi_2$.
- $\odot$ on channel types is the least commutative partial operation such that: (1) $\tau_{\mathrm{I}} \odot \overline{\tau_{\mathrm{I}}} = \langle \tau_{\mathrm{I}}, \overline{\tau_{\mathrm{I}}} \rangle$ and (2) $\tau \odot \tau = \tau$ and $\langle \overline{\tau}, \tau \rangle \odot \tau = \langle \overline{\tau}, \tau \rangle$ ($\mathsf{md}(\tau) = \mathbf{?}$). Then $A \asymp B$ iff $\tau' \odot \tau''$ is defined whenever $x : \tau' \in A$ and $x : \tau'' \in B$. If $A \asymp B$, then we set $A \odot B = (A \backslash B) \cup (B \backslash A) \cup \{x : \tau \mid x : \tau' \in A, x : \tau'' \in B, \tau = \tau' \odot \tau''\}$.

(Zero)
$$\frac{-}{\vdash_{\mathtt{I}} \mathbf{0} \triangleright \emptyset}$$

(Par)
$$\frac{\vdash_{\phi_i} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2 \quad \phi_1 \asymp \phi_2}{\vdash_{\phi_1 \odot \phi_2} P_1 \mid P_2 \triangleright A_1 \odot A_2}$$

(Res)
$$\frac{\vdash_{\phi} P \triangleright A, x:\tau \qquad \mathsf{md}(\tau) \in \{!, \updownarrow\}}{\vdash_{\phi} (\boldsymbol{\nu}\, x)P \triangleright A}$$

(Weak)
$$\frac{\mathsf{md}(\tau) \in \{?, \updownarrow\} \qquad \vdash_{\phi} P \triangleright A^{\text{-}x}}{\vdash_{\phi} P \triangleright A, x:\tau}$$

($\mathsf{In}^{\downarrow}$) $(\mathrm{X}_i \notin \mathsf{ftv}(A))$
$$\frac{\vdash_{\mathtt{O}} P \triangleright \vec{y}:\vec{\tau}, \uparrow ?A^{\text{-}x}}{\vdash_{\mathtt{I}} x(\vec{y}).P \triangleright x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{\downarrow}, A}$$

($\mathsf{In}^{!}$) $(\mathrm{X}_i \notin \mathsf{ftv}(A))$
$$\frac{\vdash_{\mathtt{O}} P \triangleright \vec{y}:\vec{\tau}, ?A^{\text{-}x}}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{!}, A}$$

(Out) $(\mathrm{X}_i \notin \mathsf{ftv}(\vec{\tau}'))$
$$\frac{\tau_i' = \tau_i[\rho_i/\mathrm{X}_i]}{\vdash_{\mathtt{O}} \overline{x}\langle \vec{y} \rangle \triangleright x:\exists \vec{\mathrm{X}}.(\vec{\tau})^{p_{\mathrm{O}}}, \vec{y}:\overline{\vec{\tau}'}}$$

Fig. 1. Polymorphic Sequential Typing

$\phi_1 \asymp \phi_2$ ensures that a well-typed process has at most one thread, while $A \asymp B$ guarantees determinism. A, B is the union of A and B, assuming their domains are disjoint; $A^{\text{-}x}$ means A such that $x \notin \mathsf{fn}(A)$; and $\vec{p}A$ indicates $\mathsf{md}(A) \subset \{\vec{p}\}$.

The typing rules are given in Figure 1, which follow structure of processes except (Weak). In (Out), we assume $y_i = y_j$ implies $\tau_i = \tau_j$, $\rho_i = \rho_j$ and $\mathrm{X}_i = \mathrm{X}_j$. $\overline{\vec{\tau}}$ is the pointwise dualisation of $\vec{\tau}$. In comparison with the first-order affine typing [7], the only difference is introduction of quantifiers in ($\mathsf{In}^{\downarrow,!}$) and (Out), each with a natural variable condition. This prefix-wise quantification, close to the one adopted in [30], quantifies only input types (resp. output types) universally (resp. existentially). More general forms of polymorphic typing exist, which are studied in [6]: this form however has the merit in that it is syntactically tractable while harnessing enough expressive power for many practical purposes. Below we list simple examples of polymorphic processes (expressions are from Section 2), followed by a basic syntactic result. Henceforth $\twoheadrightarrow$ stands for $\equiv \cup \longrightarrow^*$.

Example 1. 1. Let $\mathbb{I} \stackrel{\text{def}}{=} x:\forall \mathrm{X}.(\overline{\mathrm{X}}^{?}(\mathrm{X}^{!})^{\uparrow})^{!}$. Then $\vdash_{\mathtt{I}} \mathsf{id}\langle x \rangle \triangleright x:\mathbb{I}$.

2. $\vdash_{\mathtt{I}}$ $\mathsf{t}\langle x \rangle \triangleright x : \mathbb{B}$, $\vdash_{\mathtt{I}}$ $\mathsf{f}\langle x \rangle \triangleright x : \mathbb{B}$ and $\vdash_{\mathtt{I}}$ $\mathsf{not}\langle xy \rangle \triangleright x : \mathbb{B}, y : \overline{\mathbb{B}}$. Further let $\mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \stackrel{\text{def}}{=} \overline{x}(b_1 b_2 z)(!b_1(\vec{v}a).P_1 | !b_2(\vec{v}a).P_2 | z(b).\overline{b}\langle \vec{v}a \rangle)$ assuming $\vdash_{\mathtt{O}} P_{1,2} \triangleright \vec{v}:\vec{\tau}^{?}, a:\tau^{\uparrow}$. Then $\vdash_{\mathtt{O}}$ $\mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright x:\overline{\mathbb{B}}, \vec{v}:\vec{\tau}^{?}, a:\tau^{\uparrow}$. We can check $\mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid \mathsf{t}\langle x \rangle \twoheadrightarrow P_1 | \mathsf{t}\langle x \rangle | (\boldsymbol{\nu}\, b_2) !b_2(\vec{v}a).P_2 \approx P_1 | \mathsf{t}\langle x \rangle$ where $\approx$ is the standard (untyped) weak bisimilarity.

Proposition 1. (subject reduction) $\vdash_{\phi} P \triangleright A$ *and* $P \twoheadrightarrow P'$ *imply* $\vdash_{\phi} P' \triangleright A$.

3.4 Extension (1): Linearity

The first-order linear typing [32] refines the affine type discipline [7] by adding causality edges between typed names in an action type. Edges prevent circular causality. For example, $a.\overline{b} \mid b.\overline{a}$ is typable in the affine system, but not in the linear one. Its second-order extension simply adds prefix-wise quantification to

[32]. Thus, in Figure 1, the input prefix rules become, with $x:\tau \rightarrow A$ denoting the result of adding a new edge from $x:\tau$ to each maximal node in A:

$$\frac{(\mathsf{In}^{\downarrow})\quad (\mathrm{X}_i \notin \mathsf{ftv}(A,B)) \qquad \vdash_{\mathsf{O}} P \triangleright \vec{y}:\vec{\tau}\,, \uparrow A^{\text{-}x}, ?B^{\text{-}x}}{\vdash_{\mathsf{I}} x(\vec{y}).P \triangleright (x:\forall\vec{\mathrm{X}}.(\vec{\tau})^{\downarrow} \rightarrow A), B} \qquad \frac{(\mathsf{In}^{!})\quad (\mathrm{X}_i \notin \mathsf{ftv}(A)) \qquad \vdash_{\mathsf{O}} P \triangleright \vec{y}:\vec{\tau}, ?A^{\text{-}x}}{\vdash_{\mathsf{I}} !x(\vec{y}).P \triangleright x:\forall\vec{\mathrm{X}}.(\vec{\tau})^{!} \rightarrow A}$$

Further $\asymp$ and $\odot$ in (Par) are refined to prohibit circularity of causal chains following [32]. The resulting system preserves all key properties of the first-order linear typing, but with greater typability. We state one of the central results.

Theorem 1. (strong normalisability) *Let* $\vdash_{\phi} P \triangleright A$ *in linear polymorphic typing. Then* P *is strongly normalising with respect to* $\longrightarrow$*.*

3.5 Extension (2): State and Concurrency

The integration of imperative features and polymorphism is an old and challenging technical problem [10, 16, 29]. Here we present a basic extension of affine polymorphic processes to stateful computation. Following [12], we add a constant process $\mathsf{Ref}\langle xy\rangle$, called *reference agent.* For interacting with reference, we need *selection* $\overline{x}\mathtt{in}_i\langle\vec{z}\rangle$ which selects, in the case of reference, either read ($i = 1$) or write ($i = 2$). For reduction we have:

$$\mathsf{Ref}\langle xy\rangle|\overline{x}\mathtt{in}_1\langle c\rangle \longrightarrow \mathsf{Ref}\langle xy\rangle|\overline{c}\langle y\rangle \qquad \mathsf{Ref}\langle xy\rangle|\overline{x}\mathtt{in}_2\langle zc\rangle \longrightarrow \mathsf{Ref}\langle xz\rangle|\overline{c}$$

The first rule describes reading of the content y, the second one writing of a new content z. A significant property of reference agents is that, in combination with replication, they can represent a large class of stateful computation [2, 12].

For types, we add the mutable replication mode $!_{\mathrm{M}}$ and its dual $?_{\mathrm{M}}$, as well as adding $[\&_i\vec{\tau}_{\mathsf{O}i}]^{p_{\mathsf{I}}}$ for input types and $[\oplus_i\vec{\tau}_{\mathsf{I}i}]^{p_{\mathsf{O}}}$ for output types. For example, the type of a reference with values of type τ is $[(\tau)^{\uparrow}\&\overline{\tau}()^{\uparrow}]^{!_{\mathrm{M}}}$, which we write $\mathsf{ref}(\tau)$. There are several ways to incorporate polymorphism into mutable types. Here we present a most basic form. Let us say $\forall\mathrm{X}.\tau$ (resp. $\exists\mathrm{X}.\tau$) is *simple* when $\mathsf{md}(\tau) \neq\, !_{\mathrm{M}}$ (resp. $\mathsf{md}(\tau) \neq\, ?_{\mathrm{M}}$). We then restrict the set of polymorphic types which we consider to the simple ones, and introduce the following typing rules.

$$\frac{(\mathsf{Ref}) \qquad \mathsf{md}(\tau) \in \{!, !_{\mathrm{M}}\}}{\vdash_{\mathsf{I}} \mathsf{Ref}\langle xy\rangle \triangleright x:\mathsf{ref}(\tau), y:\overline{\tau}} \qquad \frac{(\mathsf{Sel}) \qquad -}{\vdash_{\mathsf{O}} \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright x:[\oplus_i\vec{\tau}_i]^{p_{\mathsf{O}}}, \vec{y}:\vec{\tau}_i} \qquad \frac{(\mathsf{In}^{!_{\mathrm{M}}}) \qquad \vdash_{\mathsf{O}} P \triangleright \vec{y}:\vec{\tau}, ?_{\mathrm{M}}?A^{\text{-}x}}{\vdash_{\mathsf{I}} !x(\vec{y}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{M}}}, A}$$

Note ($\mathsf{In}^{!_{\mathrm{M}}}$) allows a replicated prefix to suppress $?_{\mathrm{M}}$-actions, unlike (In$^{!}$). Also note the subject of a reference/$!_{\mathrm{M}}$-typed replication is never universally abstracted, in accordance with restriction to simple types. In spite of this limitation, a wide variety of imperative polymorphic programs are typable via encoding: for example, all benchmark programs in Leroy's thesis [16] as well as Grossman's integrations of `struct` with existentials [10] are typable. This is due to the distinction between two replicated types, $!$ and $!_{\mathrm{M}}$. For further discussions, see [6]. For incorporating concurrency, we simply ignore all IO-modes in each rule. Section 6 presents equational reasoning for stateful polymorphic processes.

4 Contextual Congruence and Parametericity

This section presents a sound and complete characterisation of the contextual congruence by a second-order logical relation for the affine polymorphic π-calculus. As a consequence we obtain relational parametricity [28] and simulation principle [19, 27], the two fundamental principles for polymorphic λ-calculi.

The contextual congruence for affine polymorphic processes is defined following its first-order counterpart [7]. Write $\mathbb{O}$ for $()^{\uparrow}$ and write $P \Downarrow_x$ when $P \longrightarrow^* (\boldsymbol{\nu}\, \vec{z})(\overline{x}\langle\vec{y}\rangle|P')$ with $x \notin \{\vec{z}\}$. Then $\cong_{\forall\exists}$ is the maximum typed congruence over polymorphic processes satisfying

$$\vdash_{\mathsf{0}} P_1 \cong_{\forall\exists} P_2 \triangleright x\!:\!\mathbb{O} \Leftrightarrow (P_1 \Downarrow_x \Leftrightarrow P_2 \Downarrow_x).$$

for all $\vdash_{\mathsf{0}} P_{1,2} \triangleright x\!:\!\mathbb{O}$. We write $P \cong_{\forall\exists}^{A,\phi} Q$ if P and Q are related by $\cong_{\forall\exists}$ under A, ϕ (and often omit ϕ or A, ϕ). We can easily check that $\equiv \cup \longrightarrow \subsetneq \cong_{\forall\exists}$.

We first consider logical relations in a simple shape. Given closed types $\tau_{1,2}$ with $\mathsf{md}(\tau_1) = \mathsf{md}(\tau_2) \in \{\uparrow, \mathbf{!}\}$, a *typed relation* $\mathfrak{R} : \tau_1 \leftrightarrow \tau_2$ is a family of binary relations $\{\mathfrak{R}_x\}_{x\in\mathcal{N}}$ over typed processes such that: (1) if $P_1\mathfrak{R}_x P_2$ then $\vdash_\phi P_i \triangleright x\!:\!\tau_i$ with $\phi = \mathtt{I}$ (resp. $\phi = \mathtt{0}$) if $\mathsf{md}(\tau_i) = \mathbf{!}$ (resp. $\mathsf{md}(\tau_i) = \uparrow$) and (2) the family is closed under injective renaming, i.e. $P\mathfrak{R}_x Q$ iff $P\binom{xy}{yx}\mathfrak{R}_y\, Q\binom{xy}{yx}$.

Given a typed relation $\mathfrak{R} : \tau_1 \leftrightarrow \tau_2$, the *dual of* $\mathfrak{R}$ *at* xu, written $\mathfrak{R}^{\perp}_{xu}$, is a relation from processes of type $x\!:\!\overline{\tau_1}, u\!:\!\mathbb{O}$ to those of type $x\!:\!\overline{\tau_2}, u\!:\!\mathbb{O}$, satisfying: $P_1\mathfrak{R}^{\perp}_{xu}P_2$ iff $(\boldsymbol{\nu}\, x)(P_1|R_1)\Downarrow_u \Leftrightarrow (\boldsymbol{\nu}\, x)(P_2|R_2)\Downarrow_u$ for each $R_1\mathfrak{R}_x R_2$. The resulting relations, called *typed co-relations*, are also taken modulo injective renaming, so that we simply write $\mathfrak{R}^{\perp}$ for the dual of $\mathfrak{R}$. Symmetrically we define the dual of a co-relation, returning to a typed relation. A $\perp\perp$*-closed relation* is a typed relation closed under double negation, i.e. $\mathfrak{R}$ such that $\mathfrak{R}^{\perp\perp} = \mathfrak{R}$.

We can now define logical relations as interpretation of open types under a *relational environment*, i.e. a function which maps type variables to $\perp\perp$-closed relations respecting action modes. The interpretation is written $(\!(\tau)\!)_\xi$ where ξ is a relational environment.

$$(\!((\tau_1^{?}..\tau_n^{?}\rho^{\uparrow})^{!})\!)_\xi \stackrel{\text{def}}{=} ((\!(\overline{\tau_1})\!)_\xi..(\!(\overline{\tau_n})\!)_\xi(\!(\rho)\!)_\xi)^{!} \qquad (\!((\tau_1..\tau_n)^{\uparrow})\!)_\xi \stackrel{\text{def}}{=} ((\!(\tau_1)\!)_\xi..(\!(\tau_n)\!)_\xi)^{\uparrow}$$

$$(\!(\mathrm{X})\!)_\xi \stackrel{\text{def}}{=} \xi(\mathrm{X}) \quad (\!(\forall\mathrm{X}.\tau)\!)_\xi \stackrel{\text{def}}{=} \forall\mathrm{X}.\lambda\mathfrak{R}.(\!(\tau)\!)_{\xi\cdot\mathrm{X}\mapsto\mathfrak{R}^{\perp\perp}} \quad (\!(\exists\mathrm{X}.\tau)\!)_\xi \stackrel{\text{def}}{=} \exists\mathrm{X}.\lambda\mathfrak{R}.(\!(\tau)\!)_{\xi\cdot\mathrm{X}\mapsto\mathfrak{R}^{\perp\perp}}$$

Above, the right-hand side of each definition uses a type-respecting function on typed relations, given in the following (definitions are presented for simpler shapes for legibility, with obvious generalisations).

$$\begin{aligned}
(\mathfrak{R}_1^{?}\mathfrak{R}_2^{\uparrow})^{!}_x &\stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid Q\ \mathfrak{R}_{1y}\ Q' \supset P \circ \overline{x}\langle yz\rangle \circ Q\ \mathfrak{R}_{2z}\ P' \circ \overline{x}\langle yz\rangle \circ Q'\} \\
(\mathfrak{R}^{!})^{\uparrow}_x &\stackrel{\text{def}}{=} \{\langle \overline{x}\langle y\rangle \circ Q,\ \overline{x}\langle y\rangle \circ Q'\rangle \mid Q\mathfrak{R}_y Q'\}^{\perp\perp} \\
\forall\mathrm{X}.\mathfrak{R}[\mathrm{X}]_x &\stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid \forall\mathfrak{R}'.P\ \mathfrak{R}[\mathfrak{R}']_x\ P'\} \\
\exists\mathrm{X}.\mathfrak{R}[\mathrm{X}]_x &\stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid \exists\mathfrak{R}'.\ P\ \mathfrak{R}[\mathfrak{R}']_x\ P'\ \}^{\perp\perp},
\end{aligned}$$

where all mentioned processes, substitutions etc. should be appropriately typed. $P \circ Q$ denotes $(\boldsymbol{\nu}\, \mathsf{fn}(P) \cap \mathsf{fn}(Q))(P|Q)$. $\mathfrak{R}[\mathrm{X}]$ indicates a type-respecting map over typed relations.[1] These rules can be read quite like logical relations for functions: for example, the first rule says that, if a pair of "resources" are related, then the corresponding pair of "results" should also be related. In fact, the construction yields, via encoding, logical relations in the usual sense for both call-by-name and call-by-value polymorphic PCF-like calculi, cf. [25, 26]. Since each rule returns a $\perp\perp$-closed relation whenever its arguments are, $(\!(\tau)\!)_\xi$ is always $\perp\perp$-closed.

The above logical relation only relates processes with a single free name. For equating processes with multiple free names, we extend logical relations to action types which are *connected* in the following sense.

Definition 1. *(A, ϕ) is* connected *if one the following holds.*

- *$\phi = \mathtt{I}$ and A contains, in its range, either a unique $!$-type and zero or more $?$-types, or a unique $\downarrow$-type, a unique $\uparrow$-type and zero or more $?$-types.*
- *$\phi = \mathtt{O}$ and A contains a unique $\uparrow$-type and zero or more $?$-types.*

If (A, ϕ) is connected, the name with the unique $\uparrow/!$ type is its principal port.

Connectedness has both practical and theoretical significance. First, in many practical examples including the embedding of programming languages, it is often enough to consider processes of connected types. Second, any process of an arbitrary action type can always be decomposed canonically into connected processes, so that results about connected processes often easily extend to non-connected processes. We now generalise the logical relation to connected types.

Definition 2. *Let (A, ϕ) be connected with principal port $x : \tau$ and let $\mathsf{fn}(A) \setminus \{x\} = \{y_j\}_{j \in J}$. Then $\cong_{\mathcal{L}}^{A,\phi}$ is a relation on processes of type (A, ϕ) which relates P and P' iff, for each ξ ($\prod_{j\in J} P_j$ denotes a parallel composition of $\{P_j\}_{j\in J}$),*

$$(\forall j \in J.\ Q_j\ (\!(\overline{A(y_j)})\!)_{\xi, y_j} Q'_j) \ \supset\ (\boldsymbol{\nu}\,\vec{y})(P \,|\, \Pi_{j\in J} Q_j)\ (\!(\tau)\!)_{\xi,x}\ (\boldsymbol{\nu}\,\vec{y})(P' \,|\, \Pi_{j\in J} Q'_j).$$

Note that $\cong_{\mathcal{L}}^{x:\tau,\phi}$ (with ϕ given corresponding to τ) coincides with $(\!(\tau)\!)_x$. The following result is proved closely following the development by Pitts [25, 26].

Theorem 2. (characterisation of $\cong_{\forall\exists}$) *$\cong_{\mathcal{L}}^{A,\phi} = \cong_{\forall\exists}^{A,\phi}$ for each connected (A, ϕ).*

Corollary 1. 1. (parametricity) *$P \cong_{\forall\exists}^{x:\forall\mathrm{X}.\tau} Q$ if and only if $P (\!(\tau)\!)_{\mathrm{X}\mapsto\mathfrak{R}} Q$ for each $\perp\perp$-closed $\mathfrak{R}$.*
2. (simulation) *$P \cong_{\forall\exists}^{x:\exists\mathrm{X}.\tau} Q$ if and only if $P (\!(\tau)\!)_{\mathrm{X}\mapsto\mathfrak{R}} Q$ for some $\perp\perp$-closed $\mathfrak{R}$.*

The construction and results extend to the whole set of affine polymorphic processes, see [6]. The same characterisation result also holds for linear polymorphic processes, where we use a $\perp\perp$-closure based on convergence to a specific boolean value (this convergence is also used for defining the contextual congruence, which is necessary since linear processes are always converging). In Section 6 we give reasoning examples which use these results. The corresponding characterisation results for non-functional polymorphic behaviour (including state [24] and control) are left as an open issue.

[1] In detail: $\mathfrak{R}[\mathrm{X}]$ should map, for fixed τ and τ' such that $\mathsf{ftv}(\tau) \cup \mathsf{ftv}(\tau') \subset \{\mathrm{X}\}$, each $\mathfrak{R}' : \rho \leftrightarrow \rho'$ of mode $\mathsf{md}(\mathrm{X})$ to a typed relation $\mathfrak{R}[\mathfrak{R}'] : \tau[\rho/\mathrm{X}] \leftrightarrow \tau'[\rho'/\mathrm{X}]$.

5 Generic Transitions and Innocence

This section discusses another basic element of the present theory, a generic labelled transition system and the induced process equivalence. While our presentation focusses on the affine polymorphic π-calculus, the construction equally applies to linear, stateful and concurrent polymorphic processes, with the same soundness result. The duality principle strongly guides the construction. The set of action labels $(l, l', \ldots)$ are given by:

$$l \ ::= \ x\langle(\vec{y})\vec{w}\rangle \mid \overline{x}\langle(\vec{y})\vec{w}\rangle \mid \boldsymbol{\tau}$$

In the first two labels, names in $\vec{y}$ are pairwise distinct and $\vec{y}$ is a (not necessarily consecutive) subsequence of $\vec{w}$ (called *objects*) and distinct from x (called *subject*). Names in $\vec{y}$ occur *bound*, while all other names occur *free*. $x(\vec{y})$ and $x\langle\vec{y}\rangle$ stand for $x\langle(\vec{y})\vec{y}\rangle$ and $x\langle(\varepsilon)\vec{y}\rangle$, respectively and similarly for output actions.

Transitions use an extended typing where type variables in action types are annotated by quantification symbols (as $\mathrm{X}^{\forall}$ and $\mathrm{X}^{\exists}$, called *universal type variable* and *existential type variable*, respectively). The original free type variables and $\forall$-quantified variables are naturally $\forall$-annotated, while $\exists$-quantified variables are $\exists$-annotated. Free $\exists$-type variables are introduced by the following added rule:

$$(\exists\text{-Var})\ \frac{\vdash_{\phi} P \triangleright A[\tau/\mathrm{X}^{\exists}]}{\vdash_{\phi} P \triangleright A}$$

which we assume to be applicable only as the last rule(s) in a derivation. As an example of typing, we have $\vdash_{\mathsf{O}} \mathsf{t}\langle y\rangle | z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle \triangleright y : \mathrm{X}^{\exists}, z : (\overline{\mathrm{X}^{\exists}})^{\downarrow}, e : (\mathbb{B})^{\uparrow}$, abstracting away the type which is both for the resource at y and for the value of the input via z. Using annotated type variables, the following predicates decide if the shape of action labels conforms to a given action type. In brief, they say that free output (resp. input) corresponds to universal type variables (resp. existential type variables), cf. Section 2. θ below denotes a sequence of quantifiers.

Definition 3. *1.* $A \vdash \boldsymbol{\tau}$ *always.*

2. $A \vdash x\langle(\vec{z})\vec{w}\rangle : \theta(\vec{\tau})^{p_{\mathrm{I}}}$ *when* $\{\vec{z}\} \cap \mathsf{fn}(A) = \emptyset$ *and* $A(x) = \theta(\vec{\tau})^{p_{\mathrm{I}}}$ *s.t.* $w_i \notin \{\vec{z}\}$ *iff* $A(w_i) = \overline{\tau_i}$ *where* τ_i *is an existential type variable.*

3. $A \vdash \overline{x}\langle(\vec{z})\vec{w}\rangle : \theta(\vec{\tau})^{p_{\mathrm{O}}}$ *when* $\{\vec{z}\} \cap \mathsf{fn}(A) = \emptyset$ *and* $A(x) = \theta(\vec{\tau})^{p_{\mathrm{O}}}$ *s.t.* $w_i \notin \{\vec{z}\}$ *iff* $A(w_i) = \overline{\tau_i}$ *where* τ_i *is a universal type variable.*

We can now introduce the transition rules (for expository purposes we focus on key instances). We start from the standard bound input.

$$(\mathsf{BIn}^{\downarrow}) \quad \vdash_{\mathtt{I}} x(\vec{y}).P \triangleright A \ \xrightarrow{x(\vec{y})} \ \vdash_{\mathsf{O}} P \triangleright A/x, \vec{y}{:}\vec{\tau} \qquad (A \vdash x(\vec{y}) : \theta(\vec{\tau})^{\downarrow})$$

This may introduce (output-moded) $\forall$-type variables, which are used as follows.

$$(\mathsf{FOut}^{\uparrow}) \quad \vdash_{\mathsf{O}} \overline{x}\langle\vec{y}\rangle \triangleright A \ \xrightarrow{\overline{x}\langle\vec{y}\rangle} \ \vdash_{\mathtt{I}} \mathbf{0} \triangleright A/x \qquad (A \vdash \overline{x}\langle\vec{y}\rangle : \theta(\vec{\mathrm{X}}^{\forall})^{\uparrow})$$

We can now infer $\vdash_{\mathtt{I}} \mathsf{id}\langle x\rangle \triangleright x{:}\mathbb{I} \xrightarrow{x(yz)}\xrightarrow{\overline{z}\langle y\rangle} \vdash_{\mathtt{I}} \mathsf{id}\langle x\rangle \triangleright x{:}\mathbb{I}$ (using a replicated version for input). Next we consider the dual situation, starting from bound output.

$$(\mathsf{BOut}^{\uparrow})\ \ \vdash_{\mathtt{0}} \overline{x}\langle\vec{y}\rangle \triangleright A \ \xrightarrow{\overline{x}(\vec{z})}\ \vdash_{\mathtt{I}} \Pi_i[z_i \to y_i]^{\tau_i} \triangleright A/x, \vec{z} : \vec{\tau} \quad (A \vdash \overline{x}(\vec{z}) : \theta(\vec{\overline{\tau}})^{\uparrow})$$

Here $[z_i \to y_i]^{\tau_i}$ is the standard copy-cat agent [7,12,14,32,33]. For example, $[a \to b]^{(()^{\uparrow})^!} \stackrel{\text{def}}{=} !a(y).\overline{b}(y')y'.\overline{y}$. This rule is best seen in view of the semantic equality $\overline{x}\langle\vec{y}\rangle \cong \overline{x}(\vec{z})\Pi_i[z_i \to y_i]^{\tau_i}$. Again this rule may introduce (input-moded) $\exists$-type variables, used by:

$$(\mathsf{FIn}^{\downarrow})\ \ \vdash_{\mathtt{I}} x(\vec{y}).P \triangleright A \ \xrightarrow{x\langle\vec{z}\rangle}\ \vdash_{\mathtt{0}} P\{\vec{z}/\vec{y}\} \triangleright A/x \odot \vec{z}{:}\vec{\mathrm{X}} \quad (A \odot \vec{z}{:}\vec{\overline{\mathrm{X}}}^{\exists} \vdash x\langle\vec{z}\rangle : \theta(\vec{\mathrm{X}}^{\exists})^{\downarrow})$$

In the side condition, we compose types for opaque resources to appear in a later derivation. The rule says an input may receive channels for opaque resources which have been exported and which are, therefore, free. We can now infer $\vdash_{\mathtt{0}} \overline{x}(yz)(\mathsf{t}\langle y\rangle | z(w).R) \triangleright x{:}\overline{\mathbb{I}}, e{:}(\mathbb{B})^{\uparrow} \xrightarrow{\overline{x}(yz)}\xrightarrow{z\langle y\rangle} \vdash_{\mathtt{0}} \mathsf{t}\langle y\rangle | R\{y/w\} \triangleright y{:}\langle \mathrm{X}^{\exists}, \overline{\mathrm{X}}^{\exists}\rangle, e{:}(\mathbb{B})^{\uparrow}$.

Since a type may carry both type variable(s) and concrete type(s), the general rule for linear input (resp. output) combines $(\mathsf{BIn}^{\downarrow})$ and $(\mathsf{FIn}^{\downarrow})$ (resp. $(\mathsf{BOut}^{\uparrow})$ and $(\mathsf{FOut}^{\uparrow})$). Similarly we have rules for replicated input/output, as well as standard composition rules. For the generated transition relation we can check, under the extended typing:

Proposition 2. *If* $\vdash_{\phi} P \triangleright A$ *and* $\vdash_{\phi} P \triangleright A \xrightarrow{l} \vdash_{\phi'} Q \triangleright B$ *then* $\vdash_{\phi'} Q \triangleright B$.

Define the weak bisimilarity $\approx_{\forall\exists}$ induced by generic transitions in the standard way. The proof of the following is then straightforward.

Proposition 3. (soundness) $\vdash_{\phi} P \approx_{\forall\exists} Q \triangleright A$ *implies* $\vdash_{\phi} P \cong_{\forall\exists} Q \triangleright A$.

The result extends to the linear/stateful extensions in §3.4/5. Further the analogue of Corollary 1(1) (parametricity) easily holds for $\approx_{\forall\exists}$. We can also show polymorphic transition sequences of a typed process can be characterised by an innocent function as in the first-order affine processes [7]. Again as in [7], finite generic innocent functions are always realisable as syntactic processes.

6 Reasoning Examples

This section discusses equational reasoning based on the theories in Sections 4 and 5, and outlines a fully abstract embedding of System F.

Inhabitation Results. We begin with an inhabitation result for $\mathbb{I}$ using generic transitions. Let $\Omega\langle x\rangle \stackrel{\text{def}}{=} !x(yz).(\boldsymbol{\nu}\, ab)(!a(w).\overline{b}\langle w\rangle | !b(w).\overline{a}\langle w\rangle | \overline{a}(c)c.\overline{z}\langle y\rangle)$ (which diverges after the initial input; from now on, the notation $\Omega\langle x\rangle$ is used for denoting such processes regardless of types). We prove that $\vdash_{\mathtt{I}} P \triangleright x{:}\mathbb{I}$ implies either $P \approx_{\forall\exists} \mathsf{id}\langle x\rangle$ or $P \approx_{\forall\exists} \Omega\langle x\rangle$. Let $\vdash_{\mathtt{I}} P \triangleright x : \mathbb{I}$. Then we have $\vdash_{\mathtt{I}} P \triangleright x :$

$\mathbb{I} \xrightarrow{x(yz)} \vdash_{\mathsf{0}} P' \triangleright x\,{:}\,\mathbb{I}, y\,{:}\,\mathrm{X}^{\forall}, z\,{:}\,(\mathrm{X}^{\forall})^{\uparrow}$. By inspecting the action type, if P' ever has an output, it can only be $\overline{z}\langle y\rangle$, in which case $P \approx_{\forall\exists} \mathsf{id}\langle x\rangle$. If not then $P \approx_{\forall\exists} \Omega\langle x\rangle$. Since $\mathsf{id}\langle x\rangle \not\approx_{\forall\exists} \Omega\langle x\rangle$, these two are all distinct inhabitants of the type. Similarly we can check $x : \mathbb{B}$ is inhabited by $\mathsf{t}\langle x\rangle$, $\mathsf{f}\langle x\rangle$ and $\Omega\langle x\rangle$. In the linear typing, we obtain the same results except we lose $\Omega\langle x\rangle$ by totality of transition.

Boolean ADTs. Next we show a simple use of logical relations for equational reasoning, taking abstract data types of opaque booleans (similar to those discussed in [22, 25]). The data type should export a "flip", or negation operation and allow reading (which means turning an opaque boolean to a concrete one). Two simple implementations in the λ-calculus with records are:

$$M \stackrel{\mathrm{def}}{=} \mathtt{pack\ bool}\ \{\mathtt{bit} = \mathsf{T},\ \mathtt{flip} = \lambda x{:}\mathtt{bool}.\neg x,\ \mathtt{read} = \lambda x{:}\mathtt{bool}.x\ \}\ \mathtt{as}\ \mathit{bool}$$
$$M' \stackrel{\mathrm{def}}{=} \mathtt{pack\ bool}\ \{\mathtt{bit} = \mathsf{F},\ \mathtt{flip} = \lambda x{:}\mathtt{bool}.\neg x,\ \mathtt{read} = \lambda x{:}\mathtt{bool}.\neg x\}\ \mathtt{as}\ \mathit{bool}$$

where $\mathit{bool} \stackrel{\mathrm{def}}{=} \exists \mathrm{X}.\{\mathtt{bit} : \mathrm{X},\ \mathtt{flip} : \mathrm{X} \to \mathrm{X}, \mathtt{read} : \mathrm{X} \to\ \mathtt{bool}\}$. M and M' can be encoded as (using a call-by-value translation of products, cf. [32]):

$$\mathtt{bool}\langle u\rangle \stackrel{\mathrm{def}}{=} \overline{u}(m_1 m_2 m_3)(Q_1|Q_2|Q_3) \qquad \mathtt{bool}'\langle u\rangle \stackrel{\mathrm{def}}{=} \overline{u}(m_1 m_2 m_3)(Q'_1|Q'_2|Q'_3)$$

where $Q_1 \stackrel{\mathrm{def}}{=} \mathsf{t}\langle m_1\rangle$, $Q_2 \stackrel{\mathrm{def}}{=} !m_2(bz).\overline{z}(b')\mathsf{not}\langle b'b\rangle$, $Q_3 \stackrel{\mathrm{def}}{=} !m_3(bz).\overline{z}\langle b\rangle$, $Q'_1 \stackrel{\mathrm{def}}{=} \mathsf{f}\langle m_1\rangle$, $Q'_2 \equiv Q_2$ and $Q'_3 \stackrel{\mathrm{def}}{=} !m_3(bz).\overline{z}(b')\mathsf{not}\langle b'b\rangle$. We can easily check these processes are typable under $u{:}\exists \mathrm{X}.\mathcal{B}[\mathrm{X}]$, where $\mathcal{B}[\mathrm{X}] \stackrel{\mathrm{def}}{=} (\mathrm{X}(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}(\overline{\mathrm{X}}(\mathbb{B})^{\uparrow})^{!})^{\uparrow}$.

We now show $\vdash_{\mathtt{I}} \mathtt{bool}\langle u\rangle \cong_{\forall\exists} \mathtt{bool}'\langle u\rangle \triangleright x : \exists \mathrm{X}.\mathcal{B}[\mathrm{X}]$. By Corollary 1(2), it is enough to establish $\mathtt{bool}\langle u\rangle (\!(\mathcal{B}[\mathrm{X}])\!)_{x,\mathrm{X}\mapsto\mathfrak{R}} \mathtt{bool}'\langle u\rangle$ for some $\bot\bot$-closed $\mathfrak{R}$. By definition this means we have to verify:

$$Q_1 \mathfrak{R}_{m_1} Q'_1, \quad Q_2 (\overline{\mathfrak{R}}(\mathfrak{R})^{\uparrow})^{!}_{m_2} Q'_2, \quad Q_3 (\overline{\mathfrak{R}}((\!(\mathbb{B})\!))^{\uparrow})^{!}_{m_3} Q'_3.$$

Take $\mathfrak{R} \stackrel{\mathrm{def}}{=} \{(\mathsf{t}\langle x\rangle, \mathsf{f}\langle x\rangle),\ (\mathsf{f}\langle x\rangle, \mathsf{t}\langle x\rangle),\ (\Omega\langle x\rangle, \Omega\langle x\rangle)\}$ (processes are taken up to $\cong_{\forall\exists}$). Then $\mathfrak{R}$ is $\bot\bot$-closed (by the inhabitation result for $\mathbb{B}$). $\mathfrak{R}$ obviously relates Q_1 and Q'_1. The key case is $Q_3 (\overline{\mathfrak{R}}((\!(\mathbb{B})\!))^{\uparrow})^{!}_{m_3} Q'_3$, which means, by definition, $Q_3 \circ \overline{m}_3\langle xw\rangle \circ S\ ((\!(\mathbb{B})\!))^{\uparrow}_{w}\ Q'_3 \circ \overline{m}_3\langle xw\rangle \circ S'$ for any $S\mathfrak{R}S'$. The case when $(S, S') = (\Omega\langle x\rangle, \Omega\langle x\rangle)$ is trivial. Let $(S, S') = (\mathsf{t}\langle x\rangle, \mathsf{f}\langle x\rangle)$. We can check both $Q_3 \circ \overline{m}_3\langle xw\rangle \circ S$ and $Q'_3 \circ \overline{m}_3\langle xw\rangle \circ S'$ reduce to, hence are $\cong_{\forall\exists}$-equivalent to, $\overline{w}\langle b\rangle \circ \mathsf{t}\langle b\rangle$. Now we use Theorem 2. Similarly when $(S, S') = (\mathsf{f}\langle x\rangle, \mathsf{t}\langle x\rangle)$. Reasoning for Q_2 and Q'_2 is similar.

Simple Boolean Agent. In Section 2, we have seen the behaviour of $S \stackrel{\mathrm{def}}{=} \overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(b)\mathsf{not}\langle bw\rangle)$ under $x : \overline{\mathbb{I}}, e : (\mathbb{B})^{\uparrow}$. Noting this process is typable in the linear typing, we show that S and $S' \stackrel{\mathrm{def}}{=} \overline{e}(b)\mathsf{f}\langle b\rangle$ are contextually congruent as linear polymorphic processes. Since S and S' have different visible traces, the use of some extensionality principle is essential. By the characterisation result along the line of Theorem 2 in the linear setting, it suffices to show $(\boldsymbol{\nu}\, x)(S|P)((\!((\mathbb{B})^{\uparrow})\!))_e(\boldsymbol{\nu}\, x)(S'|P)$ for each $\vdash_{\mathtt{I}} P \triangleright x\,{:}\,\mathbb{I}$. But if $\vdash_{\mathtt{I}} P \triangleright x\,{:}\,\mathbb{I}$ then $P \cong_{\forall\exists} \mathsf{id}\langle x\rangle$ by inhabitation. We can then check $(\boldsymbol{\nu}\, x)(S|P) \cong_{\forall\exists} (\boldsymbol{\nu}\, x)(S|\mathsf{id}\langle x\rangle) \approx S' \approx (\boldsymbol{\nu}\, x)(S'|\mathsf{id}\langle x\rangle) \cong_{\forall\exists} (\boldsymbol{\nu}\, x)(S'|P)$, hence done.

Diverging Functions. Another example which needs extensionality, but this time in the context of affine sequential processes, is the equality of two diverging functions treated by Pitts [25] (the example is attributed to Stark). Assume we are given the following two call-by-value functions:

$$F' \overset{\text{def}}{=} \texttt{letrec } f = \lambda g.fg \texttt{ in } f$$
$$G' \overset{\text{def}}{=} \texttt{letrec } f = \lambda g.\texttt{if } g\mathsf{T} \texttt{ then } (\texttt{if } g\mathsf{F} \texttt{ then } fg \texttt{ else } \mathsf{T}) \texttt{ else } fg \texttt{ in } f$$

Let $\mathsf{null}_\lambda \overset{\text{def}}{=} \forall \mathrm{X}.\mathrm{X}$, $\mathbb{B}_\lambda \overset{\text{def}}{=} \forall \mathrm{X}.(\mathrm{X} \Rightarrow \mathrm{X} \Rightarrow \mathrm{X})$ and $\alpha = \exists \mathrm{X}.((\mathrm{X} \Rightarrow \mathbb{B}_\lambda) \Rightarrow \mathbb{B}_\lambda)$. Then we can check $F \overset{\text{def}}{=} \texttt{pack } \mathsf{null}_\lambda, F' \texttt{ as } \alpha$ and $G \overset{\text{def}}{=} \texttt{pack } \mathbb{B}_\lambda, G' \texttt{ as } \alpha$ are well-typed after existential abstraction. To show F and G are equal, we first encode them as affine polymorphic processes. In the standard encoding (with recursion being translated using copy-cats), F and G are represented by, respectively, $\overline{u}(x)\Omega\langle x\rangle$ and $\overline{u}(x)P$ where (using some shorthand notations):

$$P \overset{\text{def}}{=} !x(gz).(\overline{g}(\mathsf{T}w)w(b).\mathsf{if}\ b\ \mathsf{then}\ [\overline{g}(\mathsf{F}w')w'(b').\mathsf{if}\ w'\ \mathsf{then}\ \Omega\langle u\rangle\ \mathsf{else}\ \overline{z}(\mathsf{T})]\ \mathsf{else}\ \Omega\langle u\rangle),$$

both typable under $u : \exists \mathrm{X}.(\tau)^{\uparrow}$ with $\tau = ((\mathrm{X}^{!}(\overline{\mathbb{B}})^{\downarrow})^{?}(\mathbb{B})^{\uparrow})^{!}$. We can then show $P \cong_{\forall\exists} \Omega\langle x\rangle$ using a logical relation $((\mathfrak{R}(\overline{\mathbb{B}})^{\downarrow})^{?}(\mathbb{B})^{\uparrow})^{!}_{x}$ where $\mathfrak{R}_u$ is the universal relation over $u{:}\mathbb{B}$. Detailed reasoning is given in [6].

State and Concurrency. We apply transition-based reasoning to a simple concurrent ADT, a cell with a boolean value. It allows three operations, `share`, `read` and `write`. The first returns the access pointer to the cell, while the latter two read/write a boolean value from it. The data type of this agent is:

$$\texttt{Cell}[\mathbb{B}] \overset{\text{def}}{=} \exists \mathrm{X}.(((\mathrm{X})^{\uparrow})^{!_{\mathrm{M}}}(\overline{\mathrm{X}}(\mathbb{B})^{\uparrow})^{!_{\mathrm{M}}}(\overline{\mathrm{X}}\,\overline{\mathbb{B}}()^{\uparrow})^{!_{\mathrm{M}}})^{\uparrow}.$$

Below we give two implementations. The first is centralised in that all clients have access to a single container; while, in the second, each client has a different proxy which it uses to access the "real" cell. Let

$$\mathsf{cell}\langle ul\rangle \overset{\text{def}}{=} \overline{u}(srw)(S \mid R \mid W) \qquad \mathsf{cell}'\langle ul\rangle \overset{\text{def}}{=} \overline{u}(srw)(S' \mid R' \mid W') \ ,$$

where $S \overset{\text{def}}{=} !s(z).\overline{z}\langle l\rangle$, $R \overset{\text{def}}{=} !r(cz).\overline{c}\texttt{in}_1(e)e(x).\overline{z}\langle x\rangle$ and $W \overset{\text{def}}{=} !w(cbz).\overline{c}\texttt{in}_2\langle bz\rangle$, while $S' \overset{\text{def}}{=} !s(z).\overline{z}(c)!c(z').\overline{z'}\langle l\rangle$, $R' \overset{\text{def}}{=} !r(cz).\overline{c}(e)e(r').\overline{r'}\texttt{in}_1(f)f(x).\overline{z}\langle x\rangle$ and $W' \overset{\text{def}}{=} !w(cbz).\overline{c}(w)w(r).\overline{r}\texttt{in}_2\langle bz\rangle$. Then both $\mathsf{cell}\langle ul\rangle$ and $\mathsf{cell}'\langle ul\rangle$ are typable under $u{:}\texttt{Cell}[\mathbb{B}], l{:}\overline{\mathsf{ref}(\mathbb{B})}$, with $\mathsf{ref}(\tau) \overset{\text{def}}{=} [(\tau)^{\uparrow}\&\overline{\tau}()^{\uparrow}]^{!_{\mathrm{M}}}$.

To show these two typed processes are $\approx_{\forall\exists}$-equivalent, we first note that neither manipulates boolean values non-trivially (they are *data independent* in the sense of [15]), hence both are also typable under $u : \texttt{Cell}[\mathrm{Y}^{\forall}], l : \overline{\mathsf{ref}(\mathrm{Y}^{\forall})}$. By parametricity of $\approx_{\forall\exists}$, it suffices to consider a bisimulation under this typing, which radically reduces the number of transitions. We now construct a relation $\mathcal{R}$ from the following tuples:

$$\vdash\ (S \mid R \mid W \mid \Pi_i[c_i \to l]^{\mathsf{ref}(\mathrm{Y})})\ \ \mathcal{R}\ \ (S' \mid R' \mid W' \mid \Pi_i!c_i(z).\overline{z}\langle l\rangle)$$
$$\triangleright\ s : ((\mathrm{X}^{\exists})^{\uparrow})^{!_{\mathrm{M}}}, r : (\overline{\mathrm{X}}^{\exists}(\mathrm{Y}^{\forall})^{\uparrow})^{!_{\mathrm{M}}}, w : (\overline{\mathrm{X}}^{\exists}\,\overline{\mathrm{Y}}^{\forall}()^{\uparrow})^{!_{\mathrm{M}}},\ \vec{c}{:}\vec{\mathrm{X}}^{\exists}$$

together with their derivatives ([6] gives details). Note that $\Pi_i[c_i \to l]$ on the left-hand side is generated since S is in fact *not* allowed to do a free output via z (because l is not typed by a universal type variable; though it *is* typed by an existential variable). Observing that $c_i : \mathrm{X}_i^{\exists}$ prohibits each c_i from being used as the subject of an action, while permitting its use as an object of a free input (via r and w) that in turn triggers appropriate internal reduction, we can verify $\mathcal{R}$ is a bisimulation.

Fully Abstract Embedding of System F. Using the characterisation of polymorphic transitions by innocence mentioned in Section 5, we can embed System F (the second-order λ-calculus) fully abstractly in linear polymorphic processes. The contextual equality over λ-terms is defined in the standard way [20], using observables at the polymorphic boolean type. We write $M, N, \ldots$ for polymorphic λ-terms, $\alpha, \beta, \ldots$ for their types, and $\cong_{\forall}$ for the contextual equality. We can use different encodings to reach the same result: for example we can use Turner's call-by-value encoding [30] (other encodings, including those based on call-by-name, are discussed in [6]). The mapping of types becomes:

$$\alpha^{\bullet} \stackrel{\text{def}}{=} (\alpha^{\circ})^{\uparrow} \quad \mathrm{X}^{\circ} \stackrel{\text{def}}{=} \mathrm{X}^{!} \quad (\alpha \Rightarrow \beta)^{\circ} \stackrel{\text{def}}{=} (\overline{\alpha^{\circ}}\beta^{\bullet})^{!} \quad (\forall \mathrm{X}.\alpha)^{\circ} \stackrel{\text{def}}{=} \forall \mathrm{X}.((\alpha^{\circ})^{\uparrow})^{!}$$

Write $[\![M : \alpha]\!]_u$ for the encoding of a polymorphic λ-term $M : \alpha$. Then, setting $\cong_{\forall\exists}$ to be the contextual congruence over linear polymorphic processes discussed at the end of Section 4, we obtain:

Theorem 3. (full abstraction) *Let* $\vdash M_{1,2} : \alpha$. *Then* $M_1 \cong_{\forall} M_2 : \alpha$ *if and only if* $\vdash_{\mathrm{I}} [\![M_1 : \alpha]\!]_u \cong_{\forall\exists} [\![M_2 : \alpha]\!]_u \triangleright u : \alpha^{\circ}$.

The proof uses definability arguments based on innocence as in [7] (with additional treatment of contravariant universal types), see [6] for details.

References

1. Abadi, M., Cardelli, L., and Curien, P.-L. Formal parametric polymorphism. *TCS 121*, 1-2 (1993), 9–58.
2. Abramsky, S., Honda, K., and McCusker, G. Fully abstract game semantics for general reference. In *LICS'98* (1998), IEEE, pp. 334–344.
3. Abramsky, S., and Lenisa, M. Axiomatizing fully complete models for ML polymorphic types. In *Proc. of MFCS'2000* (2000).
4. Abramsky, S., and Lenisa, M. A fully-complete PER model for ML polymorphic types. In *Proc. of CSL'2000*, LNCS. Springer, 2000.
5. Berger, M. *Towards Abstractions for Distributed Systems.* PhD thesis, Imperial College, Department of Computing, 2002.
6. Berger, M., Honda, K., and Yoshida, N. Full version of this paper, available at `www.dcs.qmul.ac.uk/~{martinb,kohei}` and `www.doc.ic.ac.uk/~yoshida`.
7. Berger, M., Honda, K., and Yoshida, N. Sequentiality and the π-calculus. In *Proc. TLCA'01* (2001), no. 2044 in LNCS, Springer, pp. 29–45.
8. Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Universite de Paris VII, 1972.

9. Girard, J.-Y. Linear logic. *Theoretical Computer Science 50* (1987).
10. Grossman, D. Existential types for imperative languages. In *ESOP02* (2002), LNCS 2305, Springer, pp. 21–35.
11. Honda, K., and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS 221* (1999), 393–456.
12. Honda, K., and Yoshida, N. A uniform type structure for secure information flow. In *POPL'02: A full version as DOC Report 2002/13, Imperial College,* available at `www.dcs.qmul.ac.uk/~kohei` (2002).
13. Hughes, D. J. D. Games and definability for system F. In *LICS'97* (1997), IEEE Computer Society Press, pp. 76–86.
14. Hyland, J. M. E., and Ong, C. H. L. On full abstraction for PCF. *Information and Computation 163* (2000), 285–408.
15. Lazic, R., Newcomb, T., and Roscoe, A. On model checking data-independent systems with arrays without reset. Tech. Rep. RR-02-02, Oxford University, 2001.
16. Leroy, X. *Polymorphic typing of an algorithmic language.* PhD thesis, University of Paris, 1992.
17. Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, parts I and II. *Info. & Comp. 100*, 1 (1992), 1–77.
18. Milner, R., Tofte, M., and Harper, R. W. *The Definition of Standard ML.* MIT Press, 1990.
19. Mitchell, J. C. On the equivalence of data representation. In *Artificial Intelligence and Mathematical Theory of Computation* (1991).
20. Mitchell, J. C. *Foundations for Programming Languages.* MIT Press, 1996.
21. Murawski, A., and Ong, C.-H. L. Evolving games and essential nets for affine polymorphism. In *Proc. of TLCA'01* (2001), no. 2044 in LNCS, Springer.
22. Pierce, B., and Sangiorgi, D. Behavioral equivalence in the polymorphic pi-calculus. *Journal of ACM 47*, 3 (2000), 531–584.
23. Pierce, B. C., and Turner, D. N. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000.
24. Pitts, A., and Stark, I. Operational reasoning for functions with local state. In *HOOTS'98* (1998), CUP, pp. 227–273.
25. Pitts, A. M. Existential Types: Logical Relations and Operational Equivalence. In *Proceedings ICALP'98* (1998), no. 1443 in LNCS, Springer, pp. 309–326.
26. Pitts, A. M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10* (2000), 321–359.
27. Plotkin, G., and Abadi, M. A logic for parameteric polymorphism. In *LICS'98* (1998), IEEE Press, pp. 42–53.
28. Reynolds, J. C. Types, abstraction and parametric polymorphism. In *Information Processing 83* (1983), R. E. A. Mason, Ed.
29. Tofte, M. Type inefrence for polymorphic references. LNCS 2305, Springer, pp. 21–35.
30. Turner, D. N. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.
31. Vasconcelos, V. Typed concurrent objects. In *Proceedings of ECOOP'94* (1994), LNCS, Springer, pp. 100–117.
32. Yoshida, N., Berger, M., and Honda, K. Strong Normalisation in the π-Calculus. In *LICS'01* (2001), J. Halpern, Ed., IEEE, pp. 311–322.
33. Yoshida, N., Honda, K., and Berger, M. Linearity and bisimulation. In *FoSSaCs'02* (2002), vol. 2303 of *LNCS*, Springer, pp. 417–433.